

Try

C H A P T E R

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 the data type.
- 4.** Declare variables and constants.
- 5.** Select the appropriate scope for a variable.
- 6.** Convert text input to numeric values.
- 7.** Perform calculations using variables and constants.
- 8.** Convert between numeric data types using implicit and explicit conversions.
- 9.** Round decimal values using the `Decimal.Round` method.
- 10.** Format values for output using the `ToString` method.
- 11.** Use `Try/Catch` blocks for error handling.
- 12.** Display message boxes with error messages.
- 13.** 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 quantityInteger As Integer
Dim priceDecimal, extendedPriceDecimal As Decimal

' Convert input text to numeric and assign values to variables.
quantityInteger = Integer.Parse(Me.quantityTextBox.Text)
priceDecimal = Decimal.Parse(Me.priceTextBox.Text)

' Calculate the product.
extendedPriceDecimal = quantityInteger * priceDecimal
```

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 *maximumInteger*.

```
maximumInteger = 100
```

| |
|----------------|
| maximumInteger |
| 100 |

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

In the preceding example, the memory location called `maximumInteger` 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. An exception to this rule occurs when you initialize a variable on the same line as the declaration.

Here are some sample declaration statements:

```
' Declare a string variable.
Dim nameString As String

' Declare integer variables.
Dim counterInteger As Integer
Dim maxInteger As Integer = 100

' Declare a named constant.
Const DISCOUNT_RATE_Decimal As Decimal = .15D
```

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. The data types in VB .NET are actually classes and the variables are objects of the class. 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

Table 3.1

| 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 |

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 calculations; 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 `sumInteger`, `SumInteger`, `suminteger`, and `SUMINTEGER` all refer to the same variable.

Note: In some 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 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. *Include the class (data type) of the variable.*

3. Begin with a lowercase letter and then capitalize each successive word of the name. Always use mixed case for variables; uppercase for constants.

Sample Identifiers

| Field of Data | Possible Identifier |
|------------------------|----------------------------|
| Social Security number | socialSecurityNumberString |
| Pay rate | payRateDecimal |
| Hours worked | hoursWorkedDecimal |
| Phone number | phoneNumberString |
| Quantity | quantityInteger |
| Tax rate (constant) | TAX_RATE_Decimal |
| Quota (constant) | QUOTA_Integer |
| Population | populationLong |

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. subString |
| 2. #SoldInteger | 8. Text |
| 3. Number Sold Integer | 9. maximum |
| 4. Number.Sold.Integer | 10. minimumRateDecimal |
| 5. amount\$Decimal | 11. maximumCheckDecimal |
| 6. Sub | 12. companyNameString |

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 `MAXIMUM_PAY_Decimal` 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 that you include the data type in the name as well as the `As` clause that actually declares the data type. Use all uppercase for the name with individual words separated by underscores.

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

Const Statement—Examples

Examples

```
Const COMPANY_NAME_String As String = "R 'n R for Reading 'n Refreshment"  
Const COMPANY_ADDRESS_String As String = "101 S. Main Street"  
Const SALES_TAX_RATE_Decimal 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 |
| 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, inside a procedure you must use the `Dim` statement. Later in this chapter you will learn to declare variables outside of a procedure, using the `Public`, `Private`, or `Dim` statement.

Declaration Statements—General Form

```
Public|Private|Dim Identifier [As Datatype]
```

If you omit the optional data type, the variable’s type defaults to `Object`. It is recommended practice to *always* declare the data type.

Note: If `Option Strict` is `On`, you receive a syntax error if you omit the data type.

Declaration Statement—Examples

Examples

```
Dim customerNameString As String
Private totalSoldInteger As Integer
Dim temperatureSingle As Single
Dim priceDecimal As Decimal
Private costDecimal 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.

You also can declare several variables in one statement; the data type you declare in the `As` clause applies to all of the variables on the line. Separate the variable names with commas. Here are some sample declarations:

```
Dim nameString, addressString, phoneString As String
Dim priceDecimal, totalDecimal As Decimal
```

Entering Declaration Statements

Visual Basic’s IntelliSense feature helps you enter `Private`, `Public`, and `Dim` statements. After you type the space that follows `VariableName As`, a list pops up (Figure 3.1). This list shows the possible entries 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.

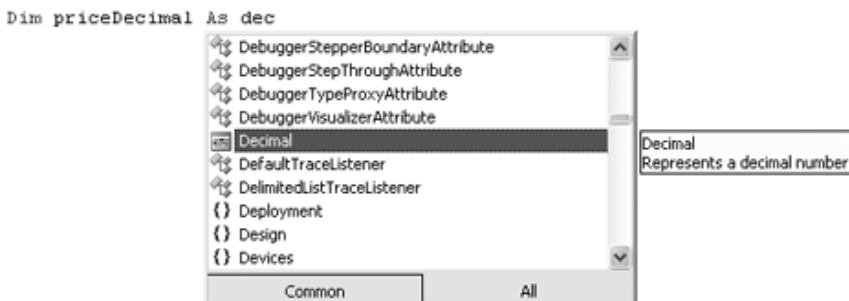
Figure 3.1

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.



Note: Some people find the IntelliSense feature annoying rather than helpful. You can turn off the feature by selecting *Tools / Options*. In the *Options* dialog box, make sure that *Show All Settings* is selected and choose *Text Editor / Basic / General*; deselect *Auto list members* and *Parameter information*.

Feedback 3.2

Write a declaration using the `Dim` statement for the following situations; 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 value.
 - (b) 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-level 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-level 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 declaration statement.

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

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 that it is known only to that procedure. Use the keyword `Dim` for local declarations. A `Dim` statement may 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 DISCOUNT_RATE_Decimal As Decimal = 0.15D

Private Sub calculateButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles calculateButton.Click)
    ' Calculate the price and discount.
    Dim quantityInteger As Integer
    Dim priceDecimal, extendedPriceDecimal, discountDecimal, discountedPriceDecimal _
        As Decimal

    ' Convert input values to numeric variables.
    quantityInteger = Integer.Parse(Me.quantityTextBox.Text)
    priceDecimal = Decimal.Parse(Me.priceTextBox.Text)

    ' Calculate values.
    extendedPriceDecimal = quantityInteger * priceDecimal
    discountDecimal = Decimal.Round( _
        (extendedPriceDecimal * DISCOUNT_RATE_Decimal), 2)
    discountedPriceDecimal = extendedPriceDecimal - discountDecimal
```

Notice the `Const` statement in the preceding example. Although you can declare named constants to be local, block, module level, or namespace in scope, just as you can variables, good programming practices dictate 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.

Note: A new feature of VB 2005 marks any variables that you declare but do not use with a gray squiggle underline. You can ignore the marks if you have just declared the variable and not yet written the code.

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. When you write module-level declarations, you can use the `Dim`, `Public`, or `Private` keyword. The preferred practice is to use either the `Public` or `Private` keyword for module-level variables rather than `Dim`. In Chapter 6 you will learn how and why to choose `Public` or `Private`. Until then we will declare all module-level variables using the `Private` keyword.

Place the declarations (`Private` 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.

```
(Declarations section)

Private ModuleLevelVariables
Const NamedConstants

Private Sub calculateButton_Click
Dim LocalVariables

...
End Sub

Private Sub summaryButton_Click
Dim LocalVariables

...
End Sub

Private Sub clearButton_Click
Dim LocalVariables

...
End Sub
```

Figure 3.3

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

```
' Declarations section of a form.

' Dimension module-level variables and constants.
Private quantitySumInteger, saleCountInteger As Integer
Private discountSumDecimal As Decimal
Const MAXIMUM_DISCOUNT_Decimal = 100.0D
```

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 `Private` and `Const` statements after the Class declaration but before your first procedure.

Figure 3.4

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

```
' Uses variables, constants, calculations, error
' handling, and a message box to the user.
'Folder: Ch03HandsOn

Public Class bookSaleForm

    ' Dimension module-level variables and constants.
    Private quantitySumInteger, saleCountInteger As Integer
    Private discountSumDecimal, discountedPriceSumDecimal As Decimal
    Const DISCOUNT_RATE_Decimal = 0.15D

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

Block-Level and Namespace-Level 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 can sometimes be useful when a project has multiple forms and/or modules, but good programming practices exclude the use of namespace-level variables.



If you use the `Me` object on all of your controls, it is easy to distinguish between controls and variables in your code. ■

Feedback 3.3

Write the declarations (`Dim`, `Private`, 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", cannot be used directly in calculations unless you first convert them to the correct data type.

Converting Strings to a Numeric Data Type

You can use a `Parse` method to convert the `Text` property of a control to its numeric form before you use the value in a calculation. The class 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 `Integer.Parse` method; to convert to a decimal value, use `Decimal.Parse`. Pass the text string that you want to convert as an **argument** of the `Parse` method.

```
' Convert input values to numeric variables.
quantityInteger = Integer.Parse(Me.quantityTextBox.Text)
priceDecimal = Decimal.Parse(Me.priceTextBox.Text)
```

```
' Calculate the extended price.
extendedPriceDecimal = quantityInteger * priceDecimal
```

Converting from one data type to another is sometimes called *casting*. In the preceding example, the `String` value from the `quantityTextBox.Text` property is cast into an `Integer` data type and the `String` from `priceTextBox.Text` is cast into a `Decimal` data type.

Using the Parse Methods

As you know, objects have methods that perform actions, such as the `Focus` method for a text box. The data types that you use to declare variables are

classes, which have properties and methods. Each of the numeric data type classes has a `Parse` method, which you will use to convert text strings into the correct numeric value for that type. The `Decimal` class has a `Parse` method that converts the value inside the parentheses to a decimal value while the `Integer` class has a `Parse` method to convert the value to an integer.

The Parse Methods—General Forms

General Forms

```
' Convert to Integer.  
Integer.Parse(StringToConvert)  
  
' Convert to Decimal  
Decimal.Parse(StringToConvert)
```

The expression you wish to convert can be the property of a control, a string variable, or a string constant. The `Parse` method returns (produces) a value that can be used as a part of a statement, such as the assignment statements in the following examples.

The Parse Methods—Examples

Examples

```
quantityInteger = Integer.Parse(Me.quantityTextBox.Text)  
priceDecimal = Decimal.Parse(Me.priceTextBox.Text)  
wholeNumberInteger = Integer.Parse(digitString)
```

The `Parse` methods examine the value stored in the argument and attempt to convert it to a number in a process called *parsing*, which means to pick apart, character by character, and convert to another format.

When a `Parse` method encounters a value that it cannot parse to a number, such as a blank or 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 `Integer.Parse` and `Decimal.Parse` methods for most of your programs. But in case you need to convert to `Long`, `Single`, or `Double`, `VB` also has a `Parse` method for each data type class.

Converting to String

When you assign a value to a variable, you must take care to assign like types. For example, you assign an integer value to an `Integer` variable and a decimal value to a `Decimal` variable. Any value that you assign to a `String` variable or the `Text` property of a control must be string. You can convert any of the numeric data types to a string value using the `ToString` method. Later in this chapter you will learn to format numbers for output using parameters of the `ToString` method.

Note: The rule about assigning only like types has some exceptions. See “Implicit Conversions” later in this chapter.

Examples

```
resultTextBox.Text = resultDecimal.ToString()  
countTextBox.Text = countInteger.ToString()  
idString = idInteger.ToString()
```

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 and truncating (dropping) any remainder. For example, if `totalMinutesInteger = 150`, then

```
hoursInteger = totalMinutesInteger \ 60
```

returns 2 for `hoursInteger`.

Mod

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

```
minutesInteger = totalMinutesInteger Mod 60
```

returns 30 for `minutesInteger`.

Exponentiation (`^`)

The exponentiation operator (`^`) raises a number to the power specified and returns (produces) a result of the `Double` data type. The following are examples of exponentiation.

```
squaredDouble = numberDecimal ^ 2        ' Square the number--Raise to the 2nd power.  
cubedDouble = numberDecimal ^ 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:

```
((score1Integer + score2Integer + score3Integer) / 3) * 1.2
```

Extra parentheses can always be used for clarity. The expressions

```
2 * costDecimal * rateDecimal
```

and

```
(2 * costDecimal) * rateDecimal
```

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.



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

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) |

Feedback 3.4

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

Assume that: xInteger = 2, yInteger = 4, zInteger = 3

1. xInteger + yInteger ^ 2
2. 8 / yInteger / xInteger
3. xInteger * (xInteger + 1)
4. xInteger * xInteger + 1
5. yInteger ^ xInteger + zInteger * 2
6. yInteger ^ (xInteger + zInteger) * 2
7. (yInteger ^ xInteger) + zInteger * 2
8. ((yInteger ^ xInteger) + zInteger) * 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

```
averageDecimal = sumDecimal / countInteger
amountDueLabel.Text = (priceDecimal - (priceDecimal * discountRateDecimal)).ToString()
Me.commissionTextBox.Text = (salesTotalDecimal * commissionRateDecimal).ToString()
```

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 text boxes. When you assign the result of a calculation to a Text property, you must place parentheses around the entire calculation and convert the result of the calculation to a string.

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 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 shortcut. The shortcuts allow you to type a variable name only once instead of having to type it on both sides of the equal sign.

For example, to add `salesDecimal` to `totalSalesDecimal`, the long version is

```
' Accumulate a total.  
totalSalesDecimal = totalSalesDecimal + salesDecimal
```

Instead you can use the shortcut assignment operator:

```
' Accumulate a total.  
totalSalesDecimal += salesDecimal
```

The two statements have the same effect.

To subtract 1 from a variable, the long version is

```
' Subtract 1 from a variable.  
countDownInteger = countDownInteger - 1
```

And the shortcut, using the `- =` operator:

```
' Subtract 1 from a variable.  
countDownInteger -= 1
```

The assignment operators that you will use most often are `+=` and `- =`. The following are examples of other assignment operators:

```
' Multiply resultInteger by 2 and assign the result to resultInteger.  
resultInteger *= 2  
  
' Divide sumDecimal by countInteger and assign the result to sumDecimal.  
sumDecimal /= countInteger  
  
' Concatenate smallString to the end of bigString.  
bigString &= smallString
```

Feedback 3.5

1. Write two statements to add 5 to `countInteger`, using (a) the standard, long version and (b) the assignment operator.
2. Write two statements to subtract `withdrawalDecimal` from `balanceDecimal`, using (a) the standard, long version and (b) the assignment operator.
3. Write two statements to multiply `priceDecimal` by `countInteger` and place the result into `priceDecimal`. Use (a) the standard, long version and (b) the assignment operator.

Option Explicit and Option Strict

Visual Basic provides two options that can significantly change the behavior of the editor and compiler. Not using these two options, **Option Explicit** and **Option Strict**, can make coding somewhat easier but provide opportunities for hard-to-find errors and very sloppy programming.

Option Explicit

When Option Explicit is turned off, you can use any variable name without first declaring it. The first time you use a variable name, VB allocates a new variable of Object data type. For example, you could write the line

```
Z = myTotal + 1
```

without first declaring either Z or myTotal. This is a throwback to very old versions of Basic that did not require variable declaration. In those days, programmers spent many hours debugging programs that had just a small misspelling or typo in a variable name.

You should always program with Option Explicit turned on. In VB .NET, the option is turned on by default for all new projects. If you need to turn it off (not a recommended practice), place the line

```
Option Explicit Off
```

before the first line of code in a file.

Option Strict

Option Strict is an option that makes VB more like other **strongly typed** languages, such as C++, Java, and C#. When Option Strict is turned on, the editor and compiler try to help you keep from making hard-to-find mistakes. Specifically, Option Strict does not allow any implicit (automatic) conversions from a wider data type to a narrower one, or between String and numeric data types.

All of the code you have seen so far in this text has been written with Option Strict turned on. With this option, you must convert to the desired data type from String or from a wider data type to a narrower type, such as from Decimal to Integer.

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

```
quantityInteger = Me.quantityTextBox.Text
```

and

```
amountInteger = amountLong
```

and

```
totalInteger += saleAmountDecimal
```

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. You can turn on Option Strict either in code or in the Project Designer. 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 myForm
```

To turn on Option Strict or Option Explicit for all files of a project, open the Project Designer by selecting *Project / Properties* or double-clicking My Project in the Solution Explorer. On the *Compile* tab you will find settings for both Option Explicit and Option Strict. By default, Option Explicit is turned on and Option Strict is turned off. Select *On* for Option Strict.

New to VB .NET 2005, you can set Option Strict on by default, which is better than setting it for each project: Select *Tools / Options / Project* and set the defaults.

Note: If *Show all settings* is checked in the *Options* dialog box, select *Projects and Solutions / VB Defaults* to find the default settings.

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.

Converting Between Numeric Data Types

In VB you can convert data from one numeric data type to another. Some conversions can be performed implicitly (automatically) and some you must specify explicitly. And some cannot be converted if the value would be lost in the conversion.

Implicit Conversions

If you are converting a value from a narrower data type to a wider type, where there is no danger of losing any precision, the conversion can be performed by an **implicit conversion**. For example, the statement

```
bigNumberDouble = smallNumberInteger
```

does not generate any error message, assuming that both variables are properly declared. The value of `smallNumberInteger` is successfully converted and stored in `bigNumberDouble`. However, to convert in the opposite direction could cause problems and cannot be done implicitly.

The following list shows selected data type conversions that can be performed implicitly in VB:

| From | To |
|---------|--|
| Byte | Short, Integer, Long, Single, Double, or Decimal |
| Short | Integer, Long, Single, Double, or Decimal |
| Integer | Long, Single, Double, or Decimal |
| Long | Single, Double, or Decimal |
| Decimal | Single, Double |
| Single | Double |

Notice that Double does not convert to any other type and you cannot convert implicitly from floating point (Single or Double) to Decimal.

Explicit Conversions

If you want to convert between data types that do not have implicit conversions, you must use an **explicit conversion**, also called **casting**. But beware: If you perform a conversion that causes significant digits to be lost, an exception is generated. (Exceptions are covered later in this chapter in the section titled “Handling Exceptions.”)

Use methods of the Convert class to convert between data types. The Convert class has methods that begin with “To” for each of the data types: ToDecimal, ToSingle, and ToDouble. However, you must specify the integer data types using their .NET class names rather than the VB data types.

| For the VB data type: | Use the method for the .NET data type: |
|-----------------------|--|
| Short | ToInt16 |
| Integer | ToInt32 |
| Long | ToInt64 |

The following are examples of explicit conversion. For each, assume that the variables are already declared following the textbook naming standards.

```
numberDecimal = Convert.ToDecimal(numberSingle)
valueInteger = Convert.ToInt32(valueDouble)
amountSingle = Convert.ToSingle(amountDecimal)
```

You should perform a conversion from a wider data type to a narrower one only when you know that the value will fit, without losing significant digits. Fractional values are rounded to fit into integer data types, and a single or double value converted to decimal is rounded to fit in 28 digits.

Performing Calculations with Unlike Data Types

When you perform calculations with unlike data types, VB performs the calculation using the wider data type. For example, `countInteger / numberDecimal` produces a decimal result. If you want to convert the result to a different

data type, you must perform a cast: `Convert.ToInt32(countInteger / numberDecimal)` or `Convert.ToSingle(countInteger / numberDecimal)`. Note, however, that VB does not convert to a different data type until it is necessary. The expression `countInteger / 2 * amountDecimal` is evaluated as integer division for `countInteger / 2`, producing an integer intermediate result; then the multiplication is performed on the integer and decimal value (`amountDecimal`), producing a decimal result.

Rounding Numbers

At times you may want to round decimal fractions. You can use the `Decimal.Round` method to round decimal values to the desired number of decimal positions.

The Round Method—General Form

General Form

```
Decimal.Round(DecimalValue, IntegerNumberOfDecimalPositions)
```

The `Decimal.Round` method returns a decimal result, rounded to the specified number of decimal positions, which can be an integer in the range 0–28.

The Round Method—Examples

Examples

```
' Round to two decimal positions.
resultDecimal = Decimal.Round(amountDecimal, 2)

' Round to zero decimal positions.
wholeDollarsDecimal = Decimal.Round(dollarsAndCentsDecimal, 0)

' Round the result of a calculation.
discountDecimal = Decimal.Round(extendedPriceDecimal * DISCOUNT_RATE_Decimal, 2)
```

The `Decimal.Round` method and the `Convert` methods round using a technique called “rounding toward even.” If the digit to the right of the final digit is exactly 5, the number is rounded so that the final digit is even.

Examples

| Decimal Value to Round | Number of Decimal Positions | Result |
|------------------------|-----------------------------|--------|
| 1.455 | 2 | 1.46 |
| 1.445 | 2 | 1.44 |
| 1.5 | 0 | 2 |
| 2.5 | 0 | 2 |

In addition to the `Decimal.Round` method, you can use the `Round` method of the `Math` class to round either decimal or double values. See Appendix B for the methods of the `Math` class.

Note: Visual Basic provides many functions for mathematical operations, financial calculations, and string manipulation. These functions can simplify many programming tasks. When Microsoft created Visual Basic .NET and moved to object-oriented programming, they made the decision to keep many functions from previous versions of VB, although the functions do not follow the OOP pattern of *object.method*. You can find many of these helpful functions in Appendix B. The authors of this text elected to consistently use OOP methods rather than mix methods and functions.

Formatting Data for Display

When you want to display numeric data in the Text property of a label or text box, you must first convert the value to string. You also can **format** the data for display, which controls the way the output looks. For example, 12 is just a number, but \$12.00 conveys more meaning for dollar amounts. Using the ToString method and formatting codes, you can choose to display a dollar sign, a percent sign, and commas. You also can 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.

If you use the ToString method with an empty argument, the method returns an unformatted string. This is perfectly acceptable when displaying integer values. For example, the following statement converts numberInteger to a string and displays it in displayTextBox.Text.

```
Me.displayTextBox.Text = numberInteger.ToString()
```

Using Format Specifier Codes

You can use the **format specifier** codes to format the display of output. These predefined codes can format a numeric value to have commas and dollar signs, if you wish.

Note: The default format of each of the formatting codes is based on the computer's regional setting. The formats presented here are for the default English (United States) values.

```
' Display as currency.  
Me.extendedPriceTextBox.Text = (quantityInteger * priceDecimal).ToString("C")
```

The “C” code specifies *currency*. By default, the string will be formatted with a dollar sign, commas separating each group of 3 digits, and 2 digits to the right of the decimal point.

```
' Display as numeric.  
Me.discountTextBox.Text = discountDecimal.ToString("N")
```

The “N” code stands for *number*. By default, the string will be formatted with commas separating each group of 3 digits, with 2 digits to the right of the decimal point.

You can specify the number of decimal positions by placing a numeric digit following the code. For example, “C0” displays as currency with zero digits to the right of the decimal point. The value is rounded to the specified number of decimal positions.

| Format Specifier Codes | Name | Description |
|------------------------|-------------|--|
| C or c | Currency | Formats with a dollar sign, commas, and 2 decimal places. Negative values are enclosed in parentheses. |
| F or f | Fixed-point | Formats as a string of numeric digits, no commas, 2 decimal places, and a minus sign at the left for negative values. |
| N or n | Number | Formats with commas, 2 decimal places, and a minus sign at the left for negative values. |
| D or d | Digits | Use only for <i>integer</i> data types. Formats with a left minus sign for negative values. Usually used to force a specified number of digits to display. |
| P or p | Percent | Multiplies the value by 100, adds a space and a percent sign, rounds to 2 decimal places; negative values have a minus sign at the left. |

Examples

| Variable | Value | Format Specifier Code | Output |
|----------------|-----------|-----------------------|------------|
| totalDecimal | 1125.6744 | "C" | \$1,125.67 |
| totalDecimal | 1125.6744 | "N" | 1,125.67 |
| totalDecimal | 1125.6744 | "N0" | 1,126 |
| balanceDecimal | 1125.6744 | "N3" | 1,125.674 |
| balanceDecimal | 1125.6744 | "F0" | 1126 |
| pinInteger | 123 | "D6" | 000123 |
| rateDecimal | 0.075 | "P" | 7.50 % |
| rateDecimal | 0.075 | "P3" | 7.500 % |
| rateDecimal | 0.075 | "P0" | 8 % |
| valueInteger | -10 | "C" | (\$10.00) |
| valueInteger | -10 | "N" | -10.00 |
| valueInteger | -10 | "D3" | -010 |

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

```
amountDecimal += chargesDecimal
Me.amountTextBox.Text = amountDecimal.ToString("C")
```

Assume that `amountDecimal` holds 1050 after the calculation and `amountTextBox.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 `amountDecimal`, not `amountTextBox.Text`. The variable `amountDecimal` holds a numeric value; `amountTextBox.Text` holds a string of (nonnumeric) characters.

You also can format `DateTime` values using format codes and the `ToString` method. Unlike the numeric format codes, the date codes are case sensitive. The strings returned are based on the computer's regional settings and can be changed. The following are default values for US-English in Windows XP.

| Date Specifier Code | Name | Description | Example of Default Setting |
|---------------------|-----------------------------|------------------------------------|----------------------------------|
| d | short date | Mm/dd/yyyy | 6/13/2005 |
| D | long date | Day, Month dd, yyyy | Monday, June 13, 2005 |
| t | short time | hh:mm AM PM | 4:55 PM |
| T | long time | hh:mm:ss AM PM | 4:55:45 PM |
| f | full date/time (short time) | Day, Month dd, yyyy hh:mm AM PM | Monday, June 13, 2005 4:55 PM |
| F | full date/time (long time) | Day, Month dd, yyyy hh:mm:ss AM PM | Monday, June 13, 2005 4:55:45 PM |
| g | general (short time) | Mm/dd/yyyy hh:mm AM PM | 6/13/2003 11:00 AM |
| G | general (long time) | Mm/dd/yyyy hh:mm:ss AM PM | 6/13/2003 11:00:15 AM |
| M or m | month | Month dd | June 13 |
| R or r | GMT pattern | Day, dd mmm yyyy hh:mm:ss GMT | Mon, 13 Jun 2005 11:00:15 GMT |

Note that you can also use methods of the `DateTime` structure for formatting dates: `ToLongDateString`, `ToShortDateString`, `ToLongTimeString`, `ToShortTimeString`. See Appendix B or MSDN for additional information.

Choosing the Controls for Program Output

Some programmers prefer to display program output in labels; others prefer text boxes. Both approaches have advantages, but whichever approach you use, you should clearly differentiate between (editable) input areas and (uneditable) output areas.

Users generally get clues about input and output fields from their color. By Windows convention, input text boxes have a white background; output text has

a gray background. The default background color of text boxes (`BackColor` property) is set to white; the default `BackColor` of labels is gray. However, you can change the `BackColor` property and the `BorderStyle` property of both text boxes and labels so that the two controls look very similar. You might wonder why a person would want to do that, but there are some very good reasons.

Using text boxes for output can provide some advantages: The controls do not disappear when the `Text` property is cleared and the borders and sizes of the output boxes can match those of the input boxes, making the form more visually uniform. Also, the user can select the text and copy it to another program using the Windows clipboard.

If you choose to display output in labels (the traditional approach), set the `AutoSize` property to `False` so that the label does not disappear when the `Text` property is blank. You also generally set the `BorderStyle` property of the labels to `Fixed3D` or `FixedSingle`, so that the outline of the label appears.

To use a text box for output, set its `ReadOnly` property to `True` (to prevent the user from attempting to edit the text) and set its `TabStop` property to `False`, so that the focus will not stop on that control when the user tabs from one control to the next. Notice that when you set `ReadOnly` to `true`, the `BackColor` property automatically changes to `Control`, which is the system default for labels.

The example programs in this chapter use text boxes, rather than labels, for output.

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 `averagePayDecimal` has a value of 123.456 and should display in a text box called `averagePayTextBox`.
2. The variable `quantityInteger`, which contains 176123, must be displayed in the text box called `quantityTextBox`.
3. The total amount collected in a fund drive is being accumulated in a variable called `totalCollectedDecimal`. What statement will display the variable in a text box called `totalTextBox` with commas and two decimal positions but no dollar signs?

A Calculation Programming Example

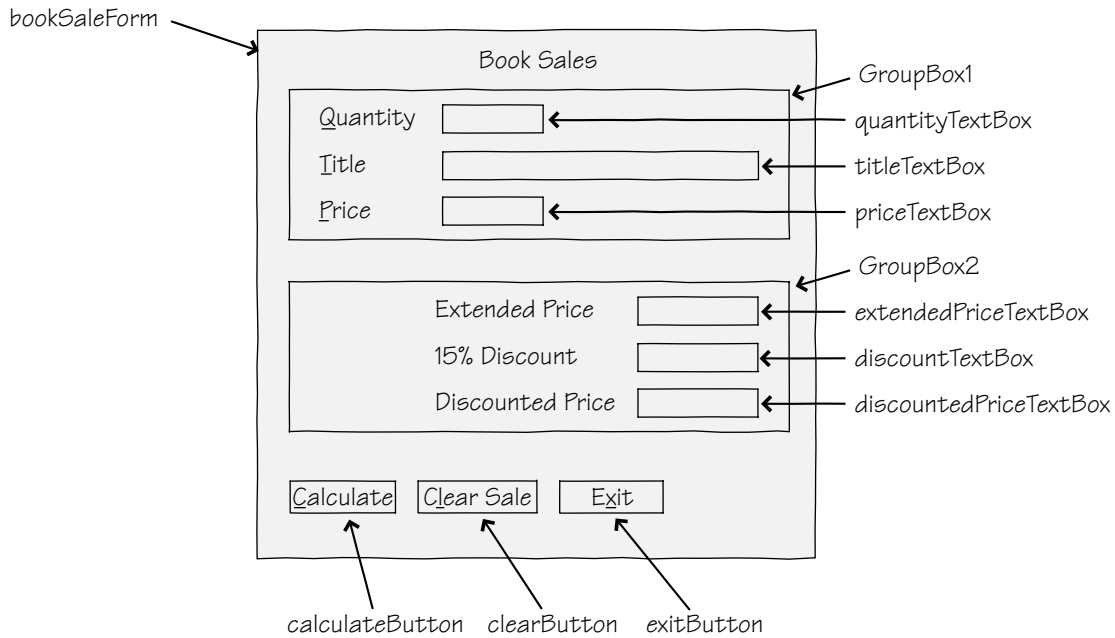
R'nR—for Reading'n Refreshment needs to calculate prices and discounts for books sold. The company is currently having a big sale, offering a 15 percent discount on all books. In this project you will calculate the amount due for a quantity of books, determine the 15 percent discount, and deduct the discount, giving the new amount due—the discounted amount. Use text boxes with the `ReadOnly` property set to `True` for the output fields.

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 |
|-----------------|--|---|
| bookSaleForm | Name Text AcceptButton CancelButton | bookSaleForm R 'n R for Reading 'n Refreshment calculateButton clearButton |
| Label1 | Text Font | Book Sales Bold, 12 point |
| GroupBox1 | Name Text | GroupBox1 (blank) |
| Label2 | Text | &Quantity |
| quantityTextBox | Name Text | quantityTextBox (blank) |
| Label3 | Text | &Title |
| titleTextBox | Name Text | titleTextBox (blank) |
| Label4 | Text | &Price |
| priceTextBox | Name Text | priceTextBox (blank) |

| Object | Property | Setting |
|------------------------|--|--|
| GroupBox2 | Name Text | GroupBox2 (blank) |
| Label5 | Text | Extended Price |
| extendedPriceTextBox | Name TextAlign ReadOnly TabStop | extendedPriceTextBox Right True False |
| Label6 | Text | 15% Discount |
| discountTextBox | Name TextAlign ReadOnly TabStop | discountTextBox Right True False |
| Label7 | Text | Discounted Price |
| discountedPriceTextBox | Name TextAlign ReadOnly TabStop | discountedPriceTextBox Right True False |
| calculateButton | Name Text | calculateButton &Calculate |
| clearButton | Name Text | clearButton C&lear Sale |
| exitButton | Name Text | exitButton 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 |
|-----------------------|--|
| calculateButton_Click | Dimension the variables. Convert the input Quantity and Price to numeric. Calculate Extended Price = Quantity * Price. Calculate and round: Discount = Extended Price * Discount Rate. Calculate Discounted Price = Extended Price – Discount. Format and display the output in text boxes. |
| clearButton_Click | Clear each text box. Set the focus in the first text box. |
| exitButton_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.

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

Figure 3.6

The form for the calculation programming example

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 BookSale
'Date:         June 2005
'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.
               Note that no error trapping is included in this version
               of the program.
'Folder:      Ch03BookSale
```

```
Public Class bookSaleForm
```

```
    ' Declare the constant.
    Const DISCOUNT_RATE_Decimal As Decimal = 0.15D
```

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

    Dim quantityInteger As Integer
    Dim priceDecimal, extendedPriceDecimal, discountDecimal, _
        discountedPriceDecimal As Decimal

    With Me
        ' Convert input values to numeric variables.
        quantityInteger = Integer.Parse(.quantityTextBox.Text)
        priceDecimal = Decimal.Parse(.priceTextBox.Text)

        ' Calculate values.
        extendedPriceDecimal = quantityInteger * priceDecimal
        discountDecimal = Decimal.Round( _
            (extendedPriceDecimal * DISCOUNT_RATE_Decimal), 2)
        discountedPriceDecimal = extendedPriceDecimal - discountDecimal

        ' Format and display answers.
        .extendedPriceTextBox.Text = extendedPriceDecimal.ToString("C")
        .discountTextBox.Text = discountDecimal.ToString("N")
        .discountedPriceTextBox.Text = discountedPriceDecimal.ToString("C")
    End With
End Sub

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

    With Me
        .titleTextBox.Clear()
        .priceTextBox.Clear()
        .extendedPriceTextBox.Clear()
        .discountTextBox.Clear()
        .discountedPriceTextBox.Clear()
        With .quantityTextBox
            .Clear()
            .Focus()
        End With
    End With
End Sub

Private Sub exitButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles exitButton.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 `Parse` methods, `Integer.Parse` and `Decimal.Parse`, 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 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, 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 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
    quantityInteger = Integer.Parse(Me.quantityTextBox.Text)
    Me.quantityTextBox.Text = quantityInteger.ToString()
Catch
    Me.messageLabel.Text = "Error in input data."
End Try
```

The Catch as it appears in the preceding example will catch any exception. You also can 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, use one of the predefined exception classes, which are all based on, or derived from, the SystemException 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 theException As FormatException
    Me.messageLabel.Text = "Error in input data."
```

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.

Common Exception Classes

Table 3.2

| Exception | Caused By |
|---|---|
| <code>FormatException</code> | Failure of a numeric conversion, such as <code>Integer.Parse</code> or <code>Decimal.Parse</code> . Usually blank or nonnumeric data. |
| <code>InvalidCastException</code> | Failure of a conversion operation. May be caused by loss of significant digits or an illegal conversion. |
| <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. |

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 theException As FormatException
    messageLabel.Text = "Error in input data: " & theException.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 theException As FormatException
    ' Statements for nonnumeric data.
Catch theException As ArithmeticException
    ' Statements for calculation problem.
Catch theException 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; each `Catch` represents a separate code block, so the variable's scope is only that block.

Later in this chapter in the “Testing Multiple Fields” section, you will see how to nest one `Try/Catch` block inside another one.

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, title bar text, and button(s) for the message box (Figure 3.7).

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 Forms

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.

General
Forms

```
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
    quantityInteger = Integer.Parse(Me.quantityTextBox.Text)
    Me.quantityTextBox.Text = quantityInteger.ToString()
Catch err As FormatException
    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 also may 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 Titlebar 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.

MessageBoxButtons

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.

MessageBoxIcon

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 <code>MessageBoxIcon</code> |
|---|
| 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 differently 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 21 possible forms of the argument list, or 21 signatures for the `Show` method. (We only showed 4 of the 21 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 14th 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 popup. 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).



TIP

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. ■

Figure 3.8

IntelliSense pops up the first of 21 signatures for the `Show` method. Use the up and down arrows to see the other possible argument lists.

1 of 21 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, displayHelpButton As Boolean) As System.Windows.Forms.DialogResult
text: The text to display in the message box.

Figure 3.9

Select the 14th 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 popup.

14 of 21 Show (**text As String**, caption As String) As System.Windows.Forms.DialogResult
text: The text to display in the message box.

Figure 3.10

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

1 of 10 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, displayHelpButton As Boolean) As System.Windows.Forms.DialogResult
caption: The text to display in the title bar of the message box.

Testing Multiple Fields

When you have more than one input field, each field presents an opportunity for an exception. If you would like your exception messages to indicate the field that caused the error, you can nest one Try/Catch block inside another one.

Nested Try/Catch Blocks

One Try/Catch block that is completely contained inside another one is called a **nested Try/Catch block**. You can nest another Try/Catch block within the Try block or the Catch block.

```
Try    ' Outer try block for first field.
      ' Convert first field to numeric .

      Try    ' Inner Try block for second field.
            ' Convert second field to numeric.

            ' Perform the calculations for the fields that passed conversion.

      Catch secondException As FormatException
        ' Handle any exceptions for the second field.

        ' Display a message and reset the focus for the second field.

      End Try ' End of inner Try block for second field.

      Catch firstException As FormatException
        ' Handle exceptions for first field.

        ' Display a message and reset the focus for the first field.

      Catch anyOtherException as Exception
        ' Handle any generic exceptions.

        ' Display a message.

End Try
```

You can nest the Try/Catch blocks as deeply as you need. Make sure to place the calculations within the most deeply nested Try; you do not want to perform the calculations unless all of the input values are converted without an exception.

By testing each Parse method individually, you can be specific about which field caused the error and set the focus back to the field in error. Also, by using the SelectAll method of the text box, you can make the text appear selected to aid the user. Here are the calculations from the earlier program, rewritten with nested Try/Catch blocks.

```
Private Sub calculateButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles calculateButton.Click
    ' Calculate the price and discount.
    Dim quantityInteger As Integer
    Dim priceDecimal, extendedPriceDecimal, discountDecimal, _
        discountedPriceDecimal, averageDiscountDecimal As Decimal
```

```

With Me
  Try
    ' Convert quantity to numeric variables.
    quantityInteger = Integer.Parse(.quantityTextBox.Text)
  Try
    ' Convert price if quantity was successful.
    priceDecimal = Decimal.Parse(.priceTextBox.Text)

    ' Calculate values for sale.
    extendedPriceDecimal = quantityInteger * priceDecimal
    discountDecimal = Decimal.Round( _
      (extendedPriceDecimal * DISCOUNT_RATE_Decimal), 2)
    discountedPriceDecimal = extendedPriceDecimal - discountDecimal

    ' Format and display answers for sale.
    .extendedPriceTextBox.Text = extendedPriceDecimal.ToString("C")
    .discountTextBox.Text = discountDecimal.ToString("N")
    .discountedPriceTextBox.Text = discountedPriceDecimal.ToString("C")

  Catch priceException As FormatException
    ' Handle price exception.
    MessageBox.Show("Price must be numeric.", "Data Entry Error", _
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
    With .priceTextBox
      .Focus()
      .SelectAll()
    End With
  End Try
  Catch quantityException As FormatException
    ' Handle quantity exception.
    MessageBox.Show("Quantity must be numeric.", "Data Entry Error", _
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
    With .quantityTextBox
      .Focus()
      .SelectAll()
    End With
  Catch anyException As Exception
    MessageBox.Show("Error: " & anyException.Message)
  End Try
End With
End Sub

```

Counting and Accumulating Sums

Programs often need to calculate the sum of 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 declare a module-level variable for the total. Then, in the `calculateButton_Click` event procedure for each sale, add the current amount to the total:

```
discountedPriceSumDecimal += discountedPriceDecimal
```

This assignment statement adds the current value for `discountedPriceDecimal` into the sum held in `discountedPriceSumDecimal`.

Counting

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

```
Private saleCountInteger as Integer
```

Then, in the `calculateButton_Click` event procedure, add 1 to the counter variable:

```
saleCountInteger += 1
```

This statement adds 1 to the current contents of `saleCountInteger`. The statement will execute one time for each time the `calculateButton_Click` event procedure executes. Therefore, `saleCountInteger` 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:

```
averageDiscountedSaleDecimal = discountedPriceSumDecimal / saleCountInteger
```

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.

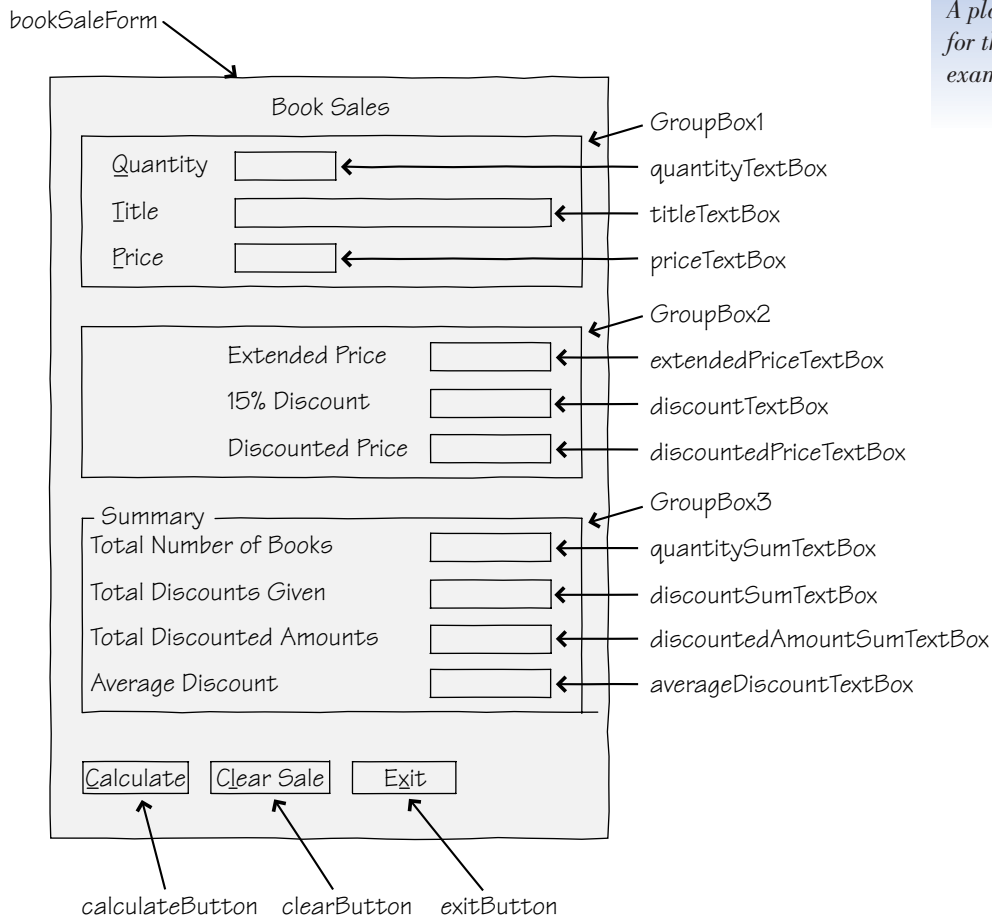
Add error handling to the program, so that missing or nonnumeric data will not cause a run-time error.

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 GroupBox3.

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

| Object | Property | Setting |
|--------------|--------------|-----------------------------------|
| bookSaleForm | Name | bookSaleForm |
| | Text | R 'n R—for Reading 'n Refreshment |
| | AcceptButton | calculateButton |
| | CancelButton | clearButton |
| Label1 | Text | Book Sales |
| | Font | Bold, 12 point |
| GroupBox1 | Name | GroupBox1 |
| | Text | (blank) |
| Label2 | Text | &Quantity |

| Object | Property | Setting |
|------------------------|---------------------------------------|--|
| quantityTextBox | Name Text | quantityTextBox (blank) |
| Label3 | Text | &Title |
| titleTextBox | Name Text | titleTextBox (blank) |
| Label4 | Text | &Price |
| priceTextBox | Name Text | priceTextBox (blank) |
| GroupBox2 | Name Text | GroupBox2 (blank) |
| Label5 | Text | Extended Price |
| extendedPriceTextBox | Name Text ReadOnly TextAlign | extendedPriceTextBox (blank) True Right |
| Label6 | Text | 15% Discount |
| discountTextBox | Name Text ReadOnly TextAlign | discountTextBox (blank) True Right |
| Label7 | Text | Discounted Price |
| discountedPriceTextBox | Name Text TextAlign ReadOnly | discountedPriceTextBox (blank) Right True |
| calculateButton | Name Text | calculateButton &Calculate |
| clearButton | Name Text | clearButton C&lear Sale |
| exitButton | Name Text | exitButton E&xit |
| GroupBox3 | Name Text | GroupBox3 Summary |
| Label8 | Text | Total Number of Books |
| quantitySumTextBox | Name Text ReadOnly TextAlign | quantitySumTextBox (blank) True Right |

| Object | Property | Setting |
|----------------------------|-----------|----------------------------|
| Label9 | Text | Total Discounts Given |
| discountSumTextBox | Name | discountSumTextBox |
| | Text | (blank) |
| | ReadOnly | True |
| | TextAlign | Right |
| Label10 | Text | Total Discounted Amounts |
| discountedAmountSumTextBox | Name | discountedAmountSumTextBox |
| | Text | (blank) |
| | ReadOnly | True |
| | TextAlign | Right |
| Label11 | Text | Average Discount |
| averageDiscountTextBox | Name | averageDiscountTextBox |
| | Text | (blank) |
| | ReadOnly | True |
| | TextAlign | Right |

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 `calculateButton_Click` event.

| Event Procedure | Actions-Pseudocode |
|------------------------------------|--|
| <code>calculateButton_Click</code> | Declare the variables. Try Convert the input Quantity to numeric. Try Convert the input 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. Format and display summary values. Catch any Price exception Display error message and reset the focus to Price. Catch any Quantity exception Display error message and reset the focus to Quantity. Catch any generic exception Display error message. |
| <code>clearButton_Click</code> | Clear each text box except Summary fields. Set the focus in the first text box. |
| <code>exitButton_Click</code> | 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. Test with nonnumeric data and blank entries.

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". The window contains a form titled "Book Sales". The form is organized into several sections:

- Input Section:** Three text boxes for "Quantity", "Title", and "Price".
- Calculated Section:** Three text boxes for "Extended Price", "15% Discount", and "Discounted Price".
- Summary Section:** Four text boxes for "Total Number of Books", "Total Discounts Given", "Total Discounted Amounts", and "Average Discount".
- Buttons:** Three buttons at the bottom: "Calculate", "Clear Sale", and "Exit".

The Project Coding Solution

```
'Project:      Chapter Example Totals and Exceptions
'Date:        June 2005
'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:      Ch03HandsOn
```

```
Public Class bookSaleForm
```

```
    ' Dimension module-level variables and constants.
    Private quantitySumInteger, saleCountInteger As Integer
    Private discountSumDecimal, discountedPriceSumDecimal As Decimal
    Const DISCOUNT_RATE_Decimal As Decimal = 0.15D
```

```
Private Sub calculateButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles calculateButton.Click
    ' Calculate the price and discount.
    Dim quantityInteger As Integer
    Dim priceDecimal, extendedPriceDecimal, discountDecimal, _
        discountedPriceDecimal, averageDiscountDecimal As Decimal
```

```
With Me
```

```
    Try
```

```
        ' Convert quantity to numeric variable.
```

```
        quantityInteger = Integer.Parse(.quantityTextBox.Text)
```

```
    Try
```

```
        ' Convert price if quantity was successful.
```

```
        priceDecimal = Decimal.Parse(.priceTextBox.Text)
```

```
        ' Calculate values for sale.
```

```
        extendedPriceDecimal = quantityInteger * priceDecimal
```

```
        discountDecimal = Decimal.Round(_
```

```
            (extendedPriceDecimal * DISCOUNT_RATE_Decimal), 2)
```

```
        discountedPriceDecimal = extendedPriceDecimal - discountDecimal
```

```
        ' Calculate summary values.
```

```
        quantitySumInteger += quantityInteger
```

```
        discountSumDecimal += discountDecimal
```

```
        discountedPriceSumDecimal += discountedPriceDecimal
```

```
        saleCountInteger += 1
```

```
        averageDiscountDecimal = discountSumDecimal / saleCountInteger
```

```
        ' Format and display answers for the sale.
```

```
        .extendedPriceTextBox.Text = extendedPriceDecimal.ToString("C")
```

```
        .discountTextBox.Text = discountDecimal.ToString("N")
```

```
        .discountedPriceTextBox.Text = discountedPriceDecimal.ToString("C")
```

```
        ' Format and display summary values.
```

```
        .quantitySumTextBox.Text = quantitySumInteger.ToString()
```

```
        .discountSumTextBox.Text = discountSumDecimal.ToString("C")
```

```
        .discountAmountSumTextBox.Text = discountedPriceSumDecimal.ToString("C")
```

```
        .averageDiscountTextBox.Text = averageDiscountDecimal.ToString("C")
```

```
Catch priceException As FormatException
```

```
    ' Handle a price exception.
```

```
    MessageBox.Show("Price must be numeric.", "Data Entry Error", _
```

```
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
```

```
    With .priceTextBox
```

```
        .Focus()
```

```
        .SelectAll()
```

```
    End With
```

```
End Try
```

```
    Catch quantityException As FormatException
        ' Handle a quantity exception.
        MessageBox.Show("Quantity must be numeric.", "Data Entry Error", _
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
        With .quantityTextBox
            .Focus()
            .SelectAll()
        End With

    Catch anException As Exception
        ' Handle any other exception.
        MessageBox.Show("Error: " & anException.Message)
    End Try
End With
End Sub

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

    With Me
        .titleTextBox.Clear()
        .priceTextBox.Clear()
        .extendedPriceTextBox.Clear()
        .discountTextBox.Clear()
        .discountedPriceTextBox.Clear()
        With .quantityTextBox
            .Clear()
            .Focus()
        End With
    End With
End Sub

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

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

Summary

1. Variables are temporary memory locations that have a name (called an *identifier*), a data type, and a scope. A constant also has a name, data type, and scope, but it also must have a value assigned to it. 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 assigned to a variable or constant. The most common data types are String, Integer, Decimal, Single, 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 the data type appended at the

end. Variable names should begin with a lowercase character and be mixed upper- and lowercase, while constants are all uppercase.

4. Intrinsic constants, such as `Color.Red` and `Color.Blue`, are predefined and built into the .NET Framework. 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 `Private` or `Dim` statement; the location of the statement determines the scope of the variable. Use the `Dim` statement to declare local variables inside a procedure; use the `Private` statement to declare module-level variables at the top of the program, outside of any procedure.
6. The scope of a variable may be namespace level, module level, local, or block level. Block-level 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-level 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 the data type of the variable or constant.
9. Use the `Parse` methods to convert text values to numeric before performing any calculations.
10. 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.
11. 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.
12. To explicitly convert between numeric data types, use the `Convert` class. Some conversions can be performed implicitly.
13. The `Decimal.Round` method rounds a decimal value to the specified number of decimal positions.
14. The `ToString` method can be used to specify the appearance of values for display. By using formatting codes, you can specify dollar signs, commas, percent signs, and the number of decimal digits to display. The method rounds values to fit the format.
15. `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.
16. A run-time error is called an *exception*; catching and taking care of exceptions is called *error trapping* and *error handling*.
17. You can trap for different types of errors by specifying the exception type on the `Catch` statement, and you 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.
18. A message box is a window for displaying information to the user.
19. The `Show` method of the `MessageBox` class is overloaded, which means that the method may be called with different argument lists, called signatures.
20. 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 | 106 | MessageBox | 126 |
| assignment operator | 110 | module-level variable | 103 |
| block-level variable | 103 | named constant | 97 |
| casting | 114 | namespace-level variable | 103 |
| constant | 96 | nested Try/Catch block | 129 |
| data type | 97 | Option Explicit | 111 |
| declaration | 97 | Option Strict | 111 |
| exception | 124 | order of precedence | 109 |
| explicit conversion | 114 | overloading | 128 |
| format | 116 | scope | 103 |
| format specifier | 116 | Show method | 126 |
| identifier | 97 | signature | 128 |
| implicit conversion | 113 | string literal | 100 |
| intrinsic constant | 101 | strongly typed | 112 |
| lifetime | 103 | Try/Catch block | 124 |
| local variable | 103 | variable | 96 |

Review Questions

1. Name and give the purpose of five data types available in Visual Basic.
2. What does *declaring a variable* mean?
3. What effect does the location of a declaration statement have on the variable it declares?
4. Explain the difference between a constant and a variable.
5. What is the purpose of the `Integer.Parse` method? The `Decimal.Parse` method?
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 `rateDecimal` display in `rateTextBox` as a percentage with three decimal digits.
9. What are implicit conversions? Explicit conversions? When would each be used?
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 carbohydrates, 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 text box. Use two other text boxes 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 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 text box, 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 text boxes.

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 text boxes. 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

| Beginning | Ending | Cost of Goods Sold | Average Inventory | Turnover |
|-----------|--------|--------------------|-------------------|----------|
| 58500 | 47000 | 400000 | \$52,750.00 | 7.6 |
| 75300 | 13600 | 515400 | 44,450.00 | 11.6 |
| 3000 | 19600 | 48000 | 11,300.00 | 4.2 |

- 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 formatted in a text box. 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 hour; divide that by 60 to get 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 text box formatted as currency.

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

| Amount | Rate | Years |
|---------|------|-------|
| 2000.00 | .15 | 5 |
| 1234.56 | .075 | 3 |

Check Figures**Future Value**

\$4,022.71

\$1,533.69

Hint: Remember that the result of an exponentiation operation is a Double data type.

- 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 text box 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 text box. Display a message to the user for any bad input data.

Calculation hint: There are 16 ounces in a pound.

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

- 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 text boxes 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 text boxes 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 hours worked, and the amount

of the store's total sales. The amount of sales needs to 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 form will allow the user to enter the salesperson'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 button 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 display 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 frame with boxes 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 boxes.

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.