©2010 Kendall Hunt Publishing

CHAPTER 3

Arithmetic Operators

LEARNING OBJECTIVES

After completing Chapter 3, students should be able to:

- ❑ Understand and use arithmetic operators in VB .NET
- ❑ Understand and interpret arithmetic expressions
- ❑ Convert string data to their numeric equivalents
- ❑ Convert numeric data to string form
- ❑ Format numeric output
- ❑ Understand the functionality of Option Explicit and Option Strict
- ❑ Use named constants
- ❑ Understand the application of module scope identifiers

3.1 Introduction

Besides working with textual data in a program such as getting the name entered in a TextBox, or displaying a message in a Label, programs often have to work with numeric data. Numeric data such as age, zip code, or a flight number need not necessarily be used in arithmetic. But more often than not, numeric data is used in some sort of calculation. For example, a program that computes the monthly payment on a loan uses the loan amount, the duration of the loan, and the interest rate applied to the loan in a formula. A program as such can compute the monthly payment as its output. Likewise, a program that computes the total electrical resistance in a parallel circuit with a few resistors uses the resistance of the resistors, in the formula given for computing the total resistance of a parallel circuit. Programming languages have a list of arithmetic operators that make such computations possible. Although all the languages have similar operators for the four basic arithmetic operators, i.e., addition, subtraction, division, and multiplication, each language has a set of arithmetic operators that must be studied when programming in that language.

©2010 Kendall Hunt Publishing

Table 3.1 VB .NET Arithmetic Operators

Operator	Meaning	Example
^	Power	2 ^ 5 results in 32
*	Multiplication	2 * 90 results in 180
/	Real division	15 / 4 results in 3.75
\	Integer division	10 \ 3 results in 3
Mod	Modulus	25 Mod 8 results in 1
+	Addition	12 + 6 results in 18
−	Subtraction	33 − 50 results in −17

3.2 Arithmetic Operators in VB .NET

There are seven arithmetic operators in VB .NET. All these operators are considered binary operators, meaning they operate on two numbers, which are referred to as Operands. Table 3.1 depicts the arithmetic operators in VB .NET in the order of their precedence. The operators' precedence rules will be discussed in the next section.

Let us have a closer look at these operators.

Power Operator (^)

VB .NET has a power operator (^) that operates on two operands. The operand on the left-hand side of the operator gets raised to the power of the operand on the right-hand side. For example, 5 ^ 2 raises 5 to the power of 2. Notice that the first operand (the one on the left-hand side) can be a whole number or a real number (a number with a fraction part). The second operand may be positive, or negative, a whole number, or a real number. For example, to take the square root of a number, one has to raise it to the power of 1/2 or 0.5. In this case, the number on the left-hand side cannot be a negative number. For example, 9 ^ 0.5 computes the square root of 9 which is 3.

Multiplication Operator (*)

The asterisk (*) is the operator for multiplication. The result of multiplication is the product of the two operands. The result may be a whole number or a real number, depending on the data type of the operands.

Division (/)

The forward slash is the operator for dividing two numbers. It divides the operand on the left-hand side of the operator by the one on the right-hand side. The result of this operation is a real number of type Double. For example, 17/2 results in 8.5. Each operand may be a whole number or a real number.

©2010 Kendall Hunt Publishing

Integer Division (\)

The backward slash is the operator for Integer Division. The outcome of Integer Division is the number of times the Operand on the right-hand side goes into the operand on the left-hand side. For example, 22\5 results in 4. The outcome of an Integer Division is always a whole number. If either or both of the operands were real numbers, VB .NET implicitly rounds them up or down to whole numbers, and then performs the Integer Division. To round a real number means replacing the number with the closest Integer value. For example, 2.67 gets rounded up to 3, whereas 2.25 gets rounded down to 2.

Modulus Operator (Mod)

The Mod Operator (pronounced modulus) is used to find the remainder of a division. The outcome of the Mod is the remainder of dividing the operand on the left-hand side of the operator by the one on the right-hand side. For example, 14 Mod 5 results in 4 which is the remainder of dividing 14 by 5 and 12.25 Mod 3 results in 0.25. One has to realize that, if the operand on the left-hand side is smaller than the one on the right, the remainder of division will be the operand on the left-hand side. For example, 4 Mod 10 will result in 4.

Addition and Subtraction Operators (+, −)

The addition and subtraction operators are self-explanatory. The result of adding or subtracting two numbers may be a whole number or a real number. For example, 4 + 6 results in 10, 1.3 + 2.7 results in 4, and 2.50 + 5.25 results in 7.75.

3.3 Arithmetic Expression

Any meaningful combination of numeric literals (e.g., 5), variables, and arithmetic operators form an arithmetic expression. An arithmetic expression is not a complete statement; therefore, it has to be part of a statement. The following are the examples of arithmetic expressions. Assume that variable Data has been declared as Integer with a value stored in it.

```
3 + 4 * 120
Data * 0.06 + 100
(12 - Data)\8 + 10
```

Precedence Rules

An arithmetic expression is evaluated from left to right, respecting the precedence order of operators, similar to algebraic expressions. Arithmetic operators have different levels of precedence. To evaluate an arithmetic expression, one has to know the precedence order of the operators. The precedence order indicates which operator takes precedence over the other operators in the order of operations. The operator with higher precedence operates before the operator with lower precedence. For example, 3 + 4 * 2 gets evaluated from left to right, but since * has higher precedence than +, multiplication takes place before addition, hence this expression evaluates to 11. If there are several operators with the same precedence in an expression, then VB .NET evaluates the expression from left to right. The only exception to the precedence rules are parentheses. Any expression enclosed in parentheses must be evaluated first. If there are several levels of nested parentheses, the expression in the innermost parentheses must be evaluated before the ones surrounding it, and so on.

©2010 Kendall Hunt Publishing

The precedence of arithmetic operators in VB .NET from highest to lowest:

1. Expression enclosed in parentheses ( ) has the highest precedence.
2. Power operator: ^
3. Unary minus: −
4. Multiplication and real division: *, /
5. Integer division: \
6. Modulus operator: Mod
7. Addition and subtraction: +, −
8. Assignment operator '=', has lower precedence than all the arithmetic operators.

There is one operator in the box above that was not explained among the arithmetic operators: the unary minus, which operates on one operand only. It can be placed on the left-hand side of a numeric literal (e.g. −6) or on the left-hand side of a numeric variable (e.g. −*N*), to represent a negative value, or to reverse the sign of a number. It is interesting to note that the unary minus has lower precedence than the power operator; therefore, −5 ^ 2 results in −25.

Let us evaluate the following arithmetic expressions:

Ex1: `3 + 5 * 3` results in `18`

$$\begin{tikzcd}
3 \arrow[dddr] & & 5 \arrow[d] & 3 \arrow[dl] \\
& & \ast & \\
& & 15 \arrow[dl] & \\
& + & & \\
& 18 & &
\end{tikzcd}$$

Ex2: `12 * 4` Mod `10` results in `8`

$$\begin{tikzcd}
12 \arrow[d] & 4 \arrow[dl] & & 10 \arrow[dddll] \\
\ast & & & \\
48 \arrow[dr] & & & \\
& \text{Mod} & & \\
& 8 & &
\end{tikzcd}$$

Ex3: `−10 ^ 2` results in `−100`

$$\begin{tikzcd}
- \arrow[dddr] & 10 \arrow[d] & 2 \arrow[dl] \\
& \text{\textasciicircum} & \\
& 100 \arrow[d] & \\
& -100 &
\end{tikzcd}$$

©2010 Kendall Hunt Publishing

Arithmetic operators with the same precedence get evaluated from left to right.

$$\begin{tikzcd}[column sep=small]
\text{Ex4: } 4.90 \arrow[dr] & + & 8 \arrow[dl] & - & 2 \arrow[ddl] & \text{results in } 10.90 \\
& + \; 12.90 \arrow[dr] & & & & \\
& & & - \; 10.90 & &
\end{tikzcd}$$

3.4 Arithmetic in VB .NET

An arithmetic expression is not a complete statement; it has to be part of a program statement. For example, one may store the result of an arithmetic expression in a variable. The following assignment statement stores the outcome of an arithmetic expression, which is 5.80, in variable Result.

```
Dim Result As Double

Result = (12 + 2.5) * 4 / 10
```

Lvalue and Rvalue

It is very important to understand the way the computer views a variable, when the variable name appears on the left-hand side, or on the right-hand side of an assignment operator.

Lvalue

When a variable's name appears on the left-hand side of an assignment operator, it represents an Lvalue or a location in the computer's random access memory (RAM). In other words, the Lvalue is an address in the computer's memory, where a value of proper data type can be stored. In the example below, variable Count is declared as Integer, and is initialized to some value. Notice that in the assignment statement, the variable's name appears on the left-hand side of the assignment operator; hence it represents a location in the computer's RAM, where a value can be stored:

```
Dim Count As Integer

Count = 20
```

Rvalue

When the variable's name appears on the right-hand side of an assignment operator, it represents an Rvalue, also known as the register value. It provides the value stored in the variable in the computer's RAM. Consider the following example, in which the variable Count from the previous example is being used.

```
Dim Price As Decimal

Price = Count * 1.5
```

In this example, the variable Count on the right-hand side of the assignment operator represents the value stored in Count, which is 20. It is multiplied by 1.5 and the result 30 is stored in variable Price. Note

©2010 Kendall Hunt Publishing

that, since variable Price is on the left-hand side of the assignment operator, it represents an Lvalue or an address in the computer's RAM.

Increment–Decrement a Variable

Consider the variable Count declared in the previous example. To increment the value stored in Count by 1, one has to write the following statement:

```
Count = Count + 1
```

It is important *not* to think of this statement as an algebraic equation. The variable Count on the right-hand side of the assignment operator represents the value stored in Count, which is 20. Since the assignment operator has lower precedence than the addition operator, the value in Count is added to 1 and then assigned to variable Count on the left-hand side of the assignment operator. Therefore, after this statement, the value stored in Count will be changed to 21. You can use the same logic to increment or decrement a variable by any value. For example, the following statement subtracts 5 from variable Count, changing the value stored in Count to 16:

```
Count = Count - 5
```

What Happens When Operands Are of Different Data Types?

The result of an arithmetic expression will be of the data type that is more precise or that consumes more bytes in the memory. For example, if one is adding an Integer and a Short, the result will be an Integer. Multiplying variables of types Decimal and Integer will result in Decimal. VB .NET does what is referred to as implicit-type conversion when assigning a data value of smaller (number of bytes) data type to a wider data type and vice versa. Implicit conversion is performed internally by VB .NET. Consider the following examples:

```
Dim Result As Double
Result = 500                 '500 is converted to Double and stored in Result
Dim Count As Integer
Count = 10 / 3               '3.33333 is rounded down to 3 and stored in Count.
Dim N As Integer = 12.78     '12.78 is rounded up to 13 and stored in N
```

3.5 Combined Assignment Operators

Combined assignment operators can be used when the variable's name appears on both sides of the assignment operator. This syntax is also available in other languages such as C and C++. The combined assignment operators exist for all the arithmetic operators: +=, −=, *=, /=, \=, Mod=, ^=. For example, consider the variable Number declared below. All the statements may be written using the combined assignment operators:

```
Dim Number As Integer
Number = 5
Number = Number + 2          Can be written as:     Number += 2
Number = Number - 6          Can be written as:     Number -= 6
Number = Number ^ 3          Can be written as:     Number ^= 3
Number = Number * (12\5)     Can be written as:     Number *= 12\5
```

©2010 Kendall Hunt Publishing

Option Explicit On

Option Explicit means that every variable must be declared before it can be used in the program. In VB .NET, Option Explicit is always *on* by default.

Option Strict On

Option Strict enforces the type compatibility when storing data in a variable, meaning the data value on the right-hand side of the assignment operator must be of the same data type as the variable on the left-hand side. One may turn this option on by typing Option Strict On at the beginning of the code window before the Form's generated code. Option Strict checks for the type compatibility and does not allow implicit type conversion between different data types.

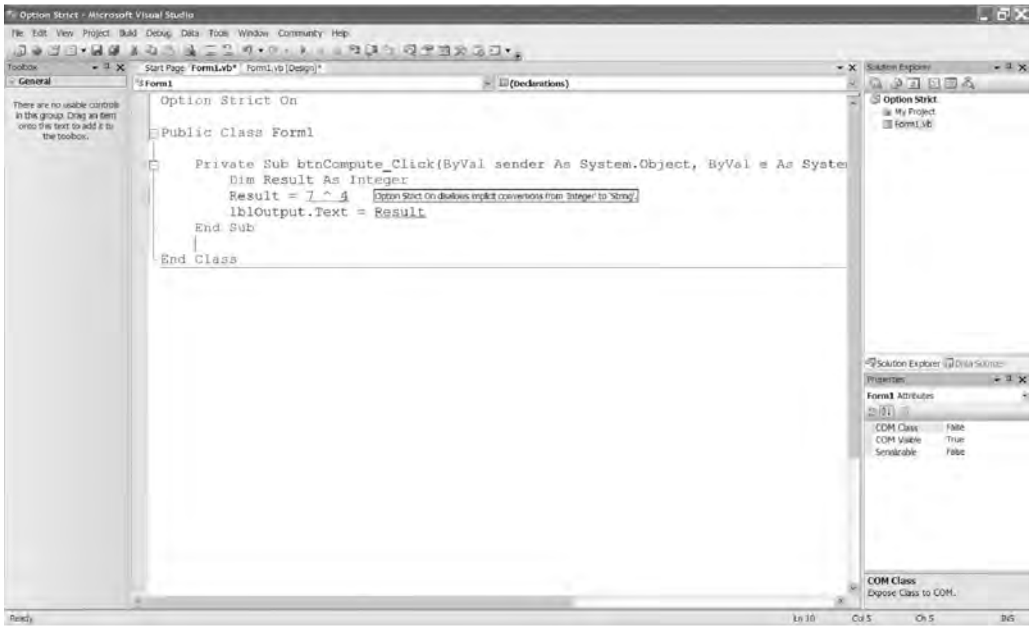
In the example shown in Figure 3.1 the Option Strict On has been typed at the beginning of the code window, hence the program generates two compile errors. In the first statement, the outcome of 7 ^ 4 is of type Double and cannot be stored in an Integer variable; in the second statement, an Integer variable is being assigned to the Text property of the Label. In both statements, the type mismatch is causing the compile error. To fix these errors either remove the Option Strict On, or use explicit-type conversion. By removing the Option Strict On, VB .NET does an implicit type conversion and the program generates no errors.

3.6 Explicit-type Conversion

Whether Option Explicit is on or off, it is a good practice to do an explicit type conversion when mixing the numeric data with the string data.

Converting a String to Its Numeric Equivalent

In a VB .NET program, you often need to get the user's input from a TextBox. The data entered in a TextBox at runtime is stored in the Text property of the TextBox; hence, it is of String data type. Even



```
Option Strict On

Public Class Form1

    Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As Syste
        Dim Result As Integer
        Result = 7 ^ 4    Option Strict On disallows implicit conversions from 'Integer' to 'String'.
        lblOutput.Text = Result
    End Sub

End Class
```

Option Strict - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Toolbox
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
Start Page Form1.vb* Form1.vb [Design]*
Form1
(Declarations)
Solution Explorer
Option Strict
My Project
Form1.vb
Properties
Form1 Attributes
COM Class False
COM Visible True
Serializable False
COM Class
Expose Class to COM.
Ready

Figure 3.1 ***Option Strict On***

©2010 Kendall Hunt Publishing

if the data entered in the TextBox is a number, it will be in string form, e.g., "25". To retrieve the numeric data entered in the TextBox, one has to store it in a numeric variable, and then use the variable in calculations. To store the numeric data entered in a TextBox in a numeric variable, you have to convert it from the string form, i.e., "25" to its numeric equivalent, i.e. 25. This is made possible by the Parse method.

Parse ( ) Method

All the data types in VB .NET are classes. Each data type has a Parse method. The Parse method takes a numeric string as an input and changes it into the number of the specified data type. Notice that the Parse method cannot parse a non-numeric string such as *"Bell"* to a number. An attempt to parse such a string to a numeric value will result in a runtime error, which causes the program to halt. This method is useful when converting a numeric string to its numeric equivalent.

Assume that the user enters the number of items purchased in the TextBox, txtItems. The following statement stores the entered data in the Integer variable Count declared below:

```
Dim Count As Integer
Count = Integer.Parse(txtItems.Text)
```

Converting the Numeric Data to String

To display numeric data in a Label, the data has to be stored in the Text property of the Label; therefore, it has to be converted to string form. Every numeric data type has a method called ToString which can be used to convert numeric data, e.g., 34 to its string equivalent, i.e., "34".

ToString( ) Method

All the numeric data types have the ToString() method. This method converts numeric data to its string equivalent.

Assume that the monthly payment on a loan has been computed and stored in variable Payment of type Decimal. The following statements display this variable in the Label, lblOutput:

```
Dim Payment As Decimal
Payment = 120.9
lblOutput.Text = Payment.ToString()
```

3.7 Formatting the Numeric Output

ToString() method can be used with or without an input argument. Without an input argument, it converts the numeric data to string form without any changes in the appearance of the data. However, a lot of times programmers want the converted string in a specific format, for example, with no digits after the decimal point, with 4 digits after the decimal point, etc. ToString() method accepts a string argument that specifies how the converted string should be formatted. Table 3.2 displays few of these format strings. Assume the following declaration. The value stored in the variable Num can be formatted and displayed as shown in Table 3.2.

```
Dim Num As Decimal = 1200
```

©2010 Kendall Hunt Publishing

TABLE 3.2 Format strings

FORMAT STRING "?"	EXAMPLE	OUTCOME STRING
C or c	Num.Tostring ("c")	$1,200.00
N or n	Num.ToString ("n")	1,200.00
Nx or nx	Num.ToString ("n3")	1,200.000
P or p, percent	0.05.ToString("P")	5.00 %

3.8 Translating Formulas to VB .NET

Often, you have to use a formula to solve a given problem, such as computing the volume of a sphere, or the monthly payment on a loan. For example, to compute the volume of a sphere with a given radius, you have to use the formula $V = 4/3\pi R^3$. To translate this formula to program statement(s), one has to declare variables to store the radius and the volume of the sphere, and then store a value in the radius. Finally, those variables have to be used in the formula to calculate the volume. In the example below, Radius is initialized to 1, and the volume is computed and stored in the variable Volume:

```
Dim Radius As Double
Dim Volume As Double
Radius = 1
Volume = 4/3 * 3.14 * Radius ^ 3
```

As another example, let us look at a generic formula that does some calculation:

$$Z = \frac{(a + b)^{-n}}{b}$$

To translate this formula to VB .NET, one has to declare variables to represent *a, b, n,* and *Z,* and use them in the formula. Since this is a generic formula, variables are declared with generic names that match the ones given in the formula.

```
Dim a As Integer
Dim b As Integer
Dim n As Integer
Dim Z As Double
```

Store some values in *a*, *b*, and *n*. Then use them in formula.

```
Z = (a + b) ^ −n / b
```

Tip Additional parentheses may be added to improve the clarity of the operations.

Z = ((a + b) ^ (– n)) / b

EXAMPLE 1

Design a VB .NET project to compute the volume of a cylinder. Allow the user to enter values for the radius and height on the Form, and have the program compute and display the volume. The volume of a cylinder is computed by multiplying the area of the base by the height of the cylinder, as given in the following formula:

Volume $= \pi R^2 H$

Where R is the radius of the base, and H is the height.

©2010 Kendall Hunt Publishing

Let us go through the PDLC (Program Development Life Cycle)

Step 1: ANALYZE THE PROBLEM:

- Input needs: radius and height
- Output needs: calculated volume
- Processing needs: Compute the volume

Step 2: DESIGN THE GUI: Based on step 1, there should be two TextBoxes, to enter input data, one Label to display the volume, and a Button to compute the volume. Let us name the objects as: txtRadius, txtHeight, lblOutput, and btnCompute. Besides these objects, there should be few descriptive Labels to describe other objects. There is no need to name descriptive Labels. Change the Text property of each object to show its purpose on the Form. Figure 3.2 shows the GUI designed for this example.

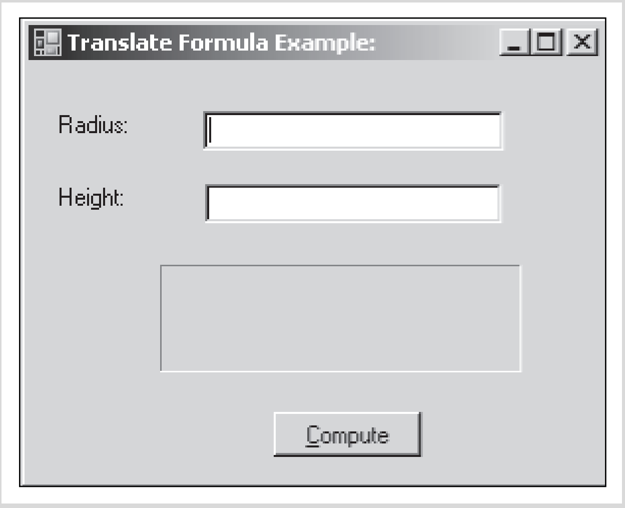


Figure 3.2 ***Compute the volume***

Step 3: DETERMINE WHICH OBJECTS ON THE FORM SHOULD RESPOND TO EVENTS.

Only the Button should respond to the click event.

Step 4: DESIGN AN ALGORITHM (LOGICAL SOLUTION) FOR THE EVENT PROCEDURE. **The pseudocode of the click event procedure of btnCompute:**

1. Declare variables to store the entered data.
2. Declare a variable to store the output.
3. Store the entered data in variables.
4. Plug in the variables into the formula given for volume, and store the result of calculation in the output variable.
5. Display the volume in the Label.

Step 5: TRANSLATE THE ALGORITHM DESIGNED IN STEP 4 TO VB .NET STATEMENTS. Allow the user to enter data values with a fraction part, by declaring the variables as Single or Double data type. Below is the translation of the given pseudocode to VB .NET code.

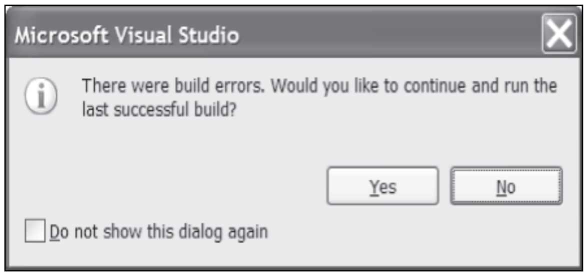
```
Private Sub btnCompute_Click (ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnCompute.Click
        Dim Radius As Single
        Dim Height As Single
        Dim Volume As Single
        Radius = Single.Parse(txtRadius.Text)
        Height = Single.Parse(txtHeight.Text)
        Volume = 3.14 * Radius ^ 2 * Height

        lblOutput.Text = “ Volume = “ & Volume.ToString(“N”)
    End Sub
End Class
```

©2010 Kendall Hunt Publishing

STEP 6: DEBUG AND TEST THE PROGRAM: The last step of the PDLC entails debugging the program and testing it by entering different input values. There are tools in the VB .NET development environment to compile, execute, and debug the program (refer to the Appendix A on Debugging). There are several ways to compile and execute a VB .NET program: by clicking on the F5 key on the keyboard, by clicking on the run icon (>) in the tool bar, or by choosing Debug on the menu bar, and Start Debugging from the submenu. Any of the ways will compile and execute the program. If there are compile errors in the program, a dialog Form shown in Figure 3.3 gets displayed, informing the user of the compile/build errors. Always close this dialog Form by clicking on the No Button. Fix the compile errors, and run the program again, until there is no build error in the program.

You should always test all the code in the program by entering varying sample inputs, and clicking on each Button. Verify the program's output with the expected results for the sample input data. If the program generates the right answer for all the sample data, one may assume that the program is working properly. Figure 3.4, is the screen capture of this project at runtime.


Microsoft Visual Studio

There were build errors. Would you like to continue and run the last successful build?

Figure 3.3 ***Build box***

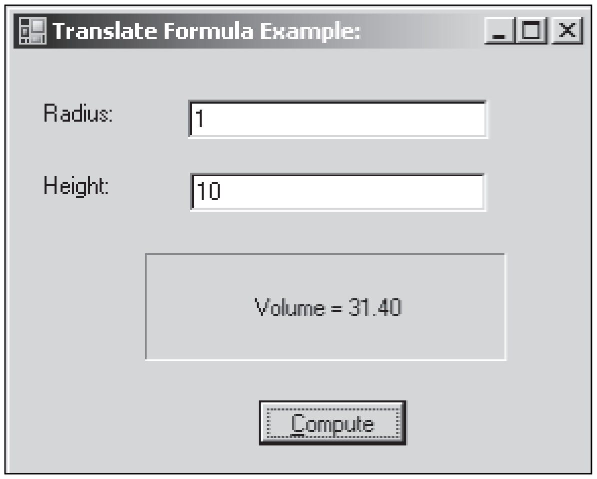


Figure 3.4 ***The project at runtime***

3.9 Named Constants

A named constant is an identifier in the program that is used to give a name to an important constant such as the tax rate = 0.08, or pi = 3.14. The purpose of a named constant is to be able to store and label a constant value that should not be changed during the program execution. For example, instead of using 0.0625 in the code for tax rate, one can name it as cTaxRate, and use that name in the code.

Declaration Syntax:

To use a named constant, it has to be declared in the program with the keyword Const. A value should be assigned to the named constant at the time of declaration. This value can not be changed later on in the program. A named constant may be declared with local scope, module scope or project scope. The general declaration syntax is:

```
[Private] Const  Identifier As DataType = Value
```

The keyword Private is used when declaring a named constant with module scope. It should be omitted when declaring the constant inside an event procedure with local scope. Module scope identifiers are

©2010 Kendall Hunt Publishing

explained at the end of this chapter. It is recommended to precede the name of a named constant with a lowercase *c*.

Examples:

Declare a local scope named constant to store the sales tax in Indiana.

```
Const cTaxRate As Double = 0.07
```

Declare a module scope named constant to store the name of the university.

```
Private Const cSchool As String = "Purdue University"
```

Advantages of Using Named Constants in a Program:

- Adds to the clarity of the program.
- It is easier to maintain, and modify the program, if the value of the constant changes in future. For example, if the tax rate changes in a few years, the programmer can update the program, by changing one single line of code.
- It protects the constant value from accidental changes in the code.
- It prevents the programmer from using any erroneous value in place of the constant. For example, the programmer might enter 0.6 for tax rate instead of 0.06.

Tip A named constant is often declared with module scope, making it accessible to the entire code written for the Form.

EXAMPLE 2

Example 1 is a good candidate for using a named constant. Let us declare the value of 3.14159 as a named constant, and use it in the click event procedure of the Button. Below is the entire code window of Example 1, modified using a named constant in the code.

```
Public Class Form1
    Private Const cPI As Double = 3.14159
    Private Sub btnCompute_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnCompute.Click
        Dim Radius As Single
        Dim Height As Single
        Dim Volume As Single
        Radius = Single.Parse(txtRadius.Text)
        Height = Single.Parse(txtHeight.Text)
        Volume = cPI * Radius ^ 2 * Height
        lblOutput.Text = " Volume = " & Volume.ToString("N")
    End Sub
End Class
```

©2010 Kendall Hunt Publishing

EXAMPLE 3

Develop a VB .NET program to be used as a simple calculator. At execution time, the user should enter two numeric values (whole or real) in provided TextBoxes, and click on the desired operator. The program should then display the outcome of the operation in the output Label, and display the symbol of the picked operator in the provided Label.

Let us go through the PDLC (Program Development Life Cycle):

Step 1: Analyze the problem:

- Input needs: two numbers
- Output needs:
  a. The arithmetic operator used
  b. The result of the operation
- Processing needs: Perform operations: add, subtract, multiply, divide, power, and modulus.

Step 2: Design the GUI: Based on step 1, there should be two TextBoxes to enter the input data. Let us name these objects txtNum1 and txtNum2. Based on the output needs, there should be two Labels. Let us name these Labels lblResult and lblOperator. The processing needs requires seven Buttons, one Button per operation. Let us name these objects: btnAdd, btnSubtract, btnProduct, and so on. Change the Text property of all the objects placed on the Form to convey their objectives on the Form. For example, change the Text property of the btnAdd to "+." Following the GUI standards explained in Appendix B of this book, one may come up with the GUI shown in Figure 3.5. The descriptive Labels are used to describe the purpose of other objects on the Form. There is no need to rename descriptive Labels. The small box between the two TextBoxes is a Label with the BackColor property changed to white for a better GUI design.

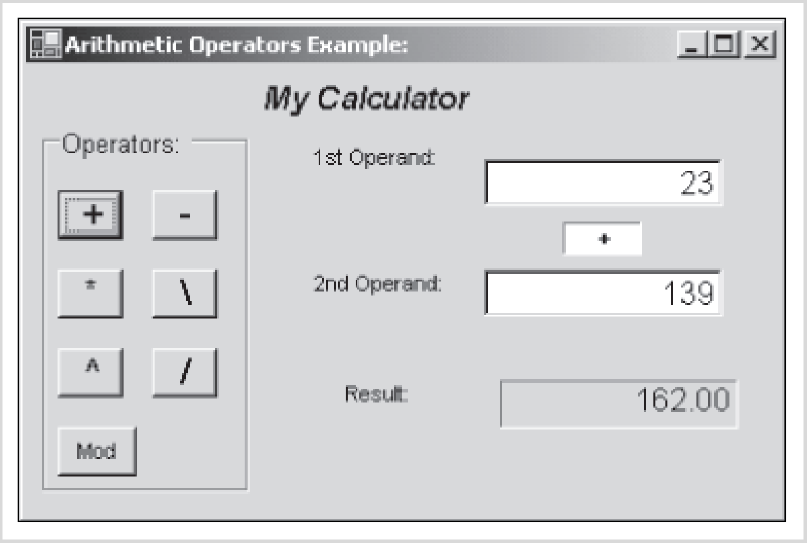


Figure 3.5 ***Calculator example***

Step 3: Determine which objects on the form should respond to events: All the Buttons should respond to their respective Click events. Therefore there should be a click event procedure for each Button placed on the Form.

Step 4: Design the algorithm for each event procedure: You must think about the way the finished project should respond to different events at the execution time. Let us develop the algorithm or the step-by-step logic for the click event procedure of btnAdd (+).

©2010 Kendall Hunt Publishing

Pseudocode of the Click event of btnAdd:

```
1. Declare variables to store the user's input.
2. Declare a variable to store the result of the operation.
3. Store the entered data in the declared variables.
4. Display the character "+" in the Label between the two TextBoxes.
5. Add the two variables and store the result in the output variable.
6. Display the result in the output Label.
```

The pseudocode for the Click event of other Buttons are very similar to the one provided for btnAdd.

Step 5: TRANSLATE THE LOGIC INTO VB .NET: This step of the PDLC entails translating the pseudocode to VB .NET code. The code for each operation should be written in the click event procedure of the corresponding Button. Below is the code for the click event procedure of btnAdd.

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
           ByVal_ e As System.EventArgs) Handles btnAdd.Click
   Dim N1 As Double
   Dim N2 As Double
   Dim Sum As Double
   N1 = Double.Parse(txtNum1.Text)
   N2 = Double.Parse(txtNum2.Text)
   lblOperator.Text = "+"
   Sum = N1 + N2
   lblOutput.Text = Sum.ToString("n")
End Sub
```

Step 6: DEBUG AND TEST THE PROGRAM: The last step of the PDLC entails debugging the program and testing it by entering different input values. One should always test all the code in the program by entering varying sample inputs, and clicking on each Button. Verify the program's output with the expected results for the sample input data. If the program generates the right answer for all the sample data, one may assume that the program is working properly.

3.10 Different Types of Errors

It is quite rare for a computer program to run perfectly the first time. There are four kinds of errors that a computer programmer may encounter: compile error, run-time error, logic error, and a data entry error.

- Compile error, also known as build error or syntax error, takes place at compilation time, when the rules of the language are violated. The compile error is amongst the easiest errors to fix. In VB .NET, such errors are underlined in the code window. By placing the mouse cursor on the word that is underlined, VB .NET provides descriptive information about the nature of that error. This feature of VB .NET is referred to as IntelliSense technology, which guides the programmer when typing the VB .NET statements.

©2010 Kendall Hunt Publishing

- Runtime error takes place during the program execution when the computer is unable to execute a statement in the program. Examples of such errors are division by zero, converting a non-numeric string to a number, or using a nonexistent function in the program.
- Logic errors are a result of using the wrong formulas in calculations. Such errors may be difficult to find and fix because VB .NET will not generate any errors. Having a good set of test data will be helpful in catching these errors.
- Data entry errors are the result of erroneous data entry by the user of the program, such as entering a wrong ID number for a student. Such errors are also hard to detect. The program should have tight data validations to reduce the chance of these errors.

3.11 Module Scope Identifiers

In chapter 2, we mentioned that, besides name, data type, and value, a variable also has a scope and a lifetime. Scope of a variable is the segment of code in which the variable can be used. A variable's scope depends on where in the program the variable is declared. A variable that is declared inside a procedure has local scope and is only accessible within that procedure; variables as such are called local variables.

A variable with module scope must be declared outside all the procedures (and functions) at the beginning of the code window, after the first line of the code generated by VB .NET; which is usually: Public Class Form1. The scope of a module scope variable is the entire Code Window. In other words, a module scope variable is accessible to all the procedures and functions in the Code Window of the Form. It is recommended that such variables be declared with the keyword Private, and their name be preceded by a lower case *m.* One may also declare a named constant with module scope, in that case the name should be preceded by mc. Declaration syntax:

```
Private mTotal As Integer
```

Lifetime of Module Scope Identifiers

Another important asset of the identifiers with module scope is that their lifetime is the duration of the program execution (only if the program is made up of a single Form). In other words, these variables come to existence when the program execution begins, and get destroyed when the program execution ends (only if the program is made up of a single Form). Hence the value stored in these variables remains in the memory throughout the program execution.

When to Use Module Scope Variables

In general you must avoid giving an identifier a scope larger than is needed. The reason is that when a variable is accessible to all the procedures and functions in the program, it is difficult to track down where it was changed in case of unexpected results. Nevertheless, there are typically two situations that qualify the declaration of a variable with module scope:

1. **Reduce redundancy:** If the variable is being used in several procedures, one may declare it with module scope. Example 3 in this chapter is a good candidate for this case. By declaring three variables for inputs, and output with module scope, one can remove their declaration from all the click event procedures, and avoid redundant statements.

©2010 Kendall Hunt Publishing

2. **Counting & Accumulating:** If the variable is being used to count the number of times a Button has been clicked or to add up the values entered in the program, it has to be declared with module scope, so that the value stored in it will stay in the memory throughout the program execution due to the lifetime of the variable. To better understand this concept, let us look at another example.

EXAMPLE 4

Develop a VB .NET project to process the quiz scores of students in a class. The program should allow the user to enter one score at a time. The user should be able to view the number of entered scores, and the computed class average. Assume that the quiz score is a whole number.

Let us go through the PDLC (Program Development Life Cycle)

Step 1: Analyze the problem:

- Input needs: quiz score
- Output needs: Number of scores and class average
- Processing needs:
  a. Enter the score
  b. Display the count and the average

Step 2: Design the GUI: Based on step-1, there should be one TextBox for entering the score, one Label to display the program's output (one might use two Labels as well). The processing needs require two Buttons, one for entering the score in the program, and the other for displaying the results. Let us name these objects as: txtScore, lblOutput, btnEnter, and btnDisplay. Change the Text property of each object to display its objective on the Form. Figure 3.6 shows the designed GUI for this example:

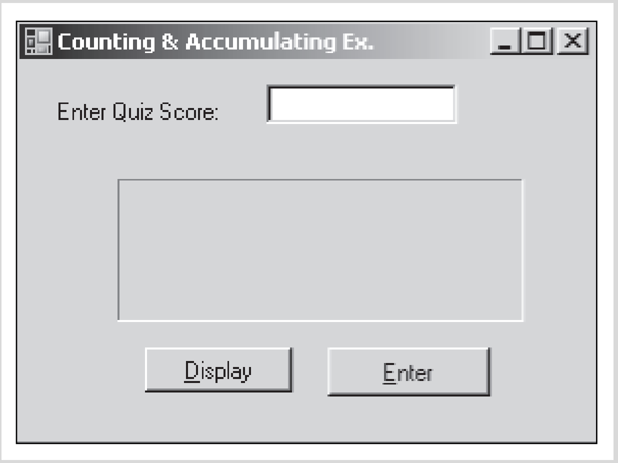
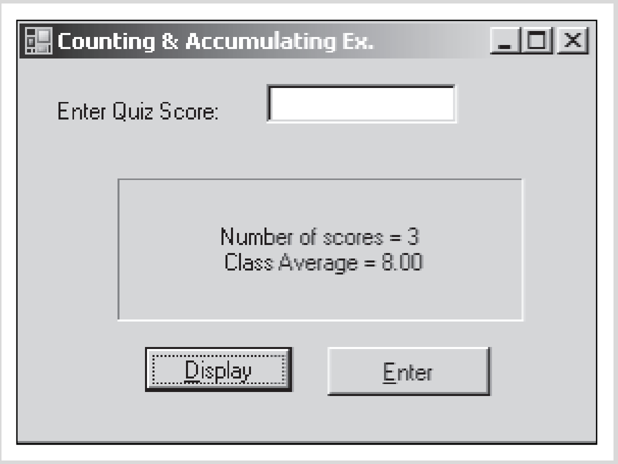


Figure 3.6 ***Module Scope Example***

Step 3: Determine which objects on the form should respond to events: The two Buttons should respond to the click event; hence we should develop the logic for those event procedures.

Step 4: Design an algorithm (logical solution) for each event procedure. One has to think about the task that each Button is responsible for. The program should use a counter that gets incremented by one each time a new score is entered by the user. There should also be another variable that adds up the entered scores. The Enter Button, should increment the number of quiz scores entered, and add the score to the total. The Display Button should compute the average by dividing the total scores by the number of scores



```
Number of scores = 3
Class Average = 8.00
```

Figure 3.7 ***Screen capture at runtime***

©2010 Kendall Hunt Publishing

and display the output. These variables are being updated in the Enter Button, and being used in the Display Button. Therefore, they must be available to both event procedures; hence be declared with module scope. Another reason for declaring these variables with module scope is that if they were declared with local scope inside the Enter Button, the value stored in them would not stay in the memory from one click of the Enter Button to the next.

Pseudocode of btnEnter:

```
1. Declare a variable to store the input data.
2. Increment the number of scores entered.
3. Add the entered score to the total scores.
4. Clear the TextBox.
5. Set the focus to txtScore, for next data entry.
```

Pseudocode of btnDisplay:

```
1. Declare a variable to store the average.
2. Compute the average by dividing the total scores by the number of scores.
3. Display the number of scores in the output Label.
4. Display the computered average on the next line of the output Label.
```

Step 5: Translate the algorithm designed in step 4 to VB .NET: One has to declare two variables with module scope, one to count the number of scores entered, and another one to add up the scores.

The Code Window, with pseudocode translated to VB .NET code:

```
Public Class Form1
    Private mCount As Integer
    Private mSum As Integer
    Private Sub btnEnter_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnEnter.Click
        Dim Score As Integer
        Score = Integer.Parse(txtScore.Text)
        mCount = mCount + 1
        mSum = mSum + Score
        txtScore.Clear()
        txtScore.Focus()
    End Sub
    Private Sub btnDisplay_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnDisplay.Click
        Dim Average As Single
```

©2010 Kendall Hunt Publishing

```
        Average = mSum / mCount
        lblShow.Text = “Number of scores =” & mCount.ToString & vbLf
        lblShow.Text = lblShow.Text & “Class Average =” & _
            Average.ToString(“n”)

    End Sub

End Class
```

Step 6: DEBUG AND TEST THE PROGRAM: Let us enter three quiz scores, 7, 8, and 9 and click on Display button to see the output. The output should be three quizzes entered, and average 8.00. Figure 3.7 is the screen capture of this run of the program.

Abbreviation Notice: Since the first line of the Click event procedures have similar list of parameters (inside the parentheses), for brevity, we will replace the parameter list with ellipsis (. . .) in the majority of the text from this point on. In other words the first line of the Click event procedure for a Button, e.g., btnExit will be written as follows:

```
Private Sub btnExit_Click (. . .) Handles btnExit.Click
```

Please notice that you should *not* use the same shorthand when typing your code in a VB project, or it will result in a build error.

Review Questions:

1. Evaluate the following arithmetic expressions.
   a. 23 * 2 / 5
   b. 3 + 5 * 2
   c. 2 * (12 + 5) Mod 9
   d. 1200 \ 2 ^ 3
   e. –10 ^ 2
2. Translate the following formula to VB .NET code. Assume variables X, Y, and Z are declared as Double, and (.) is used to show multiplication in the formula:

   $$z = \frac{X^2 + XY}{3Y}$$
3. Declare a variable to store the name of a student. In another statement, store the value entered in TextBox, txtName in this variable.
4. Declare a variable to store the number of students in a class. In another statement, store the number entered in the TextBox, txtNumber in this variable.
5. Declare two variables to store the area and the radius of a circle.
6. Store the value entered in txtRadius in the variable declared for radius.
7. Write a statement to compute the area of the circle and store the result in the variable declared for area.
8. Display the area of the circle in lblOutput, with two digits after the decimal point.

©2010 Kendall Hunt Publishing

PROGRAMMING CHALLENGES

1. Design a VB project to compute and display some statistics about a road trip the user has taken during a long weekend. The user should enter the following information:

   - Distance in miles,
   - The gallons of gas used,
   - Price per gallon,
   - The number of minutes it took to drive to destination.

   When the user clicks on "Compute" Button, the program should compute and display the following information in one large Label, on three separate lines:

   - The mileage (miles per gallon),
   - The average speed on this trip (miles per hour),
   - Cost of gas.

   The program should have Buttons to clear the input and output boxes and to end the program execution.

2. Design a VB project for a local car rental agency that calculates rental charges. The agency charges $20 per day and $0.15 per mile. The user should enter the customer's name, beginning odometer reading, ending odometer reading, and number of days the car was rented. The program should compute the miles the car was driven, and the due amount when the car is returned. The program output should be displayed in one Label, starting with a descriptive message, e.g., "Charger for customer's name:" followed by the miles driven, and the amount in due currency format on separate lines. The program should have a way to clear the I/O boxes and end the program execution. Go through the PDLC to design a suitable GUI for this project. You might consider placing the input boxes in a GroupBox.

3. Design a VB project to study the gas prices on the pump for the past four months in the local market. In this project the user should enter the regular gas prices of the first day of each month in the provided TextBoxes. There should be Buttons to perform the following:

   - Clear the I/O boxes.
   - End the program execution.
   - Display the entered regular gas prices in two columns in the output Label, i.e.,

MONTH	PRICE/GALLON
01	$1.98
02	$2.25
03	$2.95
04	$2.09

   - Compute and display the average price of the regular gas in 4 months.

©2010 Kendall Hunt Publishing

- Display the regular, super, and premium gas prices at the beginning of each month. Assume that the super gas price is 5 % more than the regular and price of the premium gas is 12 % more than the regular gas.
- Compute the financial index required for projecting the future gas prices. Assume that the financial index is computed using the following formula, where P1 . . .P4 are the regular gas prices at the beginning of the first . . . fourth month.

$$\text{Index} = \frac{P1^2 + P2^2 + P3^2 + P4^2}{P1 * P2 * P3 * P4}$$