Visual Programming IS202

Chapter Three

Variables, Constants, and Calculations

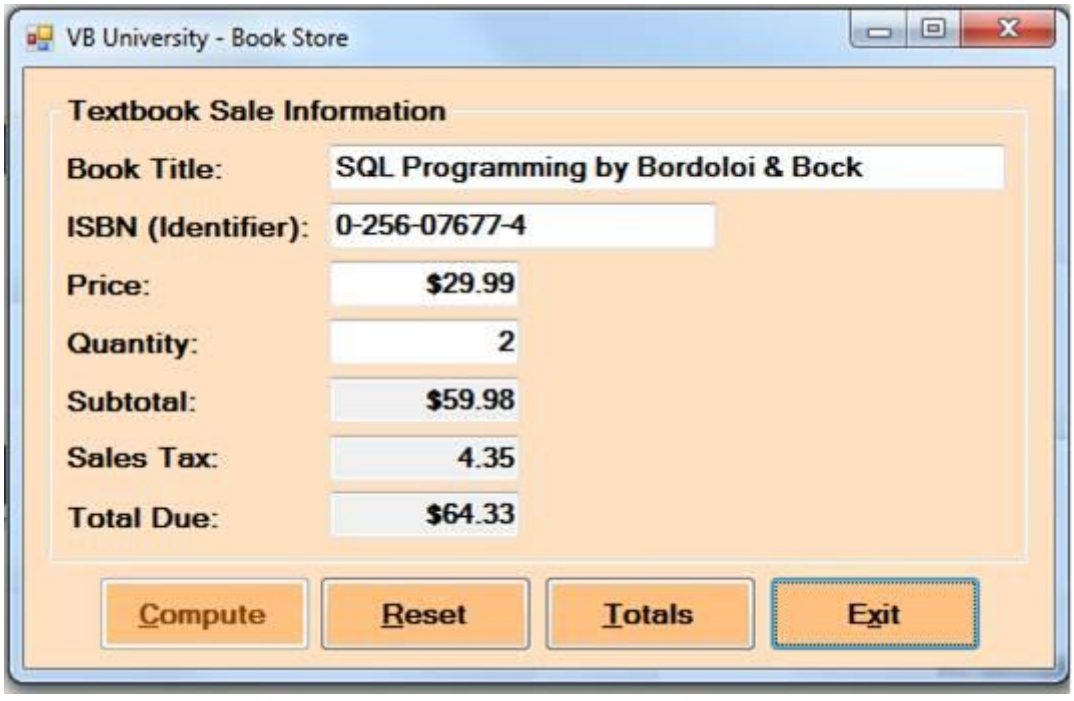
Coverage:

This chapter teaches you to write programs that involve the calculation of values for display to a form. In order to calculate values, you will often need to compute intermediate values and this will involve the storage of values to memory locations called variables. You will learn how to declare variables to store different types of data, and to convert text data to numeric data, to format output, and use Try-Catch blocks to catch data processing exceptions (also termed errors). You will also use message boxes to display messages to the application user.

INTRODUCTION

Chapter 3

In this chapter you build a project that computes information about textbook sales for the VB University. The project form is shown below and uses some of the controls that you learned in your earlier study of VB.



This project is programmed by using the **Input-Process-Output** model for programming.

Input – Application users enter values into TextBoxes (and other controls such as check boxes if such controls are used on a form). These values are input to the program. The program must convert the values entered into the textboxes and store the values to memory variables or locations in memory.

Process – The computer program includes code that processes the input values stored in memory and produces output values, also stored in memory.

Output – The output memory values computed are displayed to read-only textbox controls on the form. Either textbox controls (with the **ReadOnly** property set to **True**) or label controls are used to display output information – the key is you do not want the application user to be able to enter values into a control used to display output – I prefer to use read-only textbox controls instead of labels.

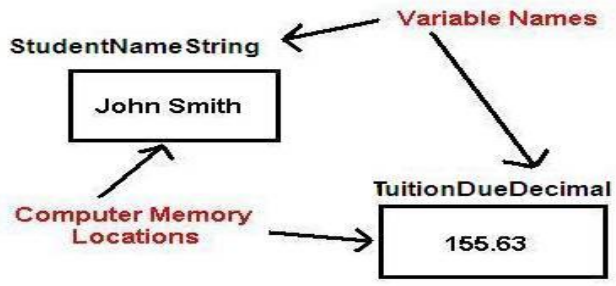
Variables, Constants, and Data Storage

Variables

- Provide a means to store data values that are not stored in files.
- Support the computation of values through their use in formulas in assignment statements.
- Represent locations in a computer's memory.
- are assigned a unique name for reference when writing computer code

This figure illustrates the concept of a **variable name** associated with locations in random access memory.

- Values are stored to memory locations (represented by the rectangle).
- The stored values can vary over time.
- Each memory location is assigned a unique variable name that is used to reference the location when you write a program.
- Later you will learn how to name variables and allocate memory locations to the storage of data.



Types of Data

VB provides numerous data types to allow programmers to optimize the use of computer memory. Each data type is described in the table shown here. Note the:

- Data type name.
- Description of data stored.
- Memory storage requirement in bytes.

Table 3.1		
Data Type	**Description of data stored**	**Memory storage in bytes**
Text Data Storage		
String	Alphanumeric data such as letters of the alphabet, digits that are not treated as numbers, and other special characters.	Size varies
Char	Stores single Unicode characters (supports any international language).	2
Numeric Data Storage – Fixed Point		
Decimal	Decimal numeric values – often used to store dollars/cents.	16
Numeric Data Storage – Floating Point		
Double	Double-precision numeric values with 14 digits of accuracy.	8
Single	Single-precision numeric values with 6 digits of accuracy.	4
Numeric Data Storage – Whole Numbers (no decimal point)		
Short	Whole numeric values in the range -32,768 to 32,767.	2
Integer	Whole numeric values in the range -2,147,483,648 to +2,147,483,647.	4
Long	Whole numeric values that are very, very large.	8
Special Data Types		
Boolean	True or False.	2
Byte	Stores binary data of values 0 to 255 – can be used to store ASCII character code values.	1
Date	Stores dates in the form 1/1/0001 to 12/31/9999.	8
Object	Stores data of any type.	4

Most business applications primarily use String, Decimal, Single, and Integer data types.

Here are examples of data that may be stored to computer memory:

- **Customer Name** – **String** – Stores alphabetic characters.
- **Social Security Number** – **String** – Numbers that are not used in computations.
- **Customer Number** – **String** – Numbers that are not used in computations.
- **Balance Due** – **Decimal** – Used in calculations and often requires a decimal point to separate dollars and cents.
- **Quantity in Inventory** – **Integer** or **Short** – Selection depends on how large the quantity in inventory will be, but the quantity is usually a whole number.
- **Sales Tax Rate** – **Single** or **Decimal** – Used to store a percentage value; also used for scientific calculations.

Naming Rules for Variables and Constants

You as the programmer decide what to name variables and constants – there are technical rules you must follow when creating variable and constant names.

- Names can include letters, digits, and the underscore, but **must** begin with a letter.
- Names cannot contain spaces or periods.
- Names cannot be **VB reserved words** such as **LET**, **PRINT**, **DIM**, or **CONST**.
- Names are **not** case sensitive. This means that **TotalInteger**, **TOTALINTEGER**, and **totalinteger** are all **equivalent** names.
- For all intensive purposes, a Name can be as long as you want (the actual limit is 16,383 characters in length).

Naming Conventions

Naming variables and constants follows the **Camel Casing** naming convention that you learned in your earlier studies. Use the following guidelines:

1. Create meaningful names – do not name a variable or constant **X** or **Y** or **XInteger**. Instead use names such as **StudentNameString**, **CountStudentsInteger**, and **AmountDueDecimal**.

2. Avoid using abbreviations unless the abbreviation is standard and well-accepted such as **SSNString** for storing social security number values.

3. Begin each name with an uppercase letter and capitalize each successive word in the name.

4. Use mixed case (such as **AmountDueDecimal**) for variables – use Uppercase for constant names (such as **TAX_RATE_SINGLE**).

This table provides sample identifier names:

Table 3.2

Data Type	Example Variable or Constant Name
Boolean	CheckedBoolean
Date	ShipDate
Decimal	AmountDueDecimal
Double	DistanceToMoonDouble
Integer	CountInteger
Long	WorldPopulationLong
Single	TAX_RATE_SINGLE
Short	ClassSizeShort
String	StudentNameString

Declaring Variables

Declare local variables with the **Dim** statement and module-level variables with the **Private** statement.

- **Dim** statement – use this to declare variables and constants inside a procedure these are local variables.
- The **Dim** statement needs to specify a variable/constant name and data type.
  - Specifying an initial value for a variable is optional -- do so if necessary.
  - If an initial value is not assigned, then a string stores the "empty string" and all numeric variable types store the value zero.
  - Examples:

```
Dim StudentNameString As String

Dim CountStudentsInteger As Integer

Dim AccountBalanceDecimal As Decimal = 100D
```

You can also declare more than one variable in a **Dim** statement.

```
Dim StudentNameString, MajorString As String

Dim SubtotalDecimal, TotalDecimal, TaxAmountDueDecimal As Decimal
```

This is an example of declaring two different module-level variables.

```
Private CountStudentsInteger As Integer

Private TotalDecimal As Decimal
```

Declaring Constants

- Declare constants with the **Const** statement.
- Constants are similar to variables – constants are values that are stored to memory locations; however, a constant cannot have its value change during program execution – constant values are generally fixed over time.
- Examples: sales tax rate or name of one of the states in the United States.

VB has two different types of constants.

1. **Intrinsic Constants** – these are defined as **enumerations** such as **Color.Red** and **Color.Blue**. These are called **intrinsic** because they are predefined in VB and always exist for your use.
2. **Named Constants** – these are constants you define with a **Const** statement. These constants are specific to your programming application.

Examples:

```
Const SALES_TAX_RATE_SINGLE As Single = 0.0725F

Const BIG_STATE_NAME_STRING As String = "Alaska"

Const TITLE_STRING As String = "Data Entry Error"
Const MAX_SIZE_INTEGER As Integer = 4000
```

String (text or character) constants are assigned values within the " " (double quote) marks. This statement stores double-quote marks as part of the string – do this by typing two double-quote marks together.

```
Const COURSE_TITLE_STRING As String = ""Programming Visual Basic""
```

Numeric constants like those shown above do NOT use double-quote marks – just type the numeric value numbers. Follow these rules for assigning numeric values to constants:

- You can use numbers, a decimal point, and a plus (+) or minus (-) sign.
- Do not include special characters such as a comma, dollar sign, or other special characters.
- Append one of these characters to the end of the **numeric constant or variable** to denote a data type declaration. If you do not use these, a whole number is assumed to be **Integer** and a fractional value is assumed to be **Double**.

```
Decimal     D   40.45D

Double      R   12576.877R

Integer     I   47852I

Long        L   9888444222L

Short       S   2588S

Single      F   0.0725F
```

Scope of Variables and Constants

Each variable (or constant) has a finite lifetime and visibility – termed the **Scope**. The **variable lifetime** is how long the variable exists before the computer operating system garbage-collects the memory allocated to the stored value.

There are four levels of scope.

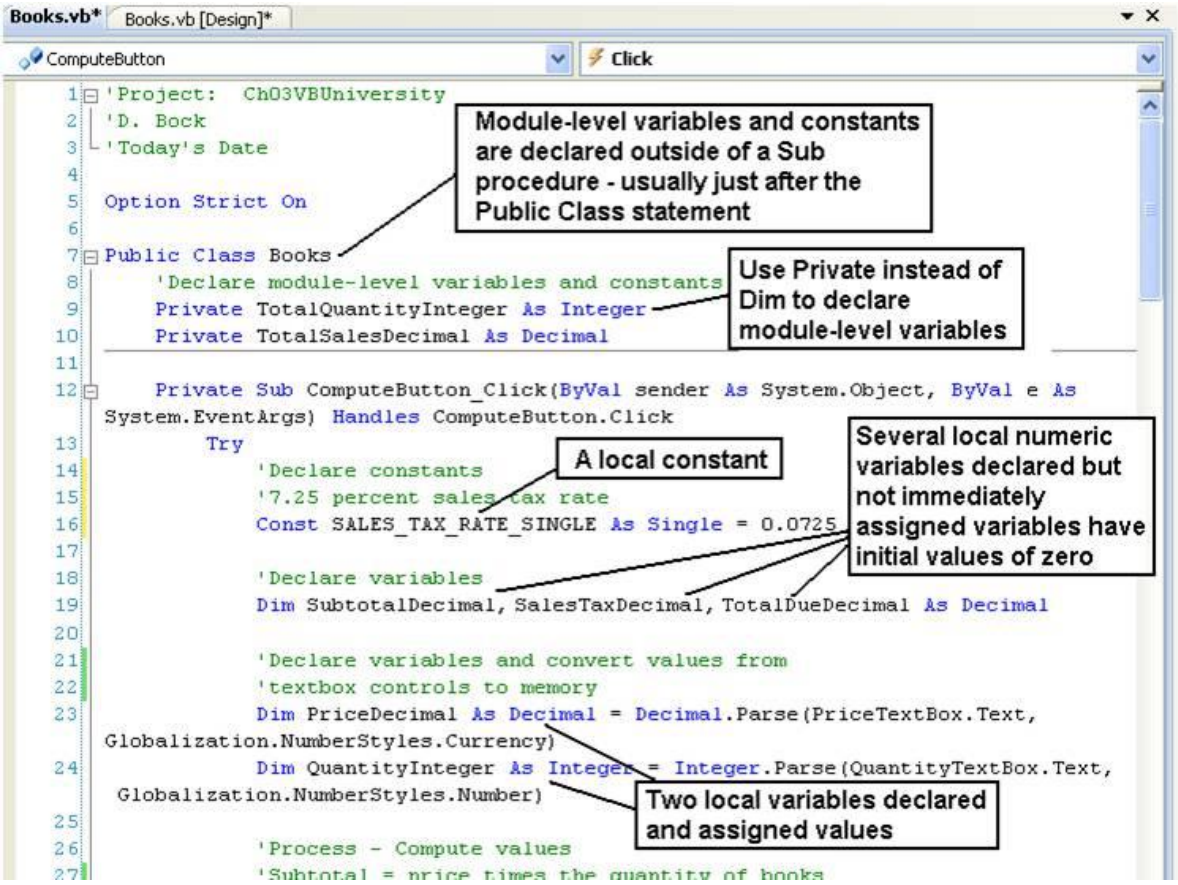
- **Namespace** (use a **Public Shared** declaration instead of **Dim**) – the variable is visible within the entire project (applicable to a project with multiple forms).
    - The variable is project-wide and can be used in any procedure in any form in the project.
    - The variable memory allocation is garbage-collected when application execution terminates.
- **Module level** (usually use **Private** to declare a variable; use **Const** to declare a constant) – a variable/constant can be used in any procedure on a specific form – it is not visible to other Forms.

- Use module-level variables when the values that are stored in their memory locations are used in more than one procedure (click event or other type of procedure).
- A module-level variable or constant is created (allocated memory) when a form loads into memory and the variable or constant remains in memory until the form is unloaded.

- **Local** (use **Dim** to declare a variable; use **Const** to declare a constant) – a variable/constant is declared and used only within a single procedure.
  - Variable **lifetime** is the period for which a variable exists.
  - When a procedure executes, such as when you click on a button control, each variable and constant declared as local within the procedure executes, "uncreated" when the procedure executes the **End Sub** statement.
  - Each time you click the button, a new set of variables and constants are created.

- **Block** (use **Dim** to declare the variable; use **Const** to declare the constant) – the variable/constant is only visible within a small portion of a procedure – Block variables/constants are rarely created.

This figure illustrates where to declare **local** versus **module-level** variables/constants. In a later chapter you will learn to declare namespace and block variables and constants and when to use the **Public** keyword in place of **Private**.



Key Points about Errors

- If you name a module-level and local variable the **same name**, VB will create two different variables! The local variable will exist within the procedure where it is named, but the module-level variable will exist elsewhere in the code for the form. This also applies to constants. **AVOID THIS ERROR**.
- When you first declare a local variable, VB will underline the variable and tell you it is "an unused local variable" – this is not really an error because the exception message will go away automatically when you use the variable name in an assignment statement.

CALCULATIONS

Calculations are performed with variables, constants, and object properties such as the **Text** property of a TextBox.

Converting Input Data Types

As part of the **Input** phase of the **Input-Process-Output** model, you must convert values from the **Text** property of a TextBox and store the converted values to memory variables.

- **Text** property – always stores **string** values, even if the string looks like a number.
- **Parse** method – converts a value from a Text property to an equivalent numeric value for storage to a numeric variable. Parse means to examine a string character by character and convert the value to another format such as decimal or integer.
- In order to parse a string that contains special characters such as a decimal point, comma, or currency symbol, use the **Globalization** enumeration shown in the coding examples below.
    - If you don't specify the Globalization value of **Globalization.NumberStyles.Currency**, then a value entered into a textbox control such as **$1,515.95** will **NOT** parse to Decimal.
    - The Globalization value **Globalization.NumberStyles.Number** will allow the **Integer.Parse** method to parse a textbox value that contains a comma, such as **1,249**.
    - VB's Intellisense will display the various possible values for the Globalization enumeration.

Example #1 – this example shows you how to declare numeric variables then store values to them from Textbox controls.

```
'Declare variables

    Dim PriceDecimal As Decimal

    Dim QuantityInteger As Integer


    'Convert values from textbox controls to memory

    PriceDecimal
    = Decimal.Parse(PriceTextBox.Text, Globalization.NumberStyles.Curre
    ncy)

    QuantityInteger
    = Integer.Parse(QuantityTextBox.Text, Globalization.NumberStyles.Nu
    mber)
```

Example #2 – this example shows you how to declare numeric variables and store values to them from Textbox controls using a single assignment statement in one step.

```
'Declare variables and convert values from textbox
'controls to memory in a single statement
Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text,
    Globalization.NumberStyles.Currency)
Dim QuantityInteger As
Integer = Integer.Parse(QuantityTextBox.Text,
    Globalization.NumberStyles.Number)
```

Older versions of VB used **named functions** to convert values. Examples are the **CDec** (convert to Decimal) and **CInt** (convert to Integer) functions shown here – you may encounter these functions in other VB books that you read or in the VB Help files.

There are some advantages to these named functions:

- A TextBox **Text** property value of **$100.00** will **NOT** generate an error if you use the **CDec** function to convert the value as shown below—the data will convert satisfactorily.
- The functions are faster and easier to type.

```
'Converts to decimal and Integer
PriceDecimal = CDec(PriceTextBox.Text)
QuantityInteger = CInt(QuantityTextBox.Text)
```

Converting Variable Values to Output Data Types

In order to display numeric variable values as output the values must be converted from numeric data types to string in order to store the data to the **Text** property of a TextBox control. Use the **ToString** method. These examples show converting strings to a numeric representation with 2 digits to the right of the decimal (**N2**) and currency with 2 digits to the right of the decimal (**C2**) as well as no digits to the right of a number (**N0 – that is N zero, not N Oh**).

```
SubtotalTextBox.Text = SubtotalDecimal.ToString("N2")
SalesTaxTextBox.Text = SalesTaxDecimal.ToString("C2")
QuantityTextBox.Text = QuantityInteger.ToString("N0")
```

Implicit Conversion – this is conversion by VB from a narrower to wider data type (less memory to more memory) – this is done automatically as there is no danger of losing any precision. In this example, an integer (4 bytes) is converted to a double (8 bytes):

```
BiggerNumberDouble = SmallerNumberInteger
```

Explicit Conversion – this is also called **Casting** and is used to convert between numeric data types that do not support implicit conversion. This table shows use of the **Convert** method to convert one numeric data type to another numeric data type. Note that fractional values are rounded when converting to integer.

Table 3.3	
Decimal	**NumberDecimal = Convert.ToDecimal(ValueSingle)**
Single	**NumberSingle = Convert.ToSingle(ValueDecimal)**
Double	**NumberDouble = Convert.ToDouble(ValueDecimal)**
Short	**NumberShort = Convert.ToInt16(ValueSingle)**
Integer	**NumberInteger = Convert.ToInt32(ValueSingle)**
Long	**NumberLong = Convert.ToInt64(ValueDouble)**

Performing Calculations – VB uses a wider data type when calculations include unlike data types. This example produces a decimal result.

```
AverageSaleDecimal = TotalSalesDecimal / CountInteger
```

Summary Rules:

- Use the **Parse** method to convert a string to a number or to parse the value in a textbox control.
- Use the **Convert** method to convert a type of number to a different type of number.

Arithmetic Operators

The **arithmetic operators** are the same as in many other programming languages. They are:

+ Addition

- Subtraction

* Multiplication

/ Division

^ Exponentiation

\ Integer Division

Mod Modulus Division

Exponentiation – This raises a number to the specified power – the result produced is data type **Double**. Example:

```
ValueSquaredDouble = NumberDecimal ^ 2
```

```
ValueCubedDouble = NumberDecimal ^ 3
```

Integer Division – Divide one integer by another leaving an integer result and discarding the remainder, if any. Example:

- If the variable **MinutesInteger = 130**, then this expression returns the value of **2 hours**.

```
HoursInteger = MinutesInteger \ 60
```

Modulus Division – This returns the remainder of a division operation. Using the same value for **MinutesInteger = 500**, this expression returns the value **20 minutes** and can be used to calculate the amount of overtime worked for an 8-hour work day.

```
MinutesInteger = MinutesInteger Mod 60
```

Order of Precedence

The **order of precedence** for expressions that have more than one operation is the same as for other programming languages.

Evaluate values and calculation symbols in this order:

(1) Values enclosed inside parentheses

(2) Exponentiation

(3) Multiplication and Division

(4) Integer Division

(5) Modulus Division

(6) Addition and Subtraction

The order of precedence is applied to an expression by evaluating the expression from left to right for values within parentheses – within parentheses VB will process the expression from left to right looking for an applying the exponentiation operator, then again from left to right applying the multiplication and division operators, etc. This left to right application of operators continues in pass-after-pass working down the order of precedence.

Use parentheses to control the application of the order of precedence of operations.

- Example #1: **(5 + 6) * 2** is evaluated:
  - first as **5 + 6 = 11**, because the parentheses force the addition operation to be evaluated before the multiplication operation,
  - next VB will multiple **11** by **2** to arrive at **22**.
- Example #2: **5 + 6 * 2** is evaluated:
  - first as **6 * 2 = 12**, because the multiplication operator is higher in the order of precedence,
  - next VB will add **5** to **12** to arrive at **17**.

Work the problems in the following table and record the results. Assume that: **X=2**, **Y=4**, and **Z=3**.

Table 3.4

Problem	Result
X + Y ^ Z	66
16 / Y / X	2
X * (X + 1)	6
X * X + 1	5
Y ^ X + Z * 2	22
Y ^ (X + Z) * 2	2048
(Y ^ X) + Z * 2	22
((Y ^ X) + Z) * 2	38

This table shows example mathematical notation and the equivalent VB expression.

Table 3.5

Mathematical Notation	VB Expression
2X	2 * X
3(X + Y)	3 * (X + Y)
(X + Y)(X – Y)	(X + Y) * (X – Y)
πr^2	3.14 * r ^ 2

Assignment Operators and Formulas

The **equal sign** is the assignment operator. It means store the value of the expression on the right side of the equal sign to the memory variable named on the left side of the equal sign. Examples:

```
ItemValueDecimal = QuantityInteger * PriceDecimal

HoursWorkedSingle = MinutesWorkedSingle / 60F
NetProfitDecimal = GrossSalesDecimal – CostGoodsSoldDecimal
```

The plus symbol combined with the equal sign allows you to **accumulate** a value in a memory variable. Examples:

```
TotalSalesDecimal += SaleAmountDecimal
```

is equivalent to the following – it means take the current value of **TotalSalesDecimal** and add to it the value of **SaleAmountDecimal** and store it back to **TotalSalesDecimal** (it gets bigger and BIGGER).

```
TotalSalesDecimal = TotalSalesDecimal + SaleAmountDecimal
```

The minus symbol combined with the equal sign allows you to decrement or count backwards. Examples:

```
CountInteger -= 1
```

is equivalent to

```
CountInteger = CountInteger - 1
```

Option Explicit and Option Strict

These options change the behavior of your coding editor and the program compiler.

- **Option Explicit** option is **ON** by default in VB.NET.
    - This option requires you to declare all variables and constants.
    - If set to Off, you do not need to declare any variables.
    - Sometimes programmers will turn this option off with the command shown here, but it is a **bad practice** because it can cause you to spend many hours trying to find errors in variable names that **Option Explicit On** will find for you.

```
Option Explicit Off
```

- **Option Strict** option is **OFF** by default in VB.
    - This option causes the editor and compiler to try to help you from making mistakes by requiring you to convert from wider data types to narrower ones (ones using less memory).
    - Helps avoid the mistake of mixing data types within an expression, for example: trying to add a string value to an integer value.
    - With **Option Strict Off**, you can write the following assignment statement to store a value from a textbox to a memory variable – VB will automatically convert the string data in the textbox to integer data for storage in the variable:

```
QuantityInteger = QuantityTextBox.Text
```

- With **Option Strict On**, you must write the following – VB will not automatically convert the data from string to integer – you must parse the data.:

```
QuantityInteger = Integer.Parse(QuantityTextBox.Text)
```

Use of **Option Strict On** is a good practice, but is not always followed in industry.

- We will almost always use Option Strict On in our programs.

- Place the command in your program after the general comments at the top of the program as the first line of code as shown here.

```
'Project:  Ch03VBUniversity
'D. Bock
'Today's Date
 Option Strict On
 Public Class Books
```

Rounding Numbers

Use the **Decimal.Round** method to round decimal values to a desired number of positions to the right of the decimal. Always specify the number of digits to round – the default is to round to the nearest whole number. Always round when multiplying and dividing or when using exponentiation as these operations can result in rounding errors. Simple subtraction and addition do not require rounding. Examples:

```
SalesTaxDecimal = Decimal.Round(SALES_TAX_RATE_DECIMAL * AmountSoldDecimal, 2)
```

```
SalesTaxDecimal = Decimal.Round(Convert.ToDecimal(SubtotalDecimal *
SALES_TAX_RATE_SINGLE), 2)
```

Formatting Data for Output

Data to be formatted for output will often use the **ToString** method you learned earlier. Additional examples are shown here for your reference in completing programming assignments:

Example #1: This shows formatting a decimal value to string for display in a textbox control – the output is formatted as currency (dollar sign, commas, 2 decimal points – the default is to format numeric output with 2 digits to the right of the decimal point).

```
SalesTaxTextBox.Text = SalesTaxDecimal.ToString("C")
```

Example #2: This shows formatting as currency, but with no digits to the right of the decimal point.

```
TotalDueTextBox.Text = TotalDueDecimal.ToString("C0")
```

Example #3: This formats the output as a number with two digits to the right of the decimal and with one or more commas as appropriate – sometimes you will not want to display a currency symbol.

```
TotalDueTextBox.Text = TotalDueDecimal.ToString("N0")
```

Formatting codes are:

- **C** or **c** – currency.
- **F** or **f** – fixed-point, to format a string of digits, no commas, and minus sign if needed.

- **N** or **n** – formats a number with commas, 2 decimal place values, and minus sign if needed.
- **D** or **d** – formats integer data types as digits to force a specific number of digits to display.
- **P** or **p** – formats percent value rounded to 2 decimal place values.
- Add a digit such as 0 to format with that number of decimal place values, e.g., **C0** or **N0** produces no digits to the right of the decimal whereas **C4** or **N4** would produce 4 digits to the right of the decimal point.

Older versions of VB used functions to format output – these are still widely used and are provided here for your reference if you find them in other textbooks.

FormatCurrency Function – This function displays output formatted as dollars and cents. The default is a dollar sign, appropriate commas, and two digits to the right of the decimal. This formats a value stored in memory named **BalanceDueDecimal** and displays it to the TextBox control named **BalanceDueTextBox**. Remember, **TextBox** controls store string values. Example:

```
BalanceDueTextBox.Text = FormatCurrency(BalanceDueDecimal)
```

FormatNumber Function – This function displays output formatted as numbers with commas and two digits to the right of the decimal. Example:

```
AmountTextBox.Text = FormatNumber(AmountDouble)
```

```
AmountTextBox.Text = FormatNumber(AmountDouble,3)
```

FormatPercent Function – This function displays output formatted as a percent – it multiples the argument by 100, adds a percent sign, and rounds to two decimal places. Example:

```
PercentFinishedTextBox.Text = FormatPercent(FinishedSingle)
```

FormatDate Function – This function formats an expression as a date and/or time. Examples:

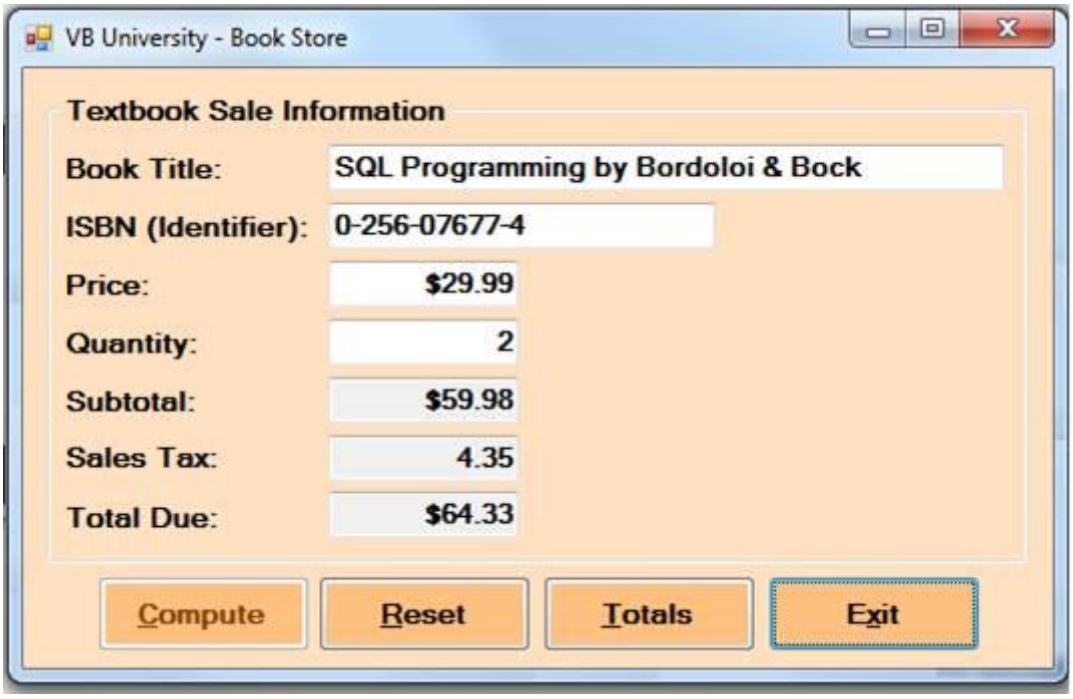
```
'Displays the value as MM/DD/YY
'Example:  2/28/07
DateTextBox.Text = FormatDateTime(StartDate, DateFormat.ShortDate)

'Displays the value as Date of week, month, day, year
'Example:  Monday, August 5, 2007
DateTextBox.Text = FormatDateTime(StartDate, DateFormat.LongDate)

'Displays the value as HH:MM (24 hour clock)
'Example:  21:15
TimeTextBox.Text = FormatDateTime(StartDateTime,
DateFormat.ShortTime)

'Displays the value as HH:MM:SS AM/PM(24 hour clock)
```

```
'Example:  6:10:24 PM
TimeTextBox.Text = FormatDateTime(StartDateTime,
DateFormat.LongTime)
```

In-Class Exercise – Computing Book Sales Information

Build the Form

Develop a project with a **Form** like that shown below.



- None of the labels used as prompts need to be named.
- The first four TextBox controls are used for data entry – use these names: **BookTitleTextBox**, **ISBNTextBox**, **PriceTextBox** and **QuantityTextBox**.
- The next three TextBox controls are used to display output. Set these properties:
- **ReadOnly** property = **True**,
- Name properties = **SubtotalTextBox**, **SalesTaxTextBox**, and **TotalDueTextBox**.
- Name the buttons **ComputeButton**, **ResetButton**, **TotalsButton**, and **ExitButton**.

Compute Button Click Event

The Click event sub procedure should compute the values to be displayed as output and display those values. Use the Input-Process-Output model.

Input: Start by entering remarks to play the logic of the procedure. Here is an example:

```
    Private Sub ComputeButton_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles ComputeButton.Click
```

```
        'Input

        'Declare constants

        '7.25 percent sales tax rate


        'Declare variables and convert values from

        'textbox controls to memory


        'Process - Compute values

        'Subtotal = price times the quantity of books


        'Sales tax = sales tax rate times the subtotal minus discount
amount


        'Total due is the subtotal minus discount amount plus sales tax


        'Output - display output formatted as currency


    End Sub
```

Do NOT try to start by coding the variables needed. Instead, each time you use a variable in an assignment statement, you then declare the variable as necessary.

- Begin by converting the **PriceTextBox** control's **Text** property value to a decimal value in memory.
- An appropriate variable name is **PriceDecimal** for the memory variable.
- This causes you to need to also declare the variable. Your code now looks like this.

```
    'Declare variables

    Dim PriceDecimal As Decimal


    'Convert values from TextBox controls to memory

    PriceDecimal
= Decimal.Parse(PriceTextBox.Text, Globalization.NumberStyles.Currency)
```

You can also combine the above two statements into a single statement if you desire.

```
    'Declare variables and convert value from textbox to memory
```

```
    Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text, Global
ization.NumberStyles.Currency)
```

Now write an assignment statement to convert the **Quantity** TextBox control's **Text** property to an integer value in memory. You will need another variable. Your code now looks like this:

```
    'Declare variables

    Dim PriceDecimal As Decimal

    Dim QuantityInteger As Integer


    'Convert values from TextBox controls to memory

    'Gives an example of using both the Convert and Parse methods.

    PriceDecimal
= Decimal.Parse(PriceTextBox.Text, Globalization.NumberStyles.Currency)

    QuantityInteger
= Integer.Parse(QuantityTextBox.Text, Globalization.NumberStyles.Number)
```

Alternatively, you can combine the above four statements into two statements.

```
    'Declare variables and convert value from textbox to memory

    Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text, Global
ization.NumberStyles.Currency)

    Dim QuantityInteger As Integer = Integer.Parse(QuantityTextBox.Text,
Globalization.NumberStyles.Number)
```

Process: You will need variables to store intermediate values that will eventually be displayed to the output TextBox controls. Examine the form. You'll see that the values are all currency so you'll use the decimal data type.

- Write the assignment statement to compute the subtotal (price multiplied by quantity).
- You might use the variable name **SubtotalDecimal** to store the subtotal value in memory.
- Also update the **Dim** statement listing that declares decimal variables.

```
    'Subtotal = price times the quantity of books

    SubtotalDecimal = PriceDecimal * QuantityInteger
```

… Go to the top of sub procedure and declare the **SubtotalDecimal** variable

```
Dim SubtotalDecimal As Decimal
```

Computing the Sales Tax: Sales tax is charged on the subtotal at the rate of **7.25%**. This requires the following actions:

- Declare a constant of data type **single** named **SALES_TAX_RATE_SINGLE** with the value **7.25%.**
- Write an assignment statement that will compute the sales tax due and assign the value to a memory variable named **SalesTaxDecimal**.
- Update the Dim statement to add **SalesTaxDecimal** to the declaration list.
- Use the **Decimal.Round** and **Convert.ToDecimal** methods to treat the expression as a decimal value and to round to the nearest penny.

```
'Sales tax = sales tax rate times the subtotal minus discount amount
SalesTaxDecimal = Decimal.Round(Convert.ToDecimal(SubtotalDecimal * SALES_TAX_RATE_SINGLE), 2)
```

… Go to the top of sub procedure and declare the **SALES_TAX_RATE_SINGLE** constant and the **SalesTaxDecimal** variable (you can add the variable to the existing Dim statement for Decimal variables).

```
Const SALES_TAX_RATE_SINGLE As Single = 0.0725    '7.25 percent rate
Dim SubtotalDecimal, SalesTaxDecimal As Decimal
```

Computing the Total Due: The total due is the formula: **subtotal + sales tax**. The total due value is stored to a memory variable named **TotalDueDecimal**. Add this variable to the **Dim** statement earlier in the sub procedure.

```
'Total due = the subtotal minus discount amount plus sales tax
TotalDueDecimal = SubtotalDecimal + SalesTaxDecimal
```

… Go to the top of sub procedure and declare the **TotalDueDecimal** variable (you can add it to the existing Dim statement).

```
Dim SubtotalDecimal, SalesTaxDecimal, TotalDueDecimal As Decimal
```

Output: Store the values from the memory variables to the **Text** property of the output TextBox controls.

- This code is straight-forward assignment statements, but requires formatting the output (if desired) to appear in currency format.
- No new variables are needed.

- The default number of digits to the right of the decimal is 2 so you do not need to specify **C2** or **N2**.

```
'Display output formatted as currency
SubtotalTextBox.Text = SubtotalDecimal.ToString("C")
SalesTaxTextBox.Text = SalesTaxDecimal.ToString("N")
TotalDueTextBox.Text = TotalDueDecimal.ToString("C")
```

Test the program.

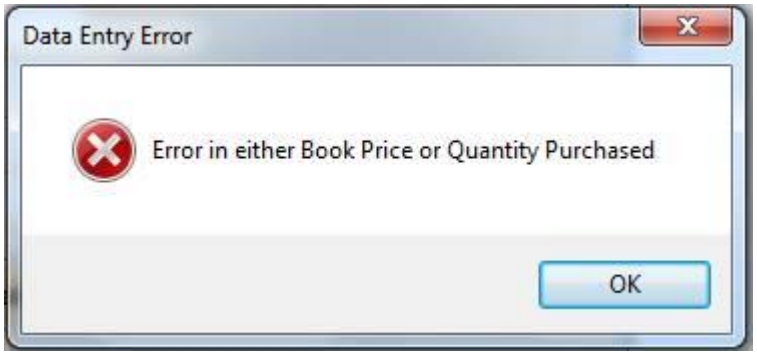
- As long as you make no data entry errors, it should produce correct output.
- If the data entered for either price or quantity is not numeric, an exception is thrown.

Handling Exceptions – Data Entry Errors

Sometimes application users will make typing errors – typing letters where they mean to type numbers.

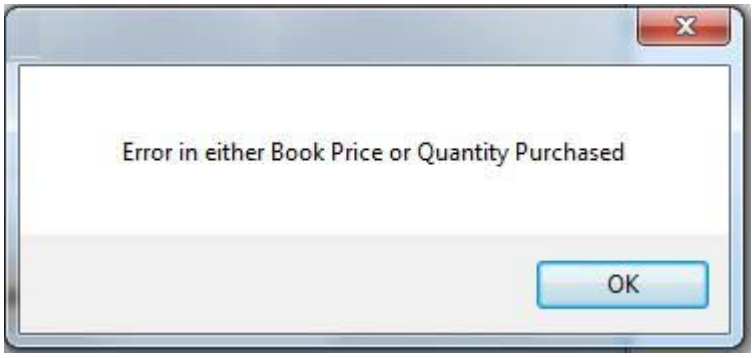
The **Parse** method returns an **exception** (also called an error) if the data entered in a TextBox cannot be converted to an appropriate numeric value, or if the TextBox is blank, or if the value contains special characters such as a percent symbol – %. In this situation, it is necessary to display an error message to the application user.

MessageBox.Show – The **MessageBox.Show** statement displays messages in the middle of the screen. You can use this to display exception messages.



The example **MessageBox.Show** statement shown below produces a "plain looking" message box like the one shown in the figure below.

```
MessageBox.Show("Error in either Book Price or Quantity Purchased")
```



You need to specify the use of specific text in the message box title bar. You can add a graphic icon and button(s) to the message box. For now use the OK button; you will learn other buttons in a later module.

MessageBox.Show Parameters – a **MessageBox.Show** method always requires a message, but has numerous optional parameters that may be included. We will use a total of four parameters—these are:

- The **message** – a string value or string variable.
- The **title bar** entry for the message box – a string value or string variable.
- The **button enumeration** – choose button(s) from the Intellisense popup.
- The **icon enumeration** – choose an icon from the Intellisense popup.

Each parameter is separated by a comma. The parameters are shown in this general format:

```
MessageBox.Show("Exception message here", "Title bar name here", button enumeration here, icon enumeration here).
```

As you type a comma between each parameter of the **MessageBox.Show** method, Intellisense will popup help to guide you in entering the title bar text value and in selecting an icon and button(s):

```
MessageBox.Show("Error in either Book Price or Quantity Purchased", "Data Entry Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
```

Try/Catch Blocks – The **Try-Catch** block is a coding technique used to catch exceptions – this is called **Exception Handling**. The general format is as follows:

```
Try
    'Place all of the code that you want to execute for the sub procedure here.
    'You can have lots of statements here.
Catch [Optional VariableName As ExceptionType]
    'Place statements for action to take when
    'an exception occurs.
    'You can also have lots of statements here.
[Finally]  'This part is optional
    'Place statements to always execute before the end
```

```
    'of the Try block.
    'You can also have lots of statements here.
End Try
```

Example:

```
Try
    QuantityInteger
= Integer.Parse(QuantityTextBox.Text, Globalization.NumberStyles.Number)
    PriceDecimal
= Decimal.Parse(PriceTextBox.Text, Globalization.NumberStyles.Currency)
    . . . other code goes here to complete the processing

Catch ex As Exception
    MessageBox.Show("Error in either Book Price or Quantity
Purchased", "Data Entry Error", MessageBoxButtons.OK,
MessageBoxIcon.Error)

End Try
```

In-Class Exercise

Modify the **ComputeButton** click event sub procedure to handle exceptions.

- Add a **Try-Catch** block to catch errors if the application user enters invalid numeric data.
- Use a **MessageBox** statement to display the appropriate message.
- Note the indentation used to aid in the readability of the code is automatically added by VB.

Go to the first line within the **ComputeButton** sub procedure. Begin by typing the word **Try** and pressing the **Enter** key. VB will add the following coding outline automatically.

```
    Private Sub ComputeButton_Click(ByVal sender As System.Object, ByVal
 e As System.EventArgs) Handles ComputeButton.Click

        Try

        Catch ex As Exception

         End Try
```

Now highlight and drag/drop (or cut/paste) all of the code within the sub procedure that you wrote earlier and paste this inside the **Try** portion of the **Try-Catch** block. Your code now looks like this:

```
    Private Sub ComputeButton_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles ComputeButton.Click

        Try

            'Declare constants
```

```
            '7.25 percent sales tax rate

            Const SALES_TAX_RATE_SINGLE As Single = 0.0725


            'Declare variables

            Dim SubtotalDecimal, SalesTaxDecimal,
TotalDueDecimal As Decimal


            'Declare variables and convert values from

            'textbox controls to memory

            Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text
, Globalization.NumberStyles.Currency)

            Dim QuantityInteger As Integer = Integer.Parse(QuantityTextBo
x.Text, Globalization.NumberStyles.Number)


            'Process - Compute values

            'Subtotal = price times the quantity of books

            SubtotalDecimal = PriceDecimal * QuantityInteger


            'Sales tax = sales tax rate times the subtotal

            SalesTaxDecimal
= Decimal.Round(Convert.ToDecimal(SubtotalDecimal *
SALES_TAX_RATE_SINGLE), 2)


            'Total due = subtotal plus sales tax

            TotalDueDecimal = SubtotalDecimal + SalesTaxDecimal


            'Display output formatted as currency

            SubtotalTextBox.Text = SubtotalDecimal.ToString("C")

            SalesTaxTextBox.Text = SalesTaxDecimal.ToString("N")

            TotalDueTextBox.Text = TotalDueDecimal.ToString("C")

        Catch ex As Exception
```

```
            MessageBox.Show("Error in either Book Price or Quantity
Purchased", "Data Entry Error", MessageBoxButtons.OK,
MessageBoxIcon.Error)

            PriceTextBox.Focus()

        End Try

    End Sub
```

Notice that all of the code that you want to execute will **ALWAYS** go inside the **Try** portion of the **Try-Catch** block.

Note the addition of code above inside the **Catch** portion of the **Try-Catch** block to display the exception message.

- Note that the **focus** is set to the **PriceTextBox** control, although the error may be in the quantity purchased.
- In the next chapter you will learn how to determine which TextBox has the invalid data.

Enabling and Disabling Controls

Controls such as buttons can be enabled and disabled (grayed out) through the **Enabled** property.

A typical approach is to have the **Reset** button disabled on startup of the system. When the **Compute** button is clicked and the calculations are displayed, the **Reset**command button is enabled at that time.

Example (assume the **ResetButton** control is disabled at design time).

```
    'Enable/disable buttons

    ComputeButton.Enabled = False

    ResetButton.Enabled = True
```

In-Class Exercise

Modify the program and sub procedure by setting button properties.

- At the design level, set the **Enabled** property to **False** for the **Reset** button on the form.
- In the sub procedure for the **Click** event of the **Compute** button, set the **Reset** button to be enabled, and the **Compute** button to be disabled as shown above. This should be coded at the end of the **Try** coding block, but just before the **Catch** statement.

```
    'Other tasks to code
```

```
    'Enable/disable buttons

    ComputeButton.Enabled = False

    ResetButton.Enabled = True

Catch ex As Exception
```

Reset Button Click Event

The **Reset** button should clear the textboxes, disable the **Reset** button, enable the **Compute** button, and set the focus back to the first TextBox.

- Begin by typing remarks into the sub procedure like this.

```
Private Sub ResetButton_Click(ByVal sender As System.Object, ByVal e As S
ystem.EventArgs) Handles ResetButton.Click

    'Clear all TextBox controls

     'Enable/disable buttons

     'Set focus to the BookTitleTextBox

 End Sub
```

- Now write the code to perform the required tasks. You should have mastered this from your previous exercises. Using the **With-End With** statement reduces the amount of typing required.

```
    Private Sub ResetButton_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles ResetButton.Click

        'Clear the text box controls

        BookTitleTextBox.Clear()

        ISBNTextBox.Clear()

        PriceTextBox.Clear()

        QuantityTextBox.Clear()

        SubtotalTextBox.Clear()

        SalesTaxTextBox.Clear()

        TotalDueTextBox.Clear()
```

```
        'Enable/disable buttons
        ComputeButton.Enabled = True
        ResetButton.Enabled = False

        'Set the focus to the book title text box control
        BookTitleTextBox.Focus()
    End Sub
```

Exit Button Click Event

The **Exit** command button should terminate the program. Use the **Me.Close()** statement.

```
    Private Sub CxitButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitButton.Click
        'Close the form
        Me.Close()
    End Sub
```

Sums, Counts, and Averages

Sums – programs often need to sum up numbers. For example, a cash register program that sums up the amount due for a sale at a grocery store. In this case we are computing a **total** by adding the **cost of an item** purchased to the **total amount** for every item (total of the sale).

Example: **TotalAmountDecimal += ItemCostDecimal**

Counts – sometimes there is a need to count how many times something happens. There are two approaches that can be taken:

- If there is a TextBox that stores the quantity of an item, for example, the quantity sold to a customer, then you use the **Sum** technique given above.

  Example: **TotalQuantityInteger += QuantitySoldInteger**

- If there is no TextBox, then your code will need to **count by one** each time, adding one to a memory variable.

Example: **TotalQuantityInteger += 1**

Averages are computed by dividing a **total** by a **count**.

Example: **AverageAmountValueDecimal = TotalAmountDecimal / TotalQuantityInteger**

Suppose that we need to display the total quantity of books sold along with the total dollar value of sales and the average value of each book sold, then display these values with in a MessageBox. This next section explains how to proceed.

Summing the Total Sales and Counting the Number of Books Sold

Because the total sales and count of books sold must be saved after every execution of the Compute button's click event sub procedure, you need to declare two module-level variables to store these values. The scope needs to be at the module-level for these values in order to retain their value as long as the program is executing.

- Add module-level declarations for two variables as shown.

```
 'Project:  Ch03VBUniversity
'D. Bock
'Today's Date

Option Strict On

Public Class Books
    'Declare module-level variables and constants
    Private TotalQuantityInteger As Integer
    Private TotalSalesDecimal As Decimal
```

Each time the application user clicks the **Compute** Button your program must accumulate the **TotalQuantityInteger** and **TotalSalesDecimal** values with the assignment statements shown here. These statements can be added to the code for the **Compute** Button after the code that enables/disables buttons.

```
    'Accumulate total sales and total books sold.
    TotalQuantityInteger += QuantityInteger
    TotalSalesDecimal += TotalDueDecimal
```

Calculating an Average

Calculating an average by dividing a sum of some value by the count of the number of times a value occurred.

The formula to calculate the average book sold value is:

```
AverageSaleDecimal = TotalSalesDecimal / TotalQuantityInteger
```

The Exception Class – Multiple Catch Blocks

An exception thrown by a program is a member of the **Exception** class of objects.

Exceptions have properties used to determine the object causing the error (**Source** property), location of the exception in your code (**Stack-Trace** property), type of exception, and cause (**Message** property).

You can have **multiple Catch blocks** in a **Try-Catch** coding block to handle different exceptions; but only ONE Catch block will execute—the first one with a matching exception.

This table lists just a few of the exceptions you can trap.

Table 3.6

Exception	Description
FormatException	Failure of data conversion for numeric data through use of Integer.Parse, or some similar conversion.
InvalidCastException	Failure to conversion operation caused by loss of significant digits or some other illegal conversion.
ArithmeticException	Calculation error such as divide by zero.
OutOfMemoryException	Not enough memory to create an object.
Exception	The generic "catch all" exception.

- Exceptions have a **hierarchy** from specific to general.
- The hierarchy is covered in the MSDN help.
- Code the Catch blocks from **specific to general**; otherwise, the **Exception** (generic) will catch all errors and none of the other Catch blocks will ever execute.

Example coding of multiple exceptions:

```
Catch exArithmeticException As ArithmeticException
```

```
    MessageBox.Show("No books have been sold yet", "Zero Sales Message",
MessageBoxButtons.OK, MessageBoxIcon.Information)

Catch ex As Exception

    MessageBox.Show("Unexpected Error-inform the system
administrator", "Unknown Error in Totals Button", MessageBoxButtons.OK,
MessageBoxIcon.Error)

End Try
```

Totals Button Click Event

Code the **Totals** Button click event as shown here.

- Calculates the average sale by dividing the total sales by the total quantity of books sold.
- Produces 3 lines of output.
- Formats all output values displayed.
- Catches error of not having sold a book.
- Catches generic errors in case an unforeseen error occurs.

```
    Private Sub TotalsButton_Click(ByVal sender As System.Object, ByVal e
 As System.EventArgs) Handles TotalsButton.Click

        Try

            'Display the total sales, total books sold,

            'and average book value in a message box.

            Dim AverageSaleDecimal As Decimal = TotalSalesDecimal /
TotalQuantityInteger

            Dim MessageString As String = "Total Sales: " &
TotalSalesDecimal.ToString("C") & ControlChars.NewLine & "Total Books
Sold: " & TotalQuantityInteger.ToString("N0") & ControlChars.NewLine
& "Average Book Value: " & AverageSaleDecimal.ToString("C")


            MessageBox.Show(MessageString, "Totals and Averages",
MessageBoxButtons.OK, MessageBoxIcon.Information)


        Catch exArithmeticException As ArithmeticException

            MessageBox.Show("No books have been sold yet", "Zero Sales
Message", MessageBoxButtons.OK, MessageBoxIcon.Information)

        Catch ex As Exception

            MessageBox.Show("Unexpected Error-inform the system
administrator", "Unknown Error in Totals Button", MessageBoxButtons.OK,
MessageBoxIcon.Error)
```

```
        End Try

    End Sub
```

Test the project.

- Run the project and click the **Totals** button – the message box should display the **No books have been sold yet** message.
- Now enter a couple of book sales – click **Totals** again and the message box should display the total sales in dollars, total books sold, and average book sale value.

Now you have learned the concepts and techniques needed to complete your next programming assignment. Happy Computing!

Solution to In-Class Exercise

```
'Project:  Ch03VBUniversity
'D. Bock
'Today's Date
Option Strict On
Public Class Books
    'Declare module-level variables and constants
    Private TotalQuantityInteger As Integer
    Private TotalSalesDecimal As Decimal

    Private Sub ComputeButton_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles ComputeButton.Click
        Try
            'Declare constants
            '7.25 percent sales tax rate
            Const SALES_TAX_RATE_SINGLE As Single = 0.0725
           'Declare variables
            Dim SubtotalDecimal, SalesTaxDecimal,
TotalDueDecimal As Decimal

            'Declare variables and convert values from
            'textbox controls to memory
```

```
            Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text
, Globalization.NumberStyles.Currency)

            Dim QuantityInteger As Integer = Integer.Parse(QuantityTextBo
x.Text, Globalization.NumberStyles.Number)


            'Process - Compute values

            'Subtotal = price times the quantity of books

            SubtotalDecimal = PriceDecimal * QuantityInteger


            'Sales tax = sales tax rate times the subtotal

            SalesTaxDecimal
= Decimal.Round(Convert.ToDecimal(SubtotalDecimal *
SALES_TAX_RATE_SINGLE), 2)


            'Total due = subtotal plus sales tax

            TotalDueDecimal = SubtotalDecimal + SalesTaxDecimal


            'Display output formatted as currency

            SubtotalTextBox.Text = SubtotalDecimal.ToString("C")

            SalesTaxTextBox.Text = SalesTaxDecimal.ToString("N")

            TotalDueTextBox.Text = TotalDueDecimal.ToString("C")


            'Other tasks to code

            'Enable/disable buttons

            ComputeButton.Enabled = False

            ResetButton.Enabled = True


            'Accumulate total sales and total books sold.

            TotalQuantityInteger += QuantityInteger

            TotalSalesDecimal += TotalDueDecimal


        Catch ex As Exception
```

```
            MessageBox.Show("Error in either Book Price or Quantity
Purchased", "Data Entry Error", MessageBoxButtons.OK,
MessageBoxIcon.Error)

            PriceTextBox.Focus()

        End Try

    End Sub


    Private Sub ResetButton_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles ResetButton.Click

        'Clear the text box controls

        BookTitleTextBox.Clear()

        ISBNTextBox.Clear()

        PriceTextBox.Clear()

        QuantityTextBox.Clear()

        SubtotalTextBox.Clear()

        SalesTaxTextBox.Clear()

        TotalDueTextBox.Clear()


        'Enable/disable buttons

        ComputeButton.Enabled = True

        ResetButton.Enabled = False


        'Set the focus to the book title text box control

        BookTitleTextBox.Focus()

    End Sub


    Private Sub ExitButton_Click(ByVal sender As System.Object, ByVal e A
s System.EventArgs) Handles ExitButton.Click

        'Close the form

        Me.Close()

    End Sub
```

```
    Private Sub TotalsButton_Click(ByVal sender As System.Object, ByVal e
 As System.EventArgs) Handles TotalsButton.Click

        Try

            'Display the total sales, total books sold,

            'and average book value in a message box.

            Dim AverageSaleDecimal As Decimal = TotalSalesDecimal /
TotalQuantityInteger

            Dim MessageString As String = "Total Sales: " &
TotalSalesDecimal.ToString("C") & ControlChars.NewLine & "Total Books
Sold: " & TotalQuantityInteger.ToString("N0") & ControlChars.NewLine
& "Average Book Value: " & AverageSaleDecimal.ToString("C")



            MessageBox.Show(MessageString, "Totals and Averages",
MessageBoxButtons.OK, MessageBoxIcon.Information)



        Catch exArithmeticException As ArithmeticException

            MessageBox.Show("No books have been sold yet", "Zero Sales
Message", MessageBoxButtons.OK, MessageBoxIcon.Information)

        Catch ex As Exception

            MessageBox.Show("Unexpected Error-inform the system
administrator", "Unknown Error in Totals Button", MessageBoxButtons.OK,
MessageBoxIcon.Error)

        End Try

    End Sub

 End Class

End Class
```