Chapter 3
Variables, Constants and Calculations

Programming In
Visual Basic.NET
Variables & Constants

• Variable
  – Memory locations that hold data that can be changed during project execution
  – Ex: intHoursWorked

• Named Constant
  – Memory locations that hold data that cannot be changed during project execution
  – Ex: curSALESTAXPERCENT
Constants

• Named
  – User defined

• Intrinsic
  – System defined within Visual Studio
  – In Chapter 23 we used the Intrinsic Color Constants
Declaration

- Variables and Named Constants must be declared before being used in code
- When you declare a Variable or Named Constant in VB
  - Reserves an area of memory
  - Assigns it a name called an Identifier
- Declaration statements are coded either
  - Beginning of a procedure
  - General Declarations of a module
Declaration Statements

- DIM is used to declare Variables
- CONST is used to declare Named Constants
- Declaration includes
  - Name, follow Naming Convention Rules
  - Data Type
  - Required Value for Constants
  - Optional Initial Value for Variables
## Data Types

<table>
<thead>
<tr>
<th>Visual Basic type</th>
<th>Common language runtime type</th>
<th>Nominal storage allocation</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>System.Boolean</td>
<td>2 bytes</td>
<td>True or False.</td>
</tr>
<tr>
<td>Byte</td>
<td>System.Byte</td>
<td>1 byte</td>
<td>0 through 255 (unsigned).</td>
</tr>
<tr>
<td>Char</td>
<td>System.Char</td>
<td>2 bytes</td>
<td>0 through 65535 (unsigned).</td>
</tr>
<tr>
<td>Date</td>
<td>System.DateTime</td>
<td>8 bytes</td>
<td>00:00:00 on January 1, 0001 through 11:59:59 PM on December 31, 9999.</td>
</tr>
<tr>
<td>Decimal</td>
<td>System.Decimal</td>
<td>16 bytes</td>
<td>0 through +/-79,228,162,514,264,337,593,543,950,335 with no decimal point; 0 through +/-7,0223162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.000000000000000000000000001 (±1E-28).</td>
</tr>
<tr>
<td>Double (double-precision floating-point)</td>
<td>System.Double</td>
<td>8 bytes</td>
<td>-1.79769313486231570E+308 through +1.79769313486231570E+308 for negative values; +1.04066545941246544E-324 for positive values.</td>
</tr>
<tr>
<td>Integer</td>
<td>System.Int32</td>
<td>4 bytes</td>
<td>-2,147,483,648 through 2,147,483,647.</td>
</tr>
<tr>
<td>Object</td>
<td>System.Object (class)</td>
<td>4 bytes</td>
<td>Any type can be stored in a variable of type Object.</td>
</tr>
<tr>
<td>Short</td>
<td>System.Int16</td>
<td>2 bytes</td>
<td>-32,768 through 32,767.</td>
</tr>
<tr>
<td>Single (single-precision floating-point)</td>
<td>System.Single</td>
<td>4 bytes</td>
<td>-3.4028235E+38 through -1.401298E45 for negative values; 1.401298E-45 through 3.4028235E+38 for positive values.</td>
</tr>
<tr>
<td>String (variable-length)</td>
<td>System.String (class)</td>
<td>Depends on implementing platform</td>
<td>0 to approximately 2 billion Unicode characters.</td>
</tr>
<tr>
<td>User-Defined Type (structure)</td>
<td>(inherits from System.ValueType)</td>
<td>Depends on implementing platform</td>
<td>Each member of the structure has a range determined by its data type and independent of the ranges of the other members.</td>
</tr>
</tbody>
</table>
Data Types – Prefixes

- Boolean  bln
- Byte     byt
- Char     chr
- Date     dat
- String   str
- Decimal  dec
- Object   depends on type of object

- Short    sht
- Integer  int
- Long     lng
- Single   sng
- Double   dbl
Declaration Examples

**Dim strName, strSSN** As String
Dim intAge As Short
Dim decPayRate As Decimal = 8.25
Dim datHireDate As Date
Dim blnInsured As Boolean
Dim lngPopulation As Long

Dim objRS

Const decDiscount_RATE As Decimal = .15

**Note:** Constants are named using all uppercase letters EXCEPT the prefix.

Variables declared without a data type default to an Object data type.
Naming Rules

• Names can consist of:
  – Letters
  – Digits
  – Underscore

• They must begin with a letter or an underscore
• They can not contain a space
• They may not be a reserved word
• Identifiers are not case sensitive
• Length limit is 16,383 characters
Type-Declaration Characters

- Append single character to the end of the Constant's Value to indicate the Data Type

  - Short   S
  - Integer I
  - Long    L
  - Decimal D
  - Single  F
  - Double  R

Const mdecDISCOUNT_RATE As Decimal = 0.15D

- This is required if you are using Option Strict On
Variables – Scope & Lifetime

- **Global/Public** (use sparingly and cautiously)
  - Available to all modules and procedures of Project
  - Initialized at start of Project
- **Module Private - Public**
  - Available to one module and all procedures within that module
  - Initialized 1st time the Form is loaded
- **Local Private**
  - Available only to the procedure it is declared in
  - Initialized every time the Procedure runs
- **Block Private** (not used until later in this course)
  - Available only to the block of code inside a procedure it is declared in
  - Initialized every time the Procedure runs. If you enter the block more than once during the procedure, a block variable retains its previous value. To avoid unexpected results in such a case, it is wise to initialize the variable each time.
Scope Declaring & Naming

- **Global/Public**
  - Declare in General Declarations as Public
    - Public gstrName as String

- **Module/Private**
  - Declare in Module’s General Declarations as Private
    - Dim mstrName as String

- **Local**
  - Declare in Event Procedures
    - Dim strName as String
Declaring Module Level Variables Example

```vbscript
' Uses variables, constants, calculations, error handling, and a message box to the user.
'Folder: Ch0302

Option Strict On

Public Class frmBooksale
  Inherits System.Windows.Forms.Form

Windows Form Designer generated code

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

Private Sub btnCalculate_Click(ByVal sender As System.Object,
Calculations

- Calculations can be performed using properties of certain objects, variables, constants, and numeric literals
- Do Not use Strings in calculations
- Values from Text property of Text Boxes
  - Are Strings, even if they contain numeric data
  - Must be converted to a Numeric Data Type
Conversion Functions

- Functions perform an action and return a value
- Expression to operate on is called the Argument
- Conversion Functions convert arguments into a numeric value of the correct data type
- Conversion Functions on Text Boxes fail if user enters nonnumeric data or leaves the Text Box blank
Conversion Functions (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Convert To</th>
</tr>
</thead>
<tbody>
<tr>
<td>CInt **</td>
<td>Integer</td>
</tr>
<tr>
<td>CDec</td>
<td>Decimal</td>
</tr>
<tr>
<td>CStr</td>
<td>String</td>
</tr>
</tbody>
</table>

** CInt rounds a little differently. It always rounds to the even number when the decimal is equal to .5.

Example:

<table>
<thead>
<tr>
<th>Value</th>
<th>Return</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CInt(“0.5”)</td>
<td>0</td>
<td>Down</td>
</tr>
<tr>
<td>CInt(“1.5”)</td>
<td>2</td>
<td>Up</td>
</tr>
<tr>
<td>CInt(“2.5”)</td>
<td>2</td>
<td>Down</td>
</tr>
<tr>
<td>CInt(“2.51”)</td>
<td>3</td>
<td>Up</td>
</tr>
<tr>
<td>CInt(“2.51”)</td>
<td>3</td>
<td>Up</td>
</tr>
</tbody>
</table>
Conversion Examples
(also review examples p 96)

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument To Be Acted Upon</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intQuantity</code></td>
<td><code>CInt(txtQuantity.Text)</code></td>
</tr>
<tr>
<td><code>decPrice</code></td>
<td><code>CDec(txtPrice.Text)</code></td>
</tr>
<tr>
<td><code>intWholeNumber</code></td>
<td><code>CInt(decFractionalValue)</code></td>
</tr>
<tr>
<td><code>decDollars</code></td>
<td><code>CDec(intDollars)</code></td>
</tr>
<tr>
<td><code>strValue</code></td>
<td><code>CStr(decValue)</code></td>
</tr>
</tbody>
</table>
# Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>\</td>
<td>Integer Division</td>
</tr>
<tr>
<td>Mod</td>
<td>Modulus (division's remainder)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>

© 2001 by The McGraw-Hill Companies, Inc. All rights reserved.
Mathematical Order of Operations

• Computers solve math formulas based on a specific order 1st, then left to right
  1. Parentheses
  2. Exponentiation
  3. Multiplication & Division
  4. Integer Division
  5. Modulus
  6. Addition & Subtraction
Mathematical Examples

- Note the use of parentheses to control

\[
\begin{align*}
3 + 4 \times 2 &= 11 & \text{Multiply then add} \\
(3 + 4) \times 2 &= 14 & \text{Parentheses control: add then multiply} \\
8 \div 4 \times 2 &= 4 & \text{Same level, left to right: divide then multiply}
\end{align*}
\]
Assignment Statement

• The = sign is an assignment operator.

```vbnet
Dim decTotal As Decimal
Dim intHours As Integer = 40
Dim intRate As Integer = 20

decTotal = intHours * intRate
```

• In the example above the calculation on the right side of the equal sign is calculated and then the assignment is made into decTotal.
Assignment Statement

• What happens when the variables contain high values but are still small enough to not generate an error when they are initialized and are used in a calculation?

Dim decTotal As Decimal  
Dim intHours As Integer = 400000000  
Dim intRate As Integer = 20

decTotal = intHours * intRate

• In the example above the calculation on the right side of the equal sign is calculated and an overflow error is generated.

• The solution is to Type Cast one of the variables in the calculation.
## More Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^=</td>
<td>Raises the value of a variable to the power of an expression and assigns the result back to the variable.</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplies the value of a variable by the value of an expression and assigns the result to the variable.</td>
</tr>
<tr>
<td>/=</td>
<td>Divides the value of a variable by the value of an expression and assigns the result to the variable.</td>
</tr>
<tr>
<td>=</td>
<td>Divides the value of a variable by the value of an expression and assigns the integer result to the variable.</td>
</tr>
<tr>
<td>+=</td>
<td>Adds the value of an expression to the value of a variable and assigns the result to the variable. Also concatenates a <strong>String</strong> expression to a <strong>String</strong> variable and assigns the result to the variable.</td>
</tr>
<tr>
<td>-=</td>
<td>Subtracts the value of an expression from the value of a variable and assigns the result to the variable.</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Concatenates a <strong>String</strong> expression to a <strong>String</strong> variable and assigns the result to the variable.</td>
</tr>
</tbody>
</table>
Option Explicit

- On by default - should be left on
- If turned off
  - Variables can be used without first being declared
  - They will be defined by VB as data type Object
- To turn off
  - Code `Option Explicit Off` or `Option Explicit` in General Declarations
  - Set in Project Properties dialog box
Option Strict

• Off by default - should be turned on
• If turned on
  – VB becomes strongly typed language
  – Will not allow implicit conversions from a wider data type to a narrower one or between String and numeric data types
• To turn on
  – Code **Option Strict On** in General Declarations
  – Set in Project Properties dialog box
FormatCurrency Function

• General Form
  – FormatCurrency(NumericExpression)
• Returns a string of characters formatted as dollars and cents
• Includes a Dollar Sign, commas, and 2 decimal places by default
• Value returned is a String and can no longer be used in calculations
FormatNumber Function

• General Form

• Formats with commas and specified number of decimal places (2 by default)
FormatPercent Function

• General Form

• Returns a string of characters formatted as a percent

• Multiplies the argument by 100, adds a percent sign and rounds to 2 decimal places by default
FormatDateTime Function

• General Form
  – FormatDateTime(Expression [, Named Format] )

• Expression can be:
  – String that holds a date or time
  – Date type variable
### Named Formats - FormatDateTime Function

<table>
<thead>
<tr>
<th>Named Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateFormat.GeneralDate</td>
<td>2/28/99 6:01:24 PM</td>
</tr>
<tr>
<td>DateFormat.ShortDate</td>
<td>2/28/99</td>
</tr>
<tr>
<td>DateFormat.LongDate</td>
<td>Sunday, February 28, 1999</td>
</tr>
<tr>
<td>DateFormat.ShortTime</td>
<td>18:01 (24 Hour Clock)</td>
</tr>
<tr>
<td>DateFormat.LongTime</td>
<td>6:01:24 PM</td>
</tr>
</tbody>
</table>
ToString Method

• Returns a String that represents the current Object.
  Object.ToString([Format])

• The next two slides show the format characters for Numeric and Date strings.
## Standard Numeric Format Strings

<table>
<thead>
<tr>
<th>Format specifier</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C or c</td>
<td>Currency</td>
<td>The number is converted to a string that represents a currency amount. The conversion is controlled by the currency format information of the <code>NumberFormatInfo</code> object used to format the number. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default currency precision given by the <code>NumberFormatInfo</code> is used.</td>
</tr>
<tr>
<td>D or d</td>
<td>Decimal</td>
<td>This format is supported for integral types only. The number is converted to a string of decimal digits (0-9) prefixed by a minus sign if the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.</td>
</tr>
<tr>
<td>E or e</td>
<td>Scientific (exponential)</td>
<td>The number is converted to a string of the form &quot;d.ddddE±ddd&quot; or &quot;d.ddddE±ddd&quot;, where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used. The case of the format specifier indicates whether to prefix the exponent with an 'E' or an 'e'. The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.</td>
</tr>
<tr>
<td>F or f</td>
<td>Fixed-point</td>
<td>The number is converted to a string of the form &quot;d.d&quot;., where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the <code>NumberFormatInfo</code> is used.</td>
</tr>
<tr>
<td>G or g</td>
<td>General</td>
<td>The number is converted to the most compact decimal form, using fixed or scientific notation. The precision specifier determines the number of significant digits in the resulting string. If the precision specifier is omitted, the number of significant digits is determined by the type of number being converted:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Int16 or UInt16: 5 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Int32 or UInt32: 10 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Int64 or UInt64: 19 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Single: 7 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Double: 15 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Decimal: 26 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trailing zeros after the decimal point are removed, and the resulting string contains a decimal point only if required. The resulting string uses fixed-point format if the exponent of the number (as produced by the 'E' format) is less than the number of significant digits, and greater than or equal to -1. Otherwise, the resulting string uses scientific format, and the case of the format specifier controls whether the format is an 'E' or an 'e'.</td>
</tr>
<tr>
<td>N or n</td>
<td>Number</td>
<td>The number is converted to a string of the form &quot;d.d,ddd,d,ddd,&quot;, where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. Thousand separators are inserted between each group of three digits to the left of the decimal point. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the <code>NumberFormatInfo</code> is used.</td>
</tr>
<tr>
<td>P or p</td>
<td>Percent</td>
<td>The number is converted to a string that represents a percent as defined by the <code>NumberFormatInfo.PercentPositivePattern</code> property or the <code>NumberFormatInfo.PercentNegativePattern</code> property. If the number is negative, the string produced is defined by the <code>NumberFormatInfo.PercentNegativePattern</code> property and starts with a minus sign. The converted number is multiplied by 100 in order to be presented as a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by <code>NumberFormatInfo</code> is used.</td>
</tr>
<tr>
<td>R or r</td>
<td>Round-trip</td>
<td>The round-trip specifier guarantees that a numeric value converted to a string will be parsed back into the same numeric value. When a numeric value is formatted using this specifier, it is first tested using the general format, with 15 spaces of precision for a <code>Double</code> and 2 spaces of precision for a <code>Single</code>. If the value is successfully parsed back to the same numeric value, then it is formatted using the general format specifier. However, if the value is not successfully parsed back to the same numeric value, then the value is formatted using 17 digits of precision for a <code>Double</code> and 9 digits of precision for a <code>Single</code>. Although a precision specifier can be appended to the round-trip format specifier, it is ignored. Round trips are given precedence over precision when using this specifier. This format is supported by floating-point types only.</td>
</tr>
<tr>
<td>X or x</td>
<td>Hexadecimal</td>
<td>The number is converted to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for the hexadecimal digits greater than 9. For example, use &quot;X&quot; to produce &quot;0XDEF&quot;, and &quot;x&quot; to produce &quot;0xedf&quot;. The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier. This format is supported for integral types only.</td>
</tr>
</tbody>
</table>
# Date and Time Format Strings

<table>
<thead>
<tr>
<th>Format specifier</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Short date pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.ShortDateFormat</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>D</td>
<td>Long date pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.LongDateFormat</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>t</td>
<td>Short time pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.ShortTimePattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>T</td>
<td>Long time pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.LongTimePattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>f</td>
<td>Full date/time pattern</td>
<td>Displays a combination of the long date and short time patterns, separated by a space.</td>
</tr>
<tr>
<td>F</td>
<td>Full date/time pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.FullDateTimePattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>g</td>
<td>General date/time pattern</td>
<td>Displays a combination of the short date and short time patterns, separated by a space.</td>
</tr>
<tr>
<td>G</td>
<td>General date/time pattern</td>
<td>Displays a combination of the short date and long time patterns, separated by a space.</td>
</tr>
<tr>
<td>M or m</td>
<td>Month day pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.MonthDayPattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>R or r</td>
<td>RFC1123 pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.RFC1123Pattern</a> property associated with the current thread or by a specified format provider. This is a defined standard and the property is read-only; therefore, it is always the same regardless of the culture used, or the format provider supplied. The property references the <a href="#">CultureInfo.InvariantCulture</a> property and follows the custom pattern 'ddd, dd MMMM yyyy HH:mm:ss GMT'. Note that the 'M' in 'GMT' needs an escape character so it is not interpreted.</td>
</tr>
<tr>
<td>s</td>
<td>Sortable date/time pattern; conforms to ISO 8601</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.SortableDateTimePattern</a> property associated with the current thread or by a specified format provider. The property references the <a href="#">CultureInfo.InvariantCulture</a> property, and the format follows the custom pattern 'yyyy-MM-dd HH:mm:ssZ'.</td>
</tr>
<tr>
<td>u</td>
<td>Universal sortable date/time pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.UniversalSortableDateTimePattern</a> property associated with the current thread or by a specified format provider. Because it is a defined standard and the property is read-only, the pattern is always the same regardless of culture or format provider. The format follows the custom pattern 'yyyy-MM-dd HH:mm:ssZ'.</td>
</tr>
<tr>
<td>U</td>
<td>Universal sortable date/time pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.FullDateTimePattern</a> property associated with the current thread or by a specified format provider. Note that the time displayed is for the Universal, rather than local time.</td>
</tr>
<tr>
<td>Y or yyyy</td>
<td>Year month pattern</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.YearMonthPattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
<tr>
<td>Any other single character</td>
<td>Unknown specifier</td>
<td>Displays a pattern defined by the <a href="#">DateTimeFormatInfo.YearMonthPattern</a> property associated with the current thread or by a specified format provider.</td>
</tr>
</tbody>
</table>
Lab 1

- Project Name: Format
Lab 1 Code

Option Strict On
Option Explicit On

Private Sub btnDisplay1_Click(...) Handles btnDisplay1.Click
    Dim decHold As Decimal = 12345.678D
    Dim decPercent As Decimal = 0.01D

    txtCurrency.Text = FormatCurrency(decHold)
    txtDate.Text = FormatDateTime(Now())
    txtNumber.Text = FormatNumber(decHold, 1)
    txtPercent.Text = FormatPercent(decPercent)
End Sub

Private Sub btnDisplay2_Click(...) Handles btnDisplay2.Click
    Dim decHold As Decimal = 12345.678D
    Dim decPercent As Decimal = 0.01D

    txtCurrency.Text = decHold.ToString("C")
    txtDate.Text = Now.ToString("G")
    txtNumber.Text = decHold.ToString("N1")
    txtPercent.Text = decPercent.ToString("P")
End Sub
Lab 2

- Project Name: BookSale

frmBooksale

btnCalculate

btnClear

btnExit

© 2001 by The McGraw-Hill Companies, Inc. All rights reserved.
Lab 2 Code

Option Strict On
Option Explicit On

Const mdecDISCOUNT_RATE As Decimal = 0.15D

Private Sub btnCalculate_Click(...) Handles btnCalculate.Click
    Dim intQuantity As Integer
    Dim decPrice, decExtendedPrice, decDiscount, decDiscountedPrice As Decimal
    intQuantity = CInt(txtQuantity.Text)
    decPrice = CDec(txtPrice.Text)
    decExtendedPrice = intQuantity * decPrice
    decDiscount = decExtendedPrice * mdecDISCOUNT_RATE
    decDiscountedPrice = decExtendedPrice - decDiscount
    lblExtendedPrice.Text = FormatCurrency(decExtendedPrice)
    lblDiscount.Text = FormatNumber(decDiscount, 2)
    lblDiscountedPrice.Text = FormatCurrency(decDiscountedPrice)
End Sub

Private Sub btnClear_Click(...) Handles btnClear.Click
    txtTitle.Clear()
    txtPrice.Clear()
    lblExtendedPrice.Text = ""
    lblDiscount.Text = ""
    lblDiscountedPrice.Text = ""
    With txtQuantity
        .Clear()
        .Focus()
    End With
End Sub

Private Sub btnExit_Click(...) Handles btnExit.Click
    Me.Close()
End Sub
Handling Exceptions

• Exceptions occur when user enters unexpected/invalid data and program code does not anticipate this possibility, such as
  – User enters nonnumeric data in Text Box and code attempts to run a Numeric Conversion Function
  – User enters data that results in division by zero
Try/Catch Blocks

• Used to catch and handle exceptions; referred to as error trapping or handling
• Enclose statements that might cause an error within Try/Catch Block
  – If an error occurs control is transferred to the Catch Block
  – Include a **Finally** statement to indicate code that should execute last whether or not an exception occurred
Try Block - General Form

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

See p 123 for list of common Exception Classes
Try Block - Example 1
Catches All Exceptions

Try
    intQuantity=CInt(txtQuantity.Text)
    lblQuantity.Text=CStr(intQuantity)
Catch
    lblMessage.Text="Error in input data."
End Try
Try Block - Example 2
Catches Specific Exception

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

Conversion exception, usually caused by nonnumeric or blank data
Try Block - Example 3
Catches Multiple Specific Exceptions

Try
  statements that may cause errors
Catch MyErr as InvalidCastException
  error messages and statements for nonnumeric data
Catch MyErr as ArithmeticException
  error messages and statements for calculation problems
Catch MyErr as Exception
  error messages and statements for any other exception
End Try

Notice you can use the same name MyErr with each Catch or you could use different Names with each one.
MessageBox Object

• Use Show Method of MessageBox to display special type of window
• Arguments of Show method
  – Message to display
  – Optional Title Bar Caption
  – Optional Button(s)
  – Optional Icon
MessageBox Syntax

• The MessageBox is an Overloaded Method
  – Signatures correspond to the Argument list
  – There are multiple Signatures to choose from
  – Arguments must be included to exactly match one of the predefined Signatures

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

- OK
- OKCancel
- RetryCancel
- YesNo
- YesNoCancel
- AbortRetryIgnore
MessageBoxIcon Constants

- Asterisk
- Error
- Exclamation
- Hand
- Information
- None
- Question
- Stop
- Warning
Lab 3

- Project Name: ErrorHandling
Lab 3 Code

Option Strict On
Option Explicit On

Private Sub btnCalc_Click(...) Handles btnCalc.Click
    Try
        Dim decPay As Decimal
        decPay = CDec(Me.txtHours.Text) * CInt(Me.txtRate.Text)
        MessageBox.Show(decPay.ToString, "Salary", MessageBoxButtons.OK, _
                        MessageBoxIcon.Information)
    Catch EH As InvalidCastException
        MessageBox.Show("The textboxes can not contain an empty string.", _
                        "Invalid Cast Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch EH As Exception
        MessageBox.Show(EH.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
Counting & Accumulating Sums

• Must use Module/Form level variables since Local/Event level variables reset to 0 each time the procedure is called

• Summing
  – \( \text{mdecOrderTotal} = \text{mdecOrderTotal} + \text{decItemPrice} \)

• Counting
  – \( \text{mintNumItems} = \text{mintNumItems} + 1 \)
  – \( \text{mintNumItems} += \text{mintNumItems} \)

• Averaging
  – \( \text{mdecAveSale} = \text{mdecOrderTotal} / \text{mintNumItems} \)