Chapter Objectives

- Define a variable
- Distinguish between variables, constants, and control objects
- Differentiate among the various data types
  - Differentiate accessing "by value" or "by reference"
- Apply naming conventions incorporating standards and indicating the data type
- Declare variables and constants
- Select the appropriate scope for a variable
- Convert text input to numeric values

Chapter Objectives – cont’d

- Perform calculations using variables and constants
- Convert between numeric data types
- Format values for output using the `ToString` method
  - The `string.format` method (quick intro…more later)
- Use `try/catch` blocks for error handling
- Display message boxes with error messages
- Accumulate sums and generate counts
Data - Variables and Constants

- **Variable**
  - A named location in memory that holds data that *can* change during project execution
  - Example: Customer's name

- **Constant**
  - Memory locations that hold data that *cannot* change during execution
  - Examples: Company name or sales tax rate

Variables and Constants – Examples

- Sample declaration statements

  ```csharp
  //Declare a string variable.
  string nameString;

  //Declare integer variables.
  int counterInteger;
  int maxInteger = 100;

  //Declare a named constant.
  const decimal DISCOUNT_RATE_Decimal = .15M;
  ```

Data Types – 1

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>true or false values</td>
</tr>
<tr>
<td>byte</td>
<td>0 to 255, binary data</td>
</tr>
<tr>
<td>char</td>
<td>Single Unicode character</td>
</tr>
<tr>
<td>DateTime</td>
<td>1/1/0001 00:00:00 through 12/31/9999 23:59:59</td>
</tr>
<tr>
<td>decimal</td>
<td>Decimal fractions, such as dollars &amp; cents, precision of 28 digits</td>
</tr>
<tr>
<td>float</td>
<td>Single-precision floating-point numbers, six digits of accuracy</td>
</tr>
<tr>
<td>double</td>
<td>Double-precision floating-point numbers, 14 digits of accuracy</td>
</tr>
<tr>
<td>short</td>
<td>Small integer in the range –32,768 to +32,767</td>
</tr>
<tr>
<td>int</td>
<td>Whole numbers in the range –2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>Larger whole numbers</td>
</tr>
<tr>
<td>string</td>
<td>Alphanumeric data: letters, digits and other characters</td>
</tr>
<tr>
<td>object</td>
<td>Any type of data</td>
</tr>
</tbody>
</table>
### Data Types – 2

- **Common types of variables and constants**
  - string
  - int
  - decimal

- **Guideline for use**
  - Data used in calculations must be numeric
    - decimal – any decimal fraction including dollars
    - int – whole number calculations
    - float and double – scientific applications
  - Text characters and numbers not used in calculations
    - string – including phone or social security numbers

### Naming Rules

- May contain letters, digits, and underscores
- Must begin with a letter or underscore
- Cannot contain spaces or periods
- Cannot be reserved words (*keywords*)
  - Examples: `print`, `name`, `value`
- Case sensitive
  - `sumInteger`, `SumInteger`, `suminteger` and `SUMINTEGER` are different variables

### Naming Conventions

- *Should* follow naming conventions
  - Identifiers must be meaningful
    - Clearly indicate the purpose
    - Do not abbreviate
    - Do not use X or Y
  - Include the class (data type)
  - Use camel casing for variables
    - Begin with a lowercase letter
    - Capitalize each successive word
  - Use uppercase for constants
    - Separate words with underscores
### Constants: Named and Intrinsic

- Constants contain a value that does not change
- Intrinsic – a.k.a. keyword identifiers
  - Built into the environment, not defined
- Named
  - Defined by programmer

### Named Constants

- Declare using keyword `const`
- Assign name, data type, value
- Data type declared and data type of value must match
- Two advantages to using
  - Easier to read
  - Precision is the same for all calculations
- Const Statement – Examples:

  ```csharp
  const string COMPANY_ADDRESS_String = "101 S. Main Street";
  const decimal SALES_TAX_RATE_Decimal = .08m;
  ```

### System-defined constants

- Declared in system class libraries
- Specify class name or group name and constant name (this is really known as an enumeration)
  - Example – `Color.Red` is the constant “Red” in the class “Color”
Declaring Variables

- Declare a variable
  - Specify data type and follow with an identifier
  - Assign an initial value (optional)

    ```csharp
    string customerNameString;
    string customerNameString = textBoxName.text;
    private int totalSoldInteger;
    int totalSoldInteger = 0;
    float temperatureFloat;
    float temperatureFloat = 32f;
    decimal priceDecimal;
    private decimal priceDecimal = 99.95m;
    ```

Use IntelliSense!!!

- IntelliSense helps in entering declaration statements
  - Type correct entry
  - List scrolls to correct section
  - When correct entry is highlighted
    - Press Enter (move to a new line of code), or Tab, spacebar, or double-click entry with mouse (stay on the same line of code)

Scope and Lifetime of Variables

- Visibility of a variable is its scope
  - Namespace
    - Also referred to as global
  - Class-level
  - Local
  - Block
- Scope of variable determined by where it is declared
Variable Lifetime

- Period of time a variable exists
  - Local or block variable
    - One execution of a method
    - After execution variables disappear, memory locations are released
  - Class-level variable
    - Entire time class is loaded (usually lifetime of project)
    - Variables maintain value for multiple executions of a method

Local Declarations

- Variable declared inside a method is local in scope
  - Known only to that method
- Must be declared prior to first use
  - All declarations should appear at the top of a method (after comments)

Class-Level Declarations

- Variables declared at class level can be used anywhere in that form’s class
- Place after opening brace for class, outside of any method
- Values remain in memory as long as the class exists in memory
Calculations

- Calculations are performed with
  - Variables
  - Constants
  - Properties of certain objects
    - i.e. Text property of text box or label
    - Character strings must be converted to correct data type first

Converting Strings to a Numeric Data Type

- Parse method converts the Text property of a control to a numeric value that can be used in calculations
- int.Parse converts text to an integer
- decimal.Parse converts text to a decimal
- Pass the text string to convert as an argument of the Parse method

```csharp
// Convert input values to numeric variables.
quantityInteger = int.Parse(quantityTextBox.Text);
priceDecimal = decimal.Parse(priceTextBox.Text);
// Calculate the extended price.
extendedPriceDecimal = quantityInteger * priceDecimal;
```

Using the Parse Methods

- Parse attempts to convert the string to the specified data type
  - Values such as blanks or nonnumeric characters cause an error
- Parse methods exist for long, float or double conversions

```csharp
quantityInteger = int.Parse(quantityTextBox.Text);
priceDecimal = decimal.Parse(priceTextBox.Text);
wholeNumberInteger = int.Parse(digitString);
```
### Converting to String

- Assignment must be of like types
  - Assign a string value to a string variable or property
  - Assign a numeric value to a numeric variable or property
- To assign the result of a calculation to the Text property of a control, use the `ToString` method

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

### Arithmetic Operations

<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>Modulus – remainder of division</td>
</tr>
</tbody>
</table>

### Arithmetic Operations

- Exponentiation
  - C# does not have an operator for exponentiation
  - Use `Math.Pow`
    - See Appendix B for Math methods
Order of Operations

- Hierarchy of operations or order of precedence in arithmetic expressions
  1. Any operation inside parentheses
  2. Multiplication and division
  3. Modulus
  4. Addition and subtraction

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4 * 2</td>
<td>11</td>
<td>Multiply then add</td>
</tr>
<tr>
<td>(3 + 4) * 2</td>
<td>14</td>
<td>Add then multiply</td>
</tr>
<tr>
<td>8 / 4 * 2</td>
<td>4</td>
<td>Divide then multiply</td>
</tr>
<tr>
<td>8 / (4 + 2)</td>
<td>1</td>
<td>Multiply then divide</td>
</tr>
</tbody>
</table>

Using Calculations in Code

- Perform calculations in assignment statements
  - Calculation appears on the right side of the assignment sign (=)
  - Result is assigned to a variable or property on the left side of the equal sign (never a constant)

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

Other Assignment Operators

- Combined assignment operators
  - Perform both a calculation and an assignment
    - += Add to the operator on the left
    - -= Subtract from the operator on the left
    - *= Multiply by operator on left and assign result to the operator
    - /= Divide into operator on left and assign result to the operator
    - += (Strings) concatenate to the operator on the left

```csharp
totalSalesDecimal += saleDecimal;
countDownInteger -= 1;
resultInteger *= 2;
sumDecimal /= countInteger;
bigString += smallString;
```
Increment and Decrement Operators

- **Increment (++)** — Adds 1 to a variable
  
  ```csharp
  countInteger++;
  ```

- **Decrement (--)** — Subtracts 1 from the variable
  
  ```csharp
  countDownInteger--;
  ```

- **Prefix notation**
  - Prefix to perform the increment before the arithmetic operation
  
  ```csharp
  Add 1 to countInteger before subtracting from 100
  
  resultInteger = 100 - ++countInteger;
  ```

- **Postfix notation**
  - Postfix to perform the increment after the arithmetic operation
  
  ```csharp
  Subtract countInteger from 100 before incrementing countInteger
  
  resultInteger = 100 - countInteger++;
  ```

Converting between Numeric Data Types

- **Explicit conversions**
  - Also called *casting*
  - Convert between data types without implicit conversions
  - Specify the destination data type in parentheses before the data value to convert
  - Generates an exception if significant digits will be lost
  
  ```csharp
  numberDecimal = (decimal) numberFloat;
  valueInt = (int) valueDouble;
  ```

Converting to Numeric Data Types

- **Convert class methods**
  - C# methods available
  - `ToDecimal`, `ToSingle`, `ToDouble`
  - Integer data types use .NET class names
  - `short` → `ToInt16`
  - `int` → `ToInt32`
  - `long` → `ToInt64`
  - Examples
    
    ```csharp
    numberDecimal = Convert.ToDecimal(numberSingle);
    valueInteger = Convert.ToInt32(valueDouble);
    valueDouble = Convert.ToDouble(textBoxInput.text);
    ```
Rounding Numbers

- Round decimal fractions using `decimal.Round`
  - Returns a decimal result rounded to the specified number of digits
    - `resultDecimal = decimal.Round(amountDecimal, 2);`
  - `decimal.Round` and `Convert` methods use "rounding toward even" if the digit to the right of the final digit is exactly 5.

<table>
<thead>
<tr>
<th>Decimal value to round</th>
<th>Number of decimal positions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.455</td>
<td>2</td>
<td>1.46</td>
</tr>
<tr>
<td>1.445</td>
<td>2</td>
<td>1.44</td>
</tr>
<tr>
<td>1.5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2.5</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Formatting Data for Display

- Display numeric data in a label or text box by converting the value to a string
  - Use `ToString` method
    - `displayTextBox.Text = numberInteger.ToString();`
  - Format the data using formatting codes as an argument of the `ToString` method
    - Specifies dollar sign, percent sign, and commas
    - Specifies the number of digits to the right of the decimal point

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C&quot; or &quot;c&quot;</td>
<td>Currency</td>
<td><code>extendedPriceTextBox.Text = (quantityInteger * priceDecimal).ToString(&quot;C&quot;);</code></td>
</tr>
<tr>
<td>&quot;N&quot; or &quot;n&quot;</td>
<td>Number</td>
<td><code>discountTextBox.Text = discountDecimal.ToString(&quot;N&quot;);</code></td>
</tr>
<tr>
<td></td>
<td>Specify number of decimal positions by placing a numeric digit following the specifier code</td>
<td><code>discountTextBox.Text = discountDecimal.ToString(&quot;N3&quot;);</code></td>
</tr>
</tbody>
</table>
More Format Specifier Codes

- Additional Format Specifier Codes
  - "F" or "f" – Fixed point
    - String of numeric digits, no commas, two decimal places, minus sign at left for negative values
  - "D" or "d" – Digits
    - Force a specified number of digits to display
    - For integer data types only, minus sign at left
  - "P" or "p" – Percent
    - Multiplies the value by 100, adds a space and a percent sign, rounds to two decimal places
  - Additional codes for dates and times
    - d, D, t, f, f, g, m, M, r, R
    - Date and time codes are case sensitive

Format Specifier Code Examples

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Format Specifier Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>totalDecimal</td>
<td>1125.6744</td>
<td>&quot;C&quot;</td>
<td>$1,125.67</td>
</tr>
<tr>
<td>totalDecimal</td>
<td>1125.6744</td>
<td>&quot;N&quot;</td>
<td>1,125.67</td>
</tr>
<tr>
<td>totalDecimal</td>
<td>1125.6744</td>
<td>&quot;N0&quot;</td>
<td>1,126</td>
</tr>
<tr>
<td>balanceDecimal</td>
<td>1125.6744</td>
<td>&quot;%d&quot;</td>
<td>1,125,674</td>
</tr>
<tr>
<td>balanceDecimal</td>
<td>1125.6744</td>
<td>&quot;%f&quot;</td>
<td>1125.674</td>
</tr>
<tr>
<td>pinInteger</td>
<td>123</td>
<td>&quot;D6&quot;</td>
<td>000123</td>
</tr>
<tr>
<td>rateDecimal</td>
<td>0.075</td>
<td>&quot;P&quot;</td>
<td>7.50 %</td>
</tr>
<tr>
<td>rateDecimal</td>
<td>0.075</td>
<td>&quot;P3&quot;</td>
<td>7.500 %</td>
</tr>
<tr>
<td>rateDecimal</td>
<td>0.075</td>
<td>&quot;P0&quot;</td>
<td>8 %</td>
</tr>
<tr>
<td>valueInteger</td>
<td>10</td>
<td>&quot;C&quot;</td>
<td>($10.00)</td>
</tr>
<tr>
<td>valueInteger</td>
<td>10</td>
<td>&quot;N&quot;</td>
<td>-10.00</td>
</tr>
<tr>
<td>valueInteger</td>
<td>10</td>
<td>&quot;D3&quot;</td>
<td>-010</td>
</tr>
</tbody>
</table>

Handling Exceptions

- Exceptions cause run–time errors
  - Blank or non–numeric data cause Parse methods to fail
  - Entering a number that results in an attempt to divide by zero
- Exception Handling catches bad data before a run–time error occurs
  - Writing code to catch exceptions is called exception handling
  - Exception handling in Visual Studio.NET is standardized for all languages that use the CLR
try/catch Blocks

- Enclose any statement(s) that might cause an error in a try/catch block
  - If an exception occurs while the statements in the try block are executing, program control transfers to the catch block
  - The code in a finally block (if included) executes last, whether or not an exception occurred

The try Block – General Form

```csharp
try
{
    // Statements that may cause error.
}
catch (ExceptionType [VariableName])
{
    // Statements for action when exception occurs.
} finally
{
    // Statements that always execute before exit of try block.
}
```

Displaying Messages in Message Boxes

- Display a message in a message box
  - User has entered invalid data
  - User neglects to enter required data
- Specify message, optional icon, title bar text and buttons
- Use Show method of the MessageBox object to display a message box
The MessageBox Object – General Form

- Arguments must match one of the formats
- Do not reverse, transpose or leave out any argument

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

Using Overloaded Methods

- **Overloading** allows the Show method to act differently for different arguments
- Each argument list is called a **signature**
  - The `Show` method has several signatures
  - Supplied arguments must exactly match one of the signatures
  - IntelliSense helps in entering the arguments (no need to look up or memorize the signatures)

Using Class-level Variables

- Declare a class level variable
  - Retains its value for multiple calls
  - Allows accumulation of totals

  ```csharp
  totalAmountDecimal += amountDecimal;
  ```

- Summing numbers

- Counting numbers
  - Add 1 to the counter variable in the calculate event method
  ```csharp
  saleCountInteger ++;
  ```

- Calculating an average
  - Divide the sum of items by the count of items
  ```csharp
  averageSaleDecimal = totalAmountDecimal / saleCountInteger;
  ```