



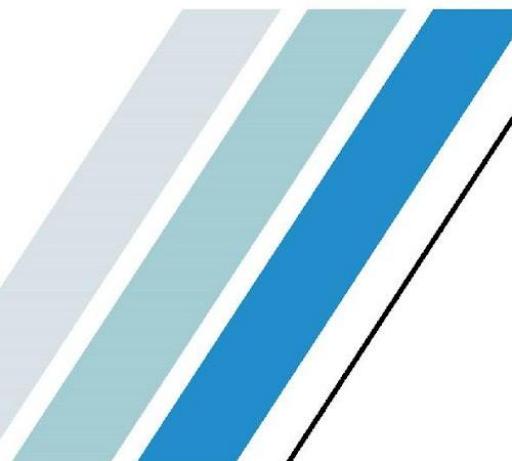
# WEEK 5

# DATA HANDLING

PF101 - OBJECT ORIENTED PROGRAMMING



# LEARNING OUTCOMES:

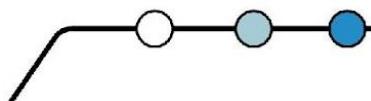
- 
- At the end of the session, the students should be able to:
1. Differentiate Types of Variables
  2. Explain different uses of Variables
  3. Define Constant, Modifiers, Statement and Directives
  4. Enumerate Visual Basic Operators and Data types
- 

## Handling Data

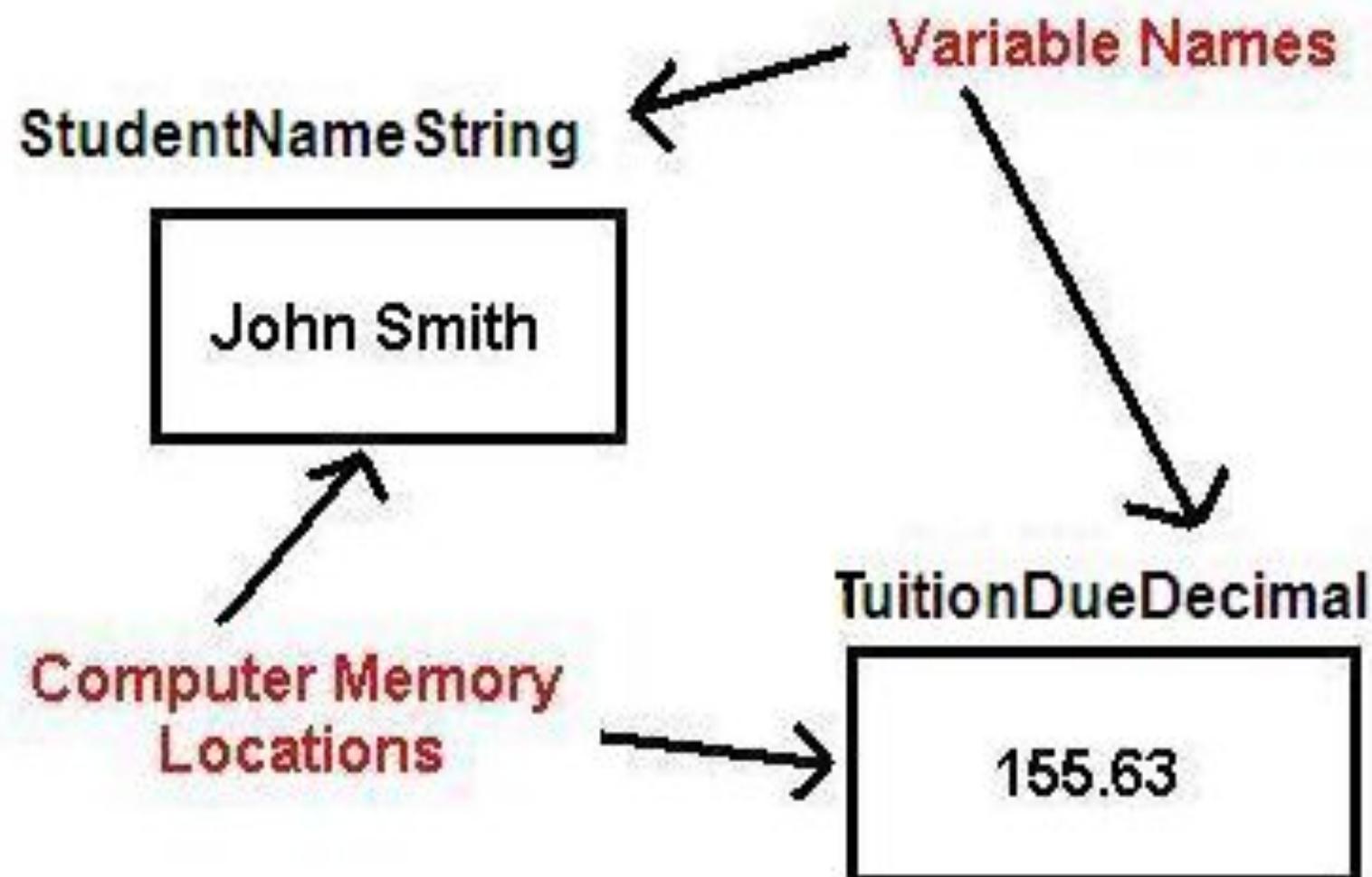
- Variables
  - Provide a means to store data values that are not stored in files.
  - Support the computation of values through their use in formulas in assignment statements.
  - Represent locations in a computer's memory.
  - are assigned a unique name for reference when writing computer code

# WEEK 5 – Data Handling

- 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.



# WEEK 5 – Data Handling



# 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.



# WEEK 5 – Data Handling

| Data Type                                    | Description of data stored   | Memory storage in bytes |
|--|--|-------------------------|
| <b>Text Data Storage</b>                     |  |                         |
| 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                       |
| <b>Numeric Data Storage – Fixed Point</b>    |  |                         |
| Decimal                                      | Decimal numeric values – often used to store dollars/cents.  | 16                      |
| <b>Numeric Data Storage – Floating Point</b> |  |                         |
| Double                                       | Double-precision numeric values with 14 digits of accuracy.  | 8                       |
| Single                                       | Single-precision numeric values with 6 digits of accuracy.   | 4                       |

# WEEK 5 – Data Handling

| 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 |

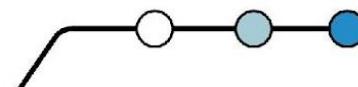
# WEEK 5 – Data Handling

- 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.

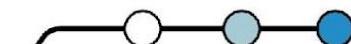


- Names cannot be **VB reserved words** such as **LET, PRINT, DIM, or CONST.**
- Names are **not** case sensitive. This means that **TotalInteger, TOTALINTEGER, and totalinteger** are all **equivalent** names.
- For all intensive purposes, a Name can be as long as you want (the actual limit is 16,383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



## Declaring Variables

- Declare variables that are local with the **Dim** statement. Declare variables that are module-level 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.
- Constants must have an initial value specified.



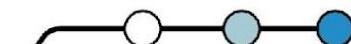
## Examples:

- **Dim StudentNameString AsString**
- **Dim CountStudentsInteger AsInteger**
- **Dim AccountBalanceDecimal AsDecimal = 100D**
- You can also declare more than one variable in a **Dim** statement.
- **Dim StudentNameString, MajorString AsString**
- **Dim SubtotalDecimal, TotalDecimal, TaxAmountDueDecimal AsDecimal**



# 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.

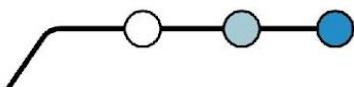
- **Intrinsic Constants** – these are defined as **enumerations** such as **Color.Red** and **Color.Blue**. These are called **intrinsic** because they are predefined in VB and always exist for your use.
- **Named Constants** – these are constants you define with a **Const** statement. These constants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



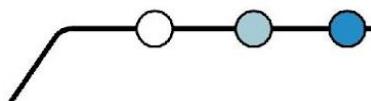
# WEEK 5 – Data Handling

Examples:

- **Const SALES\_TAX\_RATE\_SINGLE AsSingle = 0.0725F**
- **Const BIG\_STATE\_NAME\_STRING AsString = "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.



# WEEK 5 – Data Handling

- 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

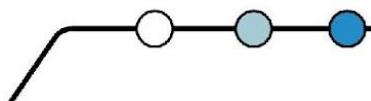
# Scope of Variables and Constants

- 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).



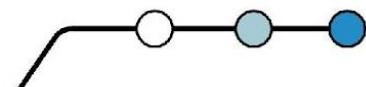
# WEEK 5 – Data Handling

- 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.
- **Modulelevel** (usually use **Private** to declare the variable; use **Const** to declare the 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).

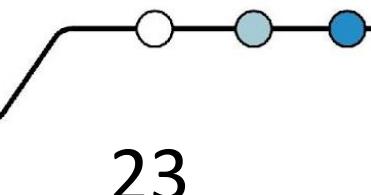


# WEEK 5 – Data Handling

- 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 the variable; use **Const** to declare the 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.

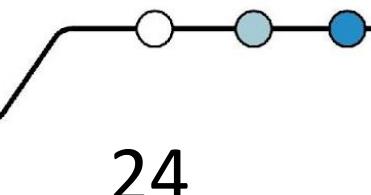


- **Block** (use **Dim** to declare the variable; use **Const** to declare the constant) – the variable/constant is only visible within a small portion of a procedure – these variables/constants are rarely created.



## WEEK 5 – Data Handling

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



# WEEK 5 – Data Handling

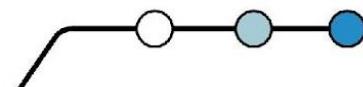
The image shows a screenshot of a Microsoft Visual Studio IDE window titled "Books.vb\* Books.vb [Design]\*". The code editor displays a VB.NET module named "Books". A series of callout boxes with arrows point from specific code snippets to explanatory text:

- An arrow points from the line "Public Class Books" to a box containing: "Module-level variables and constants are declared outside of a Sub procedure - usually just after the Public Class statement".
- An arrow points from the declaration "Private TotalQuantityInteger As Integer" to a box containing: "Use Private instead of Dim to declare module-level variables".
- An arrow points from the declaration "Const SALES\_TAX\_RATE\_SINGLE As Single = 0.0725" to a box containing: "A local constant".
- An arrow points from the declaration "Dim SubtotalDecimal, SalesTaxDecimal, TotalDueDecimal As Decimal" to a box containing: "Several local numeric variables declared but not immediately assigned variables have initial values of zero".
- An arrow points from the assignment "Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text, Globalization.NumberStyles.Currency)" to a box containing: "Two local variables declared and assigned values".
- An arrow points from the assignment "Dim QuantityInteger As Integer = Integer.Parse(QuantityTextBox.Text, Globalization.NumberStyles.Number)" to the same "Two local variables declared and assigned values" box.

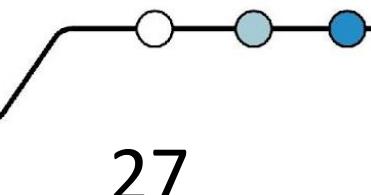
```
1 'Project: Ch03VBUniversity
2 'D. Bock
3 'Today's Date
4
5 Option Strict On
6
7 Public Class Books
8     'Declare module-level variables and constants
9     Private TotalQuantityInteger As Integer
10    Private TotalSalesDecimal As Decimal
11
12 Private Sub ComputeButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click
13     Try
14         'Declare constants
15         '7.25 percent sales tax rate
16         Const SALES_TAX_RATE_SINGLE As Single = 0.0725
17
18         'Declare variables
19         Dim SubtotalDecimal, SalesTaxDecimal, TotalDueDecimal As Decimal
20
21         'Declare variables and convert values from
22         'textbox controls to memory
23         Dim PriceDecimal As Decimal = Decimal.Parse(PriceTextBox.Text,
Globalization.NumberStyles.Currency)
24         Dim QuantityInteger As Integer = Integer.Parse(QuantityTextBox.Text,
Globalization.NumberStyles.Number)
25
26         'Process - Compute values
27         'Subtotal = price times the quantity of books
```

# Converting 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).

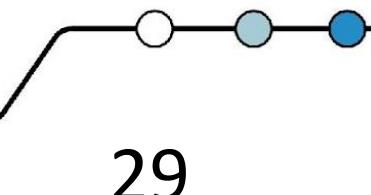


- **SubtotalTextBox.Text = SubtotalDecimal.ToString("N2")**
- **SalesTaxTextBox.Text = SalesTaxDecimal.ToString("C2")**
- **QuantityTextBox.Text = QuantityInteger.ToString("N0")**



- **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.



# WEEK 5 – Data Handling

Table 3.3

|         |   |
|---------|---|
| Decimal | <code>NumberDecimal = Convert.ToDecimal(ValueSingle)</code> |
| Single  | <code>NumberSingle = Convert.ToSingle(ValueDecimal)</code>  |
| Double  | <code>NumberDouble = Convert.ToDouble(ValueDecimal)</code>  |
| Short   | <code>NumberShort = Convert.ToInt16(ValueSingle)</code>     |
| Integer | <code>NumberInteger = Convert.ToInt32(ValueSingle)</code>   |
| Long    | <code>NumberLong = Convert.ToInt64(ValueDouble)</code>      |

- Use the **Parse** method to convert a string to a number Performing Calculations – VB uses a wider data type when calculations include unlike data types or to parse the value in a textbox control.  
This example produces a decimal result.
- **AverageSaleDecimal = TotalSalesDecimal / CountInteger**
- **Summary Rules:**
  - 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



# 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:

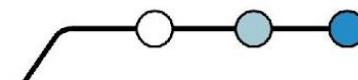
**ItemValueDecimal = QuantityInteger \* PriceDecimal**

**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**

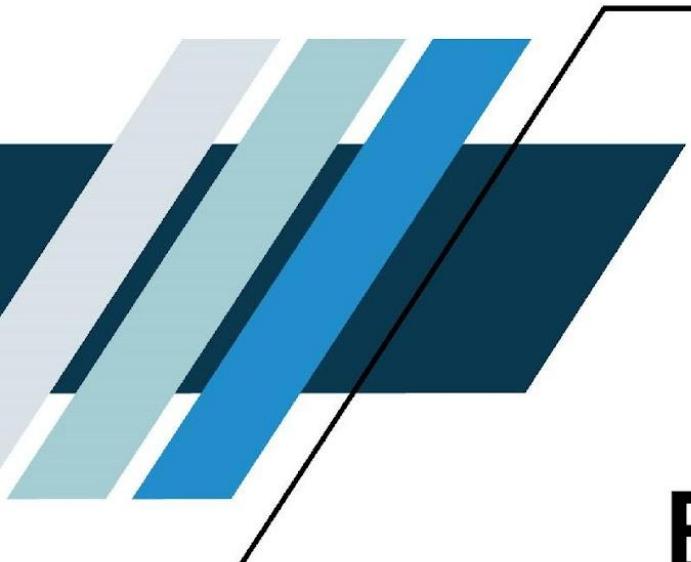


- The minus symbol combined with the equal sign allows you to decrement or count backwards.  
Examples:

**CountInteger -= 1**

- is equivalent to

**CountInteger = CountInteger - 1**



**END OF PRESENTATION.  
THANK YOU!**

