CS608 Lecture Notes

Visual Basic.NET Programming

Introduction to visual Basic.NET

Review of Language Components (Cont)

(Part III of IV)

(Lecture Notes 1C)

Prof. Abel Angel Rodriguez
CHAPTER 4  PROGRAMMING FUNDAMENTALS/.NET NAMESPACE

4.1 Using the Classes Supplied with .NET (The NameSpace)...............................................................................................................3
4.1.1 The NameSpace ........................................................................................................................................................................3
.NET Class Library & Namespace .....................................................................................................................................................3
4.2 String Class – System Namespace..................................................................................................................................................5
4.1.2 Basic String Variable Declaration & Reference Variables Revisited............................................................................................5
String Declaration ..................................................................................................................................................................................5
Reference Variables Revisited ..............................................................................................................................................................6
String Constant .......................................................................................................................................................................................7
Empty String ..........................................................................................................................................................................................7
Concatenating Strings ............................................................................................................................................................................8
ANSI (ASCII) Character Set ...............................................................................................................................................................8
4.1.3 The String Class Properties and Methods ....................................................................................................................................9
String Object Properties ........................................................................................................................................................................9
String Object Methods .......................................................................................................................................................................10
4.3 Working with Dates ..............................................................................................................................................................................15
4.3.1 Working with the Basic Date Data Type ...................................................................................................................................15
4.3.2 The DateTime Structure ...............................................................................................................................................................16
Properties of the DateTime Class ..........................................................................................................................................................16
Methods of the DateTime Class .............................................................................................................................................................17
Typical Date & Time Operations and usage .....................................................................................................................................17
Working with the DateTime Class .........................................................................................................................................................18
4.4 Methods ...............................................................................................................................................................................................21
4.4.1 Method Basics .................................................................................................................................................................................21
4.4.2 Methods & Event-Procedure Methods ......................................................................................................................................21
4.4.3 Passing Values onto Methods .......................................................................................................................................................21
4.4.4 Methods Syntax ..............................................................................................................................................................................22
4.4.5 Sub Procedure Declaration & Call ........................................................................................................................................22
Sub-Procedure Declaration: .................................................................................................................................................................22
Regular Sub-Procedure Method Call: ................................................................................................................................................23
Sub-Procedure With Arguments Declaration: ......................................................................................................................................24
Sub-Procedure With Arguments Method Call: ..................................................................................................................................25
4.4.6 Function Procedure Declaration & Call ..................................................................................................................................26
Function Procedure Declaration: ............................................................................................................................................................26
Regular Function Procedure Call: .......................................................................................................................................................27
Function Procedure with Arguments Declaration: ..................................................................................................................................28
Function Procedure With Arguments Call: .........................................................................................................................................29
4.4.7 Passing Arguments to Methods by Value (ByVal) or by Reference (ByRef) .............................................................................30
Pass-by-Value (ByVal) ..........................................................................................................................................................................30
Pass-by-Reference (ByRef) .................................................................................................................................................................33

HOMEWORK ..............................................................................................................................................................................................................37
In previous lecture we discussed the .NET Framework and it’s importance to programming using VB.NET.

The .NET Framework is composed of two main components the .NET CLR or Common Language Runtime and the .NET Class Library as shown in the figure below:

- The CLR compiles to one common machine language, while the .NET Class Library contains classes that can be used by all the languages the support the .NET Framework (VB.NET, C#, C++, etc.)

**.NET Class Library & Namespace**

- Our focus now is on the the .NET Class Library
- The the .NET Class Library contains hundreds of predefined classes for us to used to create our programs.
- These Classes are organized into groups of related Classes called Namespace
- The .NET Class Library namespace is organized into groups of related classes for User Interface, data access etc.
- In addition you can also define you own namespace and add your custom classes to the namespace.
- The table below lists some of the namespace available in the Class Library:

**Import Statement:**

- In order to use any of the Namespace Classes, you need to use the keyword Imports followed by the namespace at the beginning of the program. Syntax is as follows:

```csharp
Imports Namespace
```
Example:

- Importing the Collection Class:
  
  \[\text{Imports System.Collections} \]

- Note that the classes that belong to the **System Namespace** is automatically imported by the compiler and you don’t need to use the keyword Imports.
- The following table lists some of the available namespace:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Namespace Class Groups</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Array</td>
<td>- Array data type covered so far is actually a library class</td>
</tr>
<tr>
<td></td>
<td>Console</td>
<td>- Console application are managed by this class</td>
</tr>
<tr>
<td></td>
<td>Convert</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DateTime</td>
<td>- Date and Time functions are available via this library</td>
</tr>
<tr>
<td></td>
<td>Exception</td>
<td>Structure</td>
</tr>
<tr>
<td></td>
<td>TimeSpan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>String</td>
<td>- String variables and functionality via this library class</td>
</tr>
<tr>
<td></td>
<td>Math</td>
<td>- Math processes are provided by this library class</td>
</tr>
<tr>
<td>System.Collections</td>
<td>ArrayList</td>
<td>Powerful class to store data an objects on a list similar to an array but more powerful and flexible.</td>
</tr>
<tr>
<td></td>
<td>SortedList</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DictionaryBase</td>
<td></td>
</tr>
<tr>
<td>System.IO</td>
<td>StreamReader</td>
<td>Reading/writing to files</td>
</tr>
<tr>
<td></td>
<td>StreamWriter</td>
<td></td>
</tr>
<tr>
<td>System.Data</td>
<td>DataRow</td>
<td>Database Powerful &amp; Flexible Data Storage Classes &amp; Mechanism</td>
</tr>
<tr>
<td></td>
<td>DataTable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DataSet</td>
<td></td>
</tr>
<tr>
<td>System.Data.OleDb</td>
<td>OleDbCommand</td>
<td>OLE DB Powerful Data Access classes</td>
</tr>
<tr>
<td></td>
<td>OleDbConnection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OleDbDataAdapter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OleDbParameter</td>
<td></td>
</tr>
<tr>
<td>System.Windows.Forms</td>
<td>Button</td>
<td>Windows Application GUI Classes</td>
</tr>
<tr>
<td></td>
<td>CheckBox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Form</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Label</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Menu</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MenuItem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RadioButton</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TextBox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MessageBox</td>
<td></td>
</tr>
<tr>
<td>System.Web</td>
<td>System.data</td>
<td>Web Development Classes</td>
</tr>
<tr>
<td></td>
<td>System.data.oledb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.Web</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.Web.UI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.Web.UI.WebControls</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.Web.Services</td>
<td></td>
</tr>
</tbody>
</table>
4.2 String Class – System Namespace

4.1.1 The String Data Type

- A String is a variable that is declared from the String Class, therefore string is really an object of the String Class!
- If is an object from a Class, then it has Properties & Methods that we can use!
- In VB.NET the String Class is a Library Class from the System Namespace.
- Strings are the most common data that is processed in program. Think about it, aside from number calculations, most processing involves the user entering strings of characters or words and processing such strings. For examples sentences, phrases, words, letters, names, telephone numbers, addresses, etc.
- Because Strings are the data types that undergo more processing than any other data type, the String Class contains many properties and methods that allow you to manipulate strings. We will briefly analyze some of those features.

4.1.2 Basic String Variable Declaration & Reference Variables Revisited

String Declaration

- When you declare a variable of type String, you are declaring a String Variable. A variable of type string can be manipulated just like any other variable, except that the content is a string of characters.
- Declaration Syntax:

```
Dim strVariableName As String
```

Example 1:

- Declaring a last name string:
  ```
  Dim strLastName As String
  ```

Reference Variables Revisited

- If you recall in previous lecture we talked about reference variables.
- A string variable is actually a Reference Variable.
- Reference variables are variables declared from a Class.

```
Access Name As ClassName
```

- What this means is that a string variable that you declared and define is a Reference variables and in physical memory it contain a Pointer to an instance or Object of the String Class.
- Therefore String variable that is populated with data, is really a reference or pointer to an objects of the String Class.

- This can be tricky, and needs to be clarified. In reality when you declare a string variable, there is NO object created, only the pointer. Objects are not created until you assign a value to the variable. So the true representation in memory of the declaration of a String Variable is as follows:

```
String Variable
```

---

5
For example, in the following declaration:

```
Dim strLastName As String
```

- Is represented in memory as follows: `strLastName`  

Note that the actual variable `strLastName` contains a Pointer or a memory address.
- Only when you assign a value to the string variable will the object be created and we have the following:

```
String Variable ➔ String Object
```

**String Assignment/Definition**
- In the previous section we declared a string variable and thus created a reference pointer.
- Also if you recall our basic lecture on variables, we stated that using Variables is a three step process:
  1. **Declaration**: Introduces the variable name into a program
  2. **Definition**: Memory is actually created or reserved for the variable
  3. **Initialize**: Populate a variable with data

- When we declare a string variable we are only introducing the variable. When we define or assign a value we are actually creating the object.
- Therefore when we assign a value to a string we are creating the object and telling the String variable to point to the object.

**Example 2:**
- Declaring a last name string:
  
  ```
  Dim strLastName As String
  
  strLastName
  ```

- Defining or initializing the string variable:
  
  ```
  strLastName = “Smith”
  ```

- String Variable ➔ String Object

```
```

```
```
You can also declare and assign a string in one step as follows:

```dim strVariableName as string = "StringValue"
```

### Example 3:
- Declaring a last name string:

```
Dim strLastName As String = "Smith"
```

### String Constant
- When you declare a String that you don’t want the value to change throughout the program. You are declaring a String Constant.
- Syntax:

```const strVariableName as string = "StringValue"
```

### Example 4:
- Declaring a Constant String:

```const strCompanyName as string = "IBM Corporation"
```

- This string cannot be changed throughout the program

### Empty String
- A string that is empty or contains no text is an empty string. Empty strings are very popular when you want to clear the value of a string variable, text box etc.
- An empty String is simply assigning double quotes to the string
- Syntax:

```strVariableName = ""
```
Concatenating Strings

- You can concatenate or join two strings together using the & Operator.
- When you combine two strings using the & operator you are creating a new string.
- Syntax:

  strVariableName1 & strVariableName2

Examples:

strFirstName = “Joe”

strLastName = “Smith”

- Concatenating two strings:
  strFullName = strFirstName & strLastName
  ‘strFullName contains: “JoeSmith”

- Concatenating three strings:
  strFullName = strFirstName & “ “ & strLastName
  ‘strFullName contains: “Joe Smith”; note the space string between the names

- Concatenating three strings where one string is text character enclosed in quotes:
  strFullName = strLastName & “ “ & strFirstName
  ‘strFullName contains: “Smith, Joe”

- Concatenating string enclosed in quotes & variables string:
  strFullName = “Joe” & strLastName
  ‘strFullName contains: “JoeSmith”

  strFullName = strFirstName & “ “ & “Smith”
  ‘strFullName contains: “Joe Smith”; note the space string between the names

ANSI (ASCII) Character Set

- The ANSI Character Set is a standard used throughout the industry to represent the characters set used in computers.
- ANSI stands for American National Standard Institute.
- The ANSI Character Set was also known as the ASCII Set.
- Every character has a decimal ANSI equivalent. There are a total of 95 characters on a standard keyboard.
- For every keyboard character, there is an ANSI decimal equivalent.
- In the book’s Appendix you will find a table of ANSI character set.
- Here is an example of a few characters and their ANSI code:

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space (Blank)</td>
<td>32</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
</tr>
</tbody>
</table>

- Why is this character set important? When you manipulate or use characters, you are actually manipulating the decimal ANSI numbers that represent a character. But when you see it on the screen you see the character not the ANSI decimal value. This value is hidden and used internally to the computer.
- In your programs, you can use the ANSI characters set decimal values, but you need to use the special function Chr() using the following syntax:

  Chr(ANSI Value)

  Example:
  txtTextBox.Text = chr(65)  ‘From the table, the decimal 65 represents the letter “A”, therefore the textbox contains the text = “A”
4.1.3 The String Class Properties and Methods

- The String data type is powerful because it’s actually an Object of a class.
- Since a String is an Object as all objects it has:
  - Properties – we can get so it can return information for us
  - Methods – we can call to do something for us

### String Object Properties

- The String Class contains several properties which we can utilize. The table below list these properties, assuming that $S$ is the string object or variable:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$.Length</td>
<td>(READ-ONLY)Get property which returns an integer value containing the number of character in as string</td>
</tr>
<tr>
<td>$S$.Chars(i)</td>
<td>(READ-ONLY)Get property which returns the character located in the index location (i). Note that (i) is zero-based or starting at 0</td>
</tr>
</tbody>
</table>

### Length Property

- The Length Property returns an integer value containing the number of character in as string.
- The syntax to getting any property is $Value = Object.Property$
- As with any property the syntax to string Length Property:

```
'Strong.Length

Where:
String: Object
Length: Property
```

### Example 6:

- Getting the number of characters in a string:
  - Example 6 – ‘Getting the number of characters in a string’:
    ```vba
    Dim strName As String
    Dim intLen As Integer
    strName = “Joe Smith”
    intLen = strName.Length  'The variable intLen contains the integer 9 for number of characters including the space
    ```
String Object Methods

A string contains several methods that you can use to manipulate the string.

The following is an example of several of these string manipulation methods. Assuming that S is the string object or variable (Note that highlighted rows indicate commonly used methods and those important to this course and projects):

- **S.Copy(S)**: Returns a NEW string instance or Object, which is a copy of S (Explained below)
- **S.StartsWith("string")**: Returns Boolean True if the String S begins with the value "string"; Otherwise returns False
- **S.EndsWith("string")**: Returns Boolean True if the String S ends with the value "string"; Otherwise returns False
- **S.Equals(S2)**: Returns Boolean True if the String S2 is equal to String S; Otherwise returns False
- **S.IndexOf("string")**: Returns the index or location of the first occurrence of the character "string". Returns -1 if not found.
- **S.Insert(i, "string")**: Inserts the string value "string" into the beginning of the index i (relative to zero) and returns the new string instance.
- **S.Replace("char1", "Char2")**: Replaces "char1" with "Char2" and returns the new string instance.
- **S.SubString(StartIndex, Length)**: Returns the substring or new string containing the string of characters beginning at the starting index or StartIndex and ending with the number of characters in the substring Length.
- **S.ToUpper**: Returns the Upper Case or Capitalized version of the string
- **SToLower**: Returns the Upper Case or Capitalized version of the string
- **S.Trim**: Removes the space at the beginning (front) or end (back) of the string. Removes leading/trailing spaces. Used for cleaning up a string and removing any additional leading and trailing spaces
- **S.Split("Delimiter_String")**: Return an Array containing the separated sub strings between a Delimited string. In other words this method allows you to parse a delimited line. The delimiter is a string, for example a comma (,), space, or any delimiter within a string. For example CSV files or Comma-Delimited files use commas to separate substrings. This method returns all the substrings within the comma-delimited line and places each sub string in a cell in the array in the order in which they are found in the line. (See example below)
ToUpper, ToLower & Trim Methods

These string methods are used as follows.

Note that in some examples I may represent a string variable and the object it points to as a single box for simplicity. Remember the in reality the string variable contains a reference pointer and the value is actually the object.

Example 8:

Examples of ToUpper Method to set string to Upper Case:

- Example 8 – ‘Converting to Upper Case Letters:
  ```vbnet
dim strName, strString2 as string
strName = “Joe Smith”
strString2 = strNameToUpper
```

Example 9:

Examples of ToLower Method to convert string to lower Case:

- Example 9 – ‘Converting to Lower Case Letters:
  ```vbnet
dim strName, strString2 as string
strName = “Joe Smith”
strString2 = strNameToLower
```

Example 10:

Examples of Trim Method to remove excess spaces around a string:

- Example 10 – ‘Removing leading and trailing spaces from string:
  ```vbnet
dim strName, strString2 as string
strName = “ Joe Smith ” “note the leading and trailing additional spaces
strString2 = strName.Trim “spaces are removed
```
Copy Method

- This method may not be needed in your projects but it deserves further explanation because of how it manipulates the reference to string object.
- Before we examine this method we need to review the basic concepts of reference variables (pointers) and strings

String Reference Revisited!

- OK, the following is an example illustrates how references variables work.
- Lets look at the following example:

Example 11:

- Example 11– ‘Example of assigning string:
  
  ```vbnet
  Dim strS1, strS2 As String
  
  ‘Step1 – String value assigned to strS1
  strS1 = "Hello World"
  
  ‘Step2 – Assigns content of strS1 to strS2
  strS2 = strS1
  ```

- Note that assigning two reference string variables to each other means that both pointer are POINTING to the same object. **A new object is not created!!!!!! They are both referencing the same object!!**

- Now lets see an example of the Method `String.Copy(S)` and see how this is used to create a NEW instance of an Object.
- Lets look at this example:

Example 12:

- Example 12– ‘Assigning and creating new Object of string:

  ```vbnet
  Dim strS1, strS2 As String
  
  ‘Step1 – String value assigned to strS1
  strS1 = "Hello World"
  
  ‘Step2 – Assigns content of strS1 to strS2
  strS2 = String.Copy(strS1)
  ```

- Note that this time we have **two separate pointers pointing to two separate Objects!!**
Split Method (Important Method for Your Class Project Exams)

Parsing
- The Split() method of the string class allows you to parse a string.
- Parsing is a very common task in computer programming
- Definition of Parsing:
  - Parsing - Means breaking up a line or sentence down into components or parts. The break up is based on some delimiter or string/
  - For example, let's look at the following Comma-Delimited or Comma-Separated-Value (CSV) string:

```
Joe Smith, 111-11-1111, #1/23/1971#, 333 Jay Street, 718 260-5000
```

- If we were to parse this line, we need to break down or extract each string components of the CSV line into the sub strings WITHOUT the commas:

```
Joe Smith
111-11-1111
#1/23/1971#
333 Jay Street
718 260-5000
```

- In other words we are REMOVING the COMMAS or DELIMITER an extracting the string components. What is done with the sub string components is up to you or the program thereafter.

Using the Split() Method
- This is a very useful method. And one you will need for your projects and you will see most likely throughout your career in the field of computer science. Even if you are not a programmer, if you need to create some scripts you will need the Split() method.
- The String.Split("Delimiter") method, splits or separates a delimited string into its individual components and removes the delimiter. The method returns an array, where each component is placed into the cell of the array as shown in the illustration below:

<table>
<thead>
<tr>
<th>sMyArray</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe Smith</td>
<td>111-11-1111</td>
</tr>
<tr>
<td></td>
<td>#1/23/1971#</td>
</tr>
<tr>
<td></td>
<td>333 Jay Street</td>
</tr>
<tr>
<td></td>
<td>718 260-5000</td>
</tr>
</tbody>
</table>

- After the string components are broken down and placed in the individual cells of the array, it is your job to extract the each components using standard array syntax: sMyArray(Index)
- This method has several way of using or syntax, lets look at the basic one:

\[
sMyArray = sMyString.Split("delimiter")
\]

- Here we assume that the array MyArray has already been declared. Example:

```vbnet
Dim sMyString As String
Dim sMyArray() As String 'Empty array, no size

'Call String.Split("Delimiter") method to parse the string and assign to array
sMyArray = sMyString.Split("","")
```

- Another popular syntax is to declare and use the array at the same time:

```vbnet
Dim sMyArrays() As String = sMyString.Split("delimiter")
```
An example using this syntax:

```vba
Dim sMyString As String

'Call String.Split("Delimiter") method to parse the string and assign to array
Dim sMyArrays() As String = sMyString.Split(";")
```

Let's look at a complete example using a console application:

```
Example 13:
Example 13 – ‘Getting the number of characters in a string:

Sub Main()
    Dim sMyString As String
    Dim sName, sSSNumber, sAddress, sPhone As String
    Dim dBirthDate As Date
    sMyString = "Joe Smith, 111-11-1111, #1/23/1971#, 333 Jay Street, 718 260-5000"
    Dim sMyArray() As String = sMyString.Split(",")
    sName = sMyArray(0)
    sSSNumber = sMyArray(1)
    dBirthDate = CDate(sMyArray(2))
    sAddress = sMyArray(3)
    sPhone = sMyArray(4)
    Console.WriteLine("CSV String: " & sMyString)
    Console.WriteLine("After Parsing:")
    Console.WriteLine("Name: " & sName)
    Console.WriteLine("The social security is " & sSSNumber)
    Console.WriteLine("Birth date: " & dBirthDate)
    Console.WriteLine("The Address: " & sAddress)
    Console.WriteLine("Phone Number: " & sPhone)
End Sub
```

Output:

```
CSV String: Joe Smith, 111-11-1111, #1/23/1971#, 333 Jay Street, 718 260-5000
After Parsing:
Name: Joe Smith
The social security is 111-11-1111
Birth date: 1/23/1971
The Address: 333 Jay Street
Phone Number: 718 260-5000
```
4.3 Working with Dates

4.3.1 Working with the Basic Date Data Type

- In your basic VB.Net courses you learned about the `Date` data type.
- With this data type you were able to create date variables as follows:

```
Dim datMyDate As Date
```

- Date variables can hold values using many formats that represent a date, for example:
  - String such as Month day, Year:  "December 12, 1965"
  - Using a literal or direct value with forward slash (/):  12/12/1965
  - Using dash (-) dashes will be converted to (/):  12-12-1965

- We have also learned that when assigning a real value to a date we need to use the # symbol to enclose the date:

```
' Setting a date
datMyDate = #12/12/1965#
```

- In addition, when we used dates in the past we made use of public date functions which were available to us for working with dates, for example:

**Now()**

```
Now() – This function returns the current date & time. Example:

' Date variable declaration
Dim datMyDate As Date

' Getting Current date & time
datMyDate = Now()

'Displaying results
Console.WriteLine("The current date & time is: " & datMyDate)

Results:
The current date & time is: 2/20/2006 12:50:08 PM
```

**Today()**

```
Today() – This function returns the current date only. Example:

' Date variable declaration
Dim datMyDate As Date

' Getting Current date & time
datMyDate = Today()

'Displaying results
Console.WriteLine("The current date is: " & datMyDate)

Results:
The current date is: 2/20/2006
```
TimeOfDay()

- **TimeOfDay()** – This function returns the **CURRENT TIME** only. Example:

```vbscript
'Date variable declaration
Dim datMyTime As Date

'Getting Current time
datMyTime = TimeOfDay()

'Displaying results
Console.WriteLine("The current time is: " & datMyTime)
```

Results:

The current time is: 1:25:13 PM

- The key point here is that we declared Date variables used them, called date functions etc.
- **But did it occurred to you where are these date functions (Now(), Today(), TimeOfDay() etc., come from?**

### 4.3.2 The DateTime Structure

- It turns out that when ever you declare or use the **Date** data type you are actually using a library Structure from the **System Namespace** named **DateTime**.
- As you recall from the notes earlier in this chapter, the System Namespace contains a structure/class named **System.DateTime**.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Array</th>
<th>Console</th>
<th>Convert</th>
<th>DateTime</th>
<th>Exception</th>
<th>TimeSpan</th>
<th>String</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- It is within this structure which all the functions for working with dates are contained.
- So make the point, when you use a **Date** variable you are actually using a **DateTime** variable.

### DateTime Structure Explained

- A structure is similar to a Class. A structure contains **Properties & Methods**.
- For the sake of this explanation I will call the **DateTime** Structure a class, but keep in mind that they are slightly different but are used the same way.
- The **DateTime** class contains a list of properties and methods, and like any Class type structure, we need to follow the rules:

1. Create Class/Structure
2. Create objects/variables of the Class/Structure
3. Use object of the Class/Structure

- So in order to use the **DateTime**, we create variables and use the dot (.) operator to call the **properties** and **methods** of the **DateTime** class.
- The functions we were using up to this point such as **Now()**, **Today()** and **TimeOfDay()** are all **Property** members of the **DateTime** class.
- With that said, we could write the examples from the previous sections as follows:
Properties of the DateTime Class

- Date: Gets the date component of this instance.
- Day: Gets the day of the month represented by this instance.
- DayOfWeek: Returns an integer representing the day of the week represented by an instance. This property value ranges from zero, indicating Sunday, to six, indicating Saturday.
- DayOfYear: Gets the day of the year represented by this instance.
- Minute: Gets the minute component of the date represented by this instance.
- Month: Gets the month component of the date represented by this instance between 1 and 12.
- Now: Gets a DateTime that is the current local date and time on this computer.
- Second: Gets the seconds component of the date represented by this instance.
- TimeOfDay: Gets the time of day for this instance.
- Today: Gets the current date.
- Year: Gets the year component of the date represented by this instance. The year, between 1 and 9999.

Methods of the DateTime Class

- Add: Adds the value of the specified TimeSpan to the value of this instance.
- AddDays: Adds the specified number of days to the value of this instance.
- AddHours: Adds the specified number of hours to the value of this instance.
- AddMinutes: Adds the specified number of minutes to the value of this instance.
- AddMonths: Adds the specified number of months to the value of this instance.
- AddSeconds: Adds the specified number of seconds to the value of this instance.
- AddYears: Adds the specified number of years to the value of this instance.
- Compare: Compares two instances of DateTime and returns an indication of their relative values. Syntax: DateTime.Compare(t1, t2). Table of results:

<table>
<thead>
<tr>
<th>Value Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than zero</td>
<td>t1 is less than t2.</td>
</tr>
<tr>
<td>Zero</td>
<td>t1 equals t2.</td>
</tr>
</tbody>
</table>

- Subtract: Subtracts the specified time or duration from this instance.
- ToLongDateString: Converts the value of this instance to its equivalent long date string representation.
- ToLongTimeString: Converts the value of this instance to its equivalent long time string representation.
- ToShortDateString: Converts the value of this instance to its equivalent short date string representation.
- ToShortTimeString: Converts the value of this instance to its equivalent short time string representation.

Typical Date & Time Operations and usage

- Get the current date or time: Now, Today, TimeOfDay
- Perform date calculations: DateAdd, DateDiff, DatePart
- Return a date: DateSerial, DateValue, MonthName, WeekDayName
- Return a time: TimeSerial, TimeValue
- Set the date or time: DateTime, TimeOfDay, TimeString, Today
- Time a process: Timer
Working with the DateTime Class

- To use this class you need to create a reference variable to this class.
- The preferred syntax for creating reference variable for class objects involves a two step method. In this method, we first declare the class reference variable and in another step we create the object using the keyword `New`.
- Syntax for creating Date reference variables is the same as creating objects. You can use the one step or two step method for creating a reference variable and creating an instance of a date object. Below is the two step syntax:

```
Dim| Public | Private DateVariableName As ClassName

DateVariableName = New ClassName()
```

Example 1:

- Example of creating reference date variables and using the `NOW` property:

  ```vbnet
  'Create reference variable of DateTime class
  Dim datTodayDate As DateTime
  datTodayDate = New DateTime()

  'Call to Property that displays the current system date and time
  Console.WriteLine(datTodayDate.Now)
  ```

Example 2:

- Example of creating reference date variables and using the `Today` property:

  ```vbnet
  'Create reference variable of DateTime class
  Dim datTodayDate As DateTime
  datTodayDate = New DateTime()

  'Call to Property that displays the current system date ONLY, time is 00:00
  Console.WriteLine(datTodayDate.Today)
  ```
Example 3:
Example of using the **DayOfWeek, Now, Month, Day, Year, Hour, Minute & Second** Properties. In this example we use the one step method for declaring and creating a date variables:

```vbnet
Sub Main()
    'Create Instances of DateTime class using One-Step Method
    Dim datTodayDate As DateTime = New DateTime()
    Dim datDate As DateTime = New DateTime()

    'Display current date and time using NOW Property
    Console.WriteLine("The current date and time is " & datTodayDate.Now)

    'Get Current Date & Time and assign to another date variable
    datDate = datTodayDate.Now()

    'Display month of year using Month Property
    Console.WriteLine("The Current month of the year is " & datDate.Month)

    'Display day of month using Day Property
    Console.WriteLine("The Current day of the month is " & datDate.Day)

    'Display the year using Year Property
    Console.WriteLine("The Current year is " & datDate.Year)

    'Display the Hour using Hour Property
    Console.WriteLine("The Current Hour is " & datDate.Hour)

    'Display the minute using the Minute property
    Console.WriteLine("The Current Minute is " & datDate.Minute)

    'Display the Second using the Second Property
    Console.WriteLine("The Current Second is " & datDate.Second)

    'Display day of week using DayOfWeek Property. Note that property returns a number and we use case statement to compare and display string equivalent
    Select Case datDate.DayOfWeek
        Case 0
            Console.WriteLine("The day of Week is Sunday")
        Case 1
            Console.WriteLine("The day of Week is Monday")
        Case 2
            Console.WriteLine("The day of Week is Tuesday")
        Case 3
            Console.WriteLine("The day of Week is Wednesday")
        Case 4
            Console.WriteLine("The day of Week is Thursday")
        Case 5
            Console.WriteLine("The day of Week is Friday")
        Case 6
            Console.WriteLine("The day of Week is Saturday")
        Case Else
            Console.WriteLine("The day is incorrect")
    End Select
End Sub
```
Example 4:

Example of using the AddDays, AddMonths, AddYears, ToShortDateString, ToLongDateString & Compare Methods. In this example we use the one step method for declaring and creating a date variables:

Sub Main()
    'Create Instances of DateTime class
    Dim datCurrentDate As DateTime = New DateTime()
    Dim datToday As DateTime = New DateTime()
    Dim datFutureDate As DateTime = New DateTime()
    Dim datFirstDate As DateTime = New DateTime()
    Dim datSecondDate As DateTime = New DateTime()
    'Display current date and time using Now Property
    Console.WriteLine("The current date and time is " & datCurrentDate.Now)
    'Get Current Date & Time and assign to another date variable
    datToday = datCurrentDate.Now()
    'Add 2 days to the current date using AddDays Method
    datFutureDate = datToday.AddDays(2)
    'Display what date is 2 days later
    Console.WriteLine("Todays date is " & datToday.ToShortDateString() & _
        " Two days from today is " & datFutureDate.ToLongDateString())
    'Add 2 Months to the current date using AddMonths Method
    datFutureDate = datToday.AddMonths(2)
    'Display what date is 2 months later
    Console.WriteLine("Todays date is " & datToday.ToShortDateString() & _
        " Two months from today is " & datFutureDate.ToLongDateString())
    'Add 2 Years to the current date using AddYears Method
    datFutureDate = datToday.AddYears(2)
    'Display what date is 2 years later
    Console.WriteLine("Todays date is " & datToday.ToShortDateString() & _
        " Two Years from today is " & datFutureDate.ToLongDateString())
    'Populate Date variables with dates
    datFirstDate = #3/25/2006#
    datSecondDate = #4/12/2004#
    Console.WriteLine("Compare first date " & datFirstDate & " to second date " _
        & datSecondDate)
    'Compare first date with second date
    If DateTime.Compare(datFirstDate, datSecondDate) < 0 Then
        Console.WriteLine("First Date is less than Second Date")
    ElseIf DateTime.Compare(datFirstDate, datSecondDate) > 0 Then
        Console.WriteLine("First Date is greater than second Date")
    ElseIf DateTime.Compare(datFirstDate, datSecondDate) = 0 Then
        Console.WriteLine("First Date is equal to second Date")
    End If
End Sub
4.4 Methods

- When we introduced Objects, we stated that a regular Method is an action taken by an Object.
- Now we are going to learn how to implement Methods.
- Methods are program codes that are grouped together into a block or package for execution. This execution is the action that a Method takes when called.
- Methods are called by program instructions. When called the block of code executes.
- Methods can be placed inside forms, modules & class modules and can be executed as needed.

4.4.1 Method Basics

- There are two types of Methods:
  1) Sub Procedure: - Performs an action (Executes a block of code)
  2) Function Procedure: - Performs an action (Executes a block of code) and returns a value known as the returned value
    - Note that both the Function from the Sub Procedures perform actions, key difference is that the Function returns a value.
    - A function is like a black box, that performs some action and returns a value

- There are two steps required to using Methods:
  1) Method Declaration or Definition: - Writing the block of code or Instructions that defines the action
  2) Method Call: - Invoking the execution of the block of code

4.4.2 Methods & Event-Procedure Methods

- There is actually a third type of Method:
  - Event-Procedure Method: - A specialized Method that automatically Performs an action (Executes a block of code) executes when an action is imposed on the Object by the User
    - Note the key point here is automatic execution of the code, there is NO Method Call. You don't have to explicitly tell the program to execute the block of code; it does it automatically based on the specific event triggered by the user.

4.4.3 Passing Values onto Methods

- You can pass values of data onto a method and have the method use this data
- In order to do this, the Method declaration must have a list of the variables or parameters that will hold or store the data that will be passed.
- On the other hand, when the Method is called, you must pass the data to the variable parameter list, the data you pass is known as the argument.
- The definitions are:
  - Parameter List: - A list of variable in the declaration that specify the name and type of the variable that will temporarily store the data to be passed.
  - Argument List: - A list of variable or values that are passed when calling or executing the method.

- Don’t get these two confused, the Parameters are variables that are defined in the Method declaration, the Argument are variables or values defined during the Method Call.
4.4.4 Methods Syntax

As stated before the two steps required to creating a method are:

1) **Method Declaration or Definition:** - Writing the block of code or Instructions that defines the action
2) **Method Call:** - Invoking the execution of the block of code

4.4.5 Sub Procedure Declaration & Call

**Sub-Procedure Declaration:**

- The *Sub Procedure* Method simply executes a block of code to perform some action.
- The *declaration* involves writing the block of code or Instructions that defines the action.
- This involves using program code such as *statements, variables, math operators, if/else, loops* etc to program the desired action.
- The syntax is as follows:

```
Declaration:  
Private Sub ProcedureName()  

End Sub  

Procedure Body  
(Block of Code)
```

```
Declaration:  
Public Sub ProcedureName()  

End Sub  

Procedure Body  
(Block of Code)
```

---

**Example 1:** Write a deposit procedure that will get the Account number, Account Balance & Deposit amount from the GUI and perform the deposit operation on the given account

```vbnet
Public Sub Deposit()  
' Example of code inside body using parameter variables:  
Dim decDepositAmount As Decimal  
Dim decAcctBal As Decimal  
Dim intAcctNum As Integer  

decDepositAmount = CDec(txtDepositAmt.Text)  
decAcctBal = CDec(txtAccountBalance.Text)  
intAcctNum = CInt(txtAccountNumber.Text)  

decAcctBal = decAcctBal + decDepositAmount  

txtAccountBalance.Text = CStr(decAcctBal)
End Sub
```
Regular Sub-Procedure Method Call:

- The syntax for calling or executing the method is:

  **Calling:**
  **Syntax:**

  `ProcedureName()`

**Example1:** For the Deposit() Sub Procedure of Example1, the call is:

```
Deposit()
```

‘After the call, the deposit operation has been performed on the values from the controls of the Form

**Example2:** For the Square() procedure of Example2, the call is:

‘We assume somewhere in the Form or Module the following variable exists:
Dim intNumber As Integer
......
```
Square()
```

‘After the call of Square() the variable intNumber will be squared
Sub-Procedure With Arguments Declaration:

- If the Sub Procedure requires that you give it data or arguments, then declaration is as follows:

```vbscript
Public Sub ProcedureName( variable1 As Type, variable2 As Type, etc.. )

'Code inside body will use or manipulate the parameter’s variables

End Sub
```

Example1: Write a deposit procedure that given the account number & account balance, will get the deposit to amount from the GUI and perform the deposit operation on the given account.

```vbscript
Public Sub Deposit(intAcctnum As Integer, ByRef decAcctBal As Decimal)
'Example of code inside body using parameter variables:
Dim decDepositAmount As Decimal

'Get deposit amount from form text box
decDepositAmount = CDec(txtDepositAmt.Text)

'Perform deposit on parameter variable
decAcctBal = decAcctBal + decDepositAmount

'Tell the user deposit was made and on which account
MessageBox.Show("A deposit of" & decDepositAmount & " was made to account" & intAcctNum & " your balance is " & decAcctBal , "Deposit Results", MessageBoxButtons.OK, MessageBoxIcon.Information

End Sub
```

Example2: Write a procedure that squares the value passed as argument

```vbscript
Public Sub Square (intNumber As Integer)
'Code to perform square of variable x here!
intNumber = intNumber * intNumber

End Sub
```
Sub-Procedure With Arguments Method Call:

☐ The syntax for calling or executing the method is:

```
Calling:
Syntax:
ProcedureName( Arg1, Arg2, etc)
```

`Argument list`

Example 1: For the Deposit() Sub Procedure of Example 1, two arguments are passed to the procedure. It is assumed these variables are created elsewhere:

```
Deposit ( intAccountNumber, decAccountBalance)
```

‘After the call, the deposit operation is performed on the variables passed as arguments

Example 2: For the Square() procedure of Example 2, the value to be squared is passed as arguments, the call is:

‘We assume somewhere in the Form or Module the following variable exists:
Dim intNumber As Integer
......
```
Square(intNumber)
```

‘After the call of Square() the variable intNumber will be squared
4.4.6 Function Procedure Declaration & Call

Function Procedure Declaration:

- The Function Procedure Method executes a block of code to perform some action and returns a value.
- The declaration involves declaring the type of data to return in the Header & writing the block of code or Instructions that defines the action and returning the value.
- The following rules apply to Functions:
  - Use the keyword Return to return data to the calling program.
  - The data returned value must be of the same as the return type as specified by the declaration.

- The syntax is as follows:

```
Declaration:
Private Function FunctionName() As ReturnType

End Function

Public Function FunctionName() As ReturnType

Return ReturnValue

End Function
```

Example 1: Write a deposit function that will get the Account number, Account Balance & Deposit amount from the GUI and perform the deposit operation on the given account. The new balance is returned to the calling program.

**Public Function Deposit () As Decimal**

```
'Example of code inside body using parameter variables:
Dim decDepositAmount As Decimal
Dim decAcctBal As Decimal
Dim intAcctNum As Integer

'Get deposit amount, balance & account number from Form
decDepositAmount = CDec(txtDepositAmt.Text)
decAcctBal = CDec(txtAccountBalance.Text)
intAcctNum = CInt(txtAccountNumber.Text)

'Perform deposit on parameter variable
decAcctBal = decAcctBal + decDepositAmount

Return decAcctBal

'Tell the user deposit was made and on which account
MessageBox.Show ("A deposit of" & decDepositAmount & " was made to account" &
intAcctNum & " your balance is" & decAcctBal, "Deposit Results",
MessageBoxButtons.OK, MessageBoxIcon.Information

End Function
```
Regular Function Procedure Call:

- The following rules apply to Function Calls:
  - The data variable that is to receive the value returned by the function must be of the same type as the return type value of the Function declaration.
  - A function can only return one value

- The syntax for calling or executing the method is:

  **Calling**:

  \( \text{Variable} = \text{FunctionName()} \)

---

**Example 2**: Write a procedure that squares a module level variable intNumber and returns the valued squared (Assume this variable was already declared out of the procedure and is global)

Dim intNumber As Integer

**Public Function Square() As Integer**

‘Code to perform square of variable intNumber

intNumber = intNumber * intNumber

Return intNumber

End Function

---

**Example 1**: For the Deposit() Function Procedure of Example 1, assuming that a variable decBalance is created to receive the return value, the call is:

Dim decAccountBalance As Decimal

\( \text{decAccountBalance} = \text{Deposit()} \)

‘After the call, the deposit operation has been performed on the values from the controls of the Form and the balance is returned to the calling program. The variable decAccountBalance contains the new balance

---

**Example 2**: For the Square() Function of Example 2, the call is:

‘We assume somewhere in the Form or Module the following variable exists:

Dim intNumber As Integer

Dim intSquareResults As Integer

\( \text{intSquareResults} = \text{Square()} \)

‘After the call of Square() the variable intNumber will be squared and the function returns the results and assigns it to the variable intSquareResults
Function Procedure with Arguments Declaration:

- If the Sub Procedure requires that you give it data or arguments, then declaration is as follows:

```
Declaration:
Public Function FunctionName( variable1 As Type, variable2 As Type, etc.. ) As ReturnType

'Code inside body will use or manipulate the parameter's variables
Return ReturnValue

End Function
```

Example 1: Write a deposit procedure that given the account number & account balance, will get the deposit to amount from the GUI and perform the deposit operation on the given account. The new balance is returned to the calling program.

```
Public Function Deposit(ByVal intAcctnum As Integer, decAcctBal As Decimal) As Decimal
'Example of code inside body using parameter variables:
Dim decDepositAmount As Decimal

'Get deposit amount from form text box
decDepositAmount = CDec(txtDepositAmt.Text)

'Perform deposit on parameter variable
decAcctBal = decAcctBal + decDepositAmount

'Tell the user deposit was made and on which account
MessageBox.Show(“A deposit of” & decDepositAmount & “ was made to account” & intAcctNum & “ your balance is “ & decAcctBal, “Deposit Results”, MessageBoxButtons.OK, MessageBoxIcon.Information

'Return the balance
Return decAcctBal

End Function
```

Example 2: Write a procedure that squares the value passed as argument and returns the result of the square.

```
Public Sub Square (intNumber As Integer) As Integer
intNumber = intNumber * intNumber

Return intNumber

End Sub
```
Function Procedure With Arguments Call:

- The syntax for calling or executing the method is:

```
Calling:

Variable = FunctionName(Arg1, Arg2, etc)
```

---

**Example1:** For the Deposit() Sub Procedure of Example1, two arguments are passed to the procedure. It is assumed these variables are created elsewhere and a variable is available to store the results.

Dim decAccountBalance As Decimal

```
decAccountBalance = Deposit(intAccountNumber, decAccountBalance)
```

‘After the call, the deposit operation is performed on the variables passed as arguments

---

**Example2:** For the Square() procedure of Example2, the value to be squared is passed as arguments, the call is:

‘We assume somewhere in the Form or Module the following variable exists:
Dim intNumber As Integer
Dim intSquareResults As Integer
......
```
intSquareResults = Square(intNumber)
```

‘After the call of Square() the variable intSquareResults stores the results from the square
4.4.7 Passing Arguments to Methods by Value (ByVal) or by Reference (ByRef)

- When you pass values onto a method, the parameters are passed either by Value (ByVal) or by Reference (ByRef)
- This is a very important distinction that programmers need to clearly understand, otherwise programming errors will occur.
- We are referring how the arguments passed during a method call are transferred to the parameter variables that will store the value within the method.

**Pass-by-Value (ByVal)**

- **Pass-by-value:** When arguments are passed to a method, a *copy* of the argument variable is assigned to the parameter variable inside the method.
- In short, the value that is passed to the parameter by the argument is a copy of the original argument.
- The keyword used to indicate that a parameter variable uses Pass-by-value is the keyword *ByVal*.
- Note that Pass by value is the default in VB.NET, which means you don’t need to specify the key word *ByVal* in the parameter list.
- Why is this so important? Well, if you don’t pay close attention to this, you can develop some serious programming errors.
- Let’s look at an example:

```
Public Sub Deposit(ByVal decAcctBal As Decimal, ByVal decDepAmt As Decimal)
    'Perform deposit on parameter variable
    decAcctBal = decAcctBal + decDepAmt
End Sub
```

---

**Example1a:** Write a deposit procedure that given the account balance & amount to deposit, performs the deposit operation. Note that the balance parameter is passed-by-value.

```
Public Sub Deposit(ByVal decAcctBal As Decimal, ByVal decDepAmt As Decimal)
    'Perform deposit on parameter variable
    decAcctBal = decAcctBal + decDepAmt
End Sub
```
**Example1b - Calling Program:** This code illustrates programming errors caused by not understanding pass-by-value. This calling program, simulates the call to the Deposit(x,x) Sub Procedure of Example1a. Two variables representing the account number and account balance are initialized and passed as argument to the Deposit procedure. The amount to deposit is assumed that is contained in the textbox of a form. The user is informed of the deposit and the new balance:

```vbnet
Dim intMyAccountNumber As Integer
Dim decAccountBalance As Decimal
Dim decDepositAmount As Decimal

'Balance & account number are assigned value
decAccountBalance = 1000.00
intMyAccountNumber = 111

'Account Balance is displayed to user
MessageBox.Show("Your current balance is " & decAccountBalance)

'Get deposit amount from form text box. Assume that the textbox contains $500.00
decDepositAmount = CDec(txtDepositAmt.Text)

'Method is called with variable arguments list to perform the deposit
Deposit (decAccountBalance, decDepositAmount)

'After the call, the user is informed of deposited and the new balance
MessageBox.Show("Account " & intMyAccountNumber & " you have made a deposit of " & decDepositAmount & " you're new balance is " & decAccountBalance)
```

**OUTPUT:**
Your current balance is 1000.00
Account 111 you have made a deposit of 500.00 you're new balance is 1000.00.

---

**Discussion:**

- The program displayed erroneous results. Why?
  - Since the procedure Deposit() passes it arguments by value, the data stored in the parameters `decAcctBal` and `decDepAmt` are a copy of the arguments `decAccountBalance`, `decDepositAmount`.
  - The variables being processed inside the procedure `decAcctBal` and `decDepAmt` are local variables and will not be seen or exist after the procedure completes its execution.
  - Therefore when the user is prompted, the information processed is wrong!!
  - See diagram below for a visual representation of **pass-by-value**: 

---

---

---
Passing by value has the following advantages and disadvantages:

<table>
<thead>
<tr>
<th>Advantages to Pass-by-Value</th>
<th>Disadvantages to Pass-by-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good data protection since original value is not being affected or altered by function call. Only the copy gets exposed to the function.</td>
<td>If an Object containing lots of data is passed and a copy is made, there will be double the memory being used, thus system takes a performance hit.</td>
</tr>
<tr>
<td>Easier to use and understand</td>
<td>System takes a performance hit</td>
</tr>
<tr>
<td></td>
<td>Can only return one value (explained in next section)</td>
</tr>
</tbody>
</table>

- The data being processed by the Parameters are copies of the original Arguments.
- Once method is finished executing, the local variables will NOT exist and will not be seen by the calling program.

To solve the problem above, using pass-by-value, you would need to use a Function instead of a Sub procedure.
Using a Function, you can return the value required from the function.
Pass-by-Reference (ByRef)

- Passing by reference uses a different mechanism, instead of a value being passed to a Method, a reference or an alias to the original variable is passed.
- We say an alias is passed because the argument passed takes on a new name but is the same variable.
- What is actually passed is the address of the original variable.
- In short, the value that is passed to the parameter by the argument is a **POINTER** to the original argument.
- The keyword used to indicate that a parameter variable uses Pass-by-Reference is the keyword **ByRef**
- Let’s look at the previous example but this time we will use pass-by-reference:

```
Example 1a: Write a deposit procedure that given the account balance & amount to deposit, performs the deposit operation. Note that the balance parameter is **passed-by-reference** in this example.

Public Sub Deposit(ByRef decAcctBal As Decimal, ByRef decDepAmt As Decimal)

'Perform deposit on parameter variable
decAcctBal = decAcctBal + decDepAmt

End Sub
```
Discussion:

- This time the program displayed the correct results.
  - Since the procedure Deposit() passes it arguments by reference, the data stored in the parameters `decAcctBal` and `decDepAmt` are pointers or references to the original arguments `decAccountBalance`, `decDepositAmount`.
  - The variables being processed inside the procedure `decAcctBal` and `decDepAmt` are local variables and will not be seen or exist after the procedure completes its execution. Nevertheless, since they are references to the originals, when they no longer exist, the original values have already been altered.
  - See diagram below for a visual representation of pass-by-reference:
Passing by reference has the following advantages and disadvantages:

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helps performance by eliminating the overhead of copying large amount of data by directly accessing the caller’s data.</td>
<td>Security is compromised because the caller’s data can be corrupted by the called method since the original value is accessed.</td>
</tr>
<tr>
<td>Provides a mechanism for returning more than one value form the function back to the calling program</td>
<td>Corrupted data caused by method can go undetected.</td>
</tr>
</tbody>
</table>

- The data being processed by the Parameters are references to the original Arguments.
- The method is actually processing the original data in the arguments NOT a copy.
- At the end of the procedure, the original values will reflect the changes made by the procedure.
Final word on Pass-by-Value & Pass-by-Reference

Which should you use and when?

- This will depend on the procedure you need to write, and what you want in return.
- Usually it is better to use Pass-by-Value since the original data is NOT affected. Bugs caused by procedures corrupting the data object via Pass-by-Reference can go undetected and may be difficult to debug.
- On the other hand, if you know the procedure is processing a large amount of data and you feel the performance hit is too great, then Pass-by-Reference should be the choice, but the programmer needs to take caution and make sure no data corruption occurs.
Homework Assignment 2

Write a Module-Driven Windows application, which uses a custom dialog box representing a login screen used to accept a username and password from the users. The requirements are as follows:

Form Requirements:
1) The form will have two text boxes one for Username and the other for Password. Also two labels and two Command buttons for OK & Cancel. Similar to the previous example
2) Set the properties appropriately so the custom dialog behaves as a login screen dialog.
3) The code inside the form should be kept as simple as possible, DO NOT code any complex code in the OK event of the login form as you are accustomed to from CS101 & CS508. Simple hide the form when OK button is clicked.
4) The Form should NOT perform any complex processing or file Input/Output functions. All I/O functionality should be handled by a Standard Module (See next section)
5) The form is to contain a Sub Method that takes two parameters GetUserInfoAndDisplayForm(U, P). Parameter U represents the Username & Parameter P contains the password. These parameters are to be set as Pass-by-Reference. In the body, the method should make the form display itself and extract the username and password from the login form text boxes and assign to the two parameters. Example of usage from SUB MAIN(): objLoginFormObject.GetUserInfoAndDisplayform(user, pass)

Standard Module Requirements:
6) This program is a Module-Driven Windows application and should be controlled by a Standard Module. What this means is that the program should be controlled from the code inside the Standard Module Sub Main() method. This means that the when the program starts, the startup Object should be the Module Sub Main() and not the login form.
7) The standard module should have all the code to communicate with the user. All messages to the user or I/O functionality should be displayed from the Module Sub Main() procedure ONLY, not the login form.
8) The program should have 5 username: joe, angel, sam, mary, nancy
9) The usernames should be stored in an array of string named UsernameDatabase.
10) The program should have 5 Password: 111, 222, 333, 444, 555
11) The passwords should be stored in an array of integers named PasswordDatabase.
12) The arrays should be declared public and in the declaration section of the module
13) In the Standard Module, create a function named Authenticate(X,Y), which takes two parameters using pass-by-value. The first parameter represents the Username & the second the Password. This Function searches the PasswordDatabase & UsernameDatabase array for the Password/Username and compares the parameters to the values in the arrays to verify if a match is found and returns a Boolean value. If the search is completed with a match another Boolean value is returned. No other operations should be performed in this function, only authentication! This function is to reside in the module, not the Login Form!

Requirements for Sub Main() Procedure in Standard Module:
14) The program always runs or loops, prompting and displaying the user login screen and the result of the login validation.
15) The loop should run until the user enters the magic value for username = -1 and password = -1, in which case the loop should terminate.
16) Once the username & password are entered on the login screen, and the OK button clicked so the form will hide, the program (controlled by Sub Main()) should call the Authenticate Function to verify that these values are valid. From the returned value of the Authenticate Function, the program should decide if access is granted or denied and display message boxes stating if the user has access or reject any unauthorized users. The process should repeat via the loop. REMEMBER DO NOT PERFORM THESE OPERATIONS FROM THE LOGIN FORM OK_CLICK EVENT or THE AUTHENTICATE METHOD. This should be done from the module Sub Main() procedure.

Required Results:
17) Working Project fully debugged and tested.
18) Note that non-working or partially working program will not be accepted. The program has to work. If you are having difficulty than get help or contact the professor for help. No group effort accepted, everyone should work on their own project. Nevertheless, professional courtesy is allowed. You can ask for advice from other students or suggestion if you are stuck, but no code copying. Students should not copy each others code (both students will get an Grade = F if I find you copied)
19) Due Next Class (Monday).