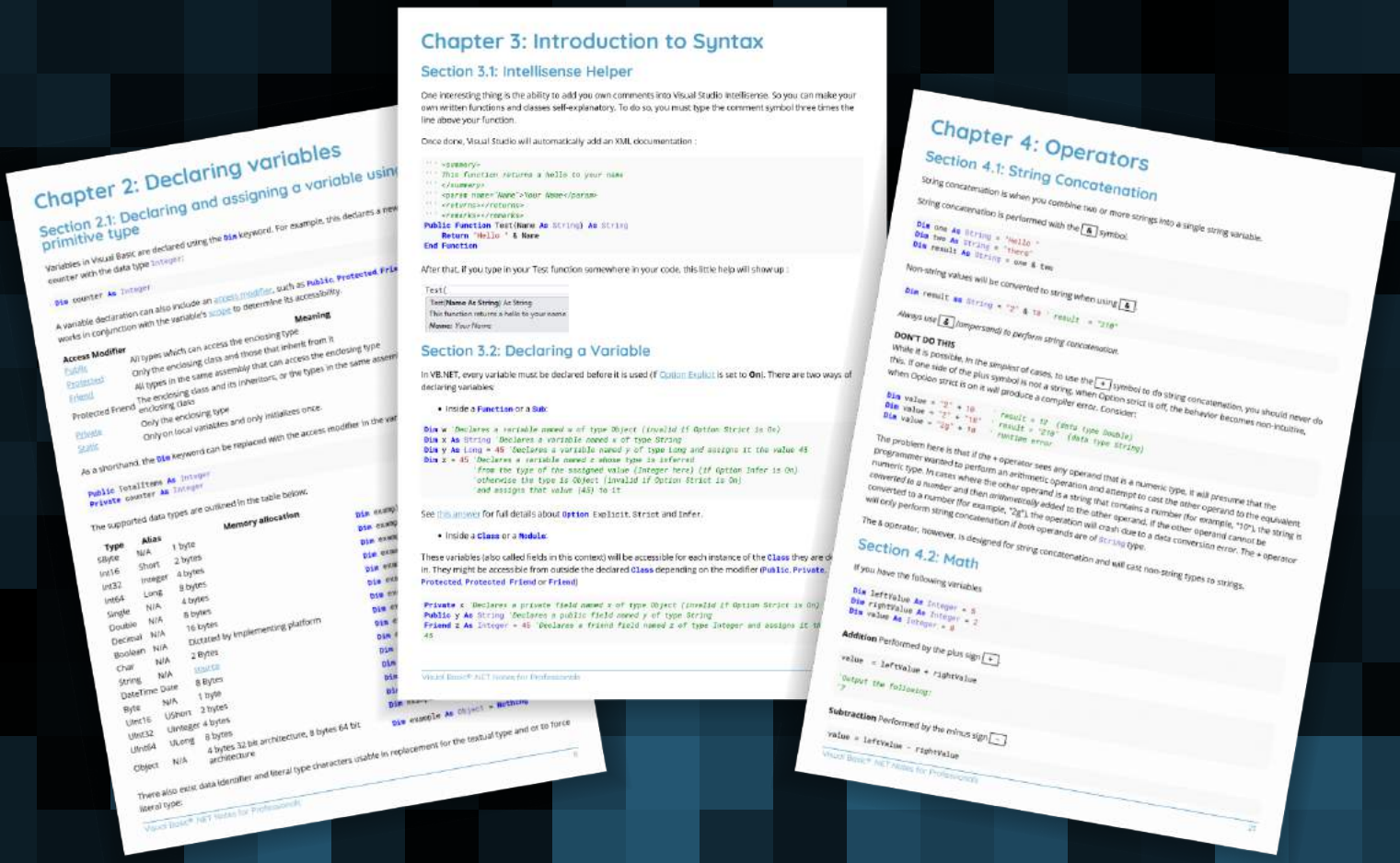
Visual Basic® .NET

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Visual Basic® .NET group(s) or company(s). All trademarks and registered trademarks are the property of their respective owners

Contents

About 1
Chapter 1: Getting started with Visual Basic .NET Language 2
Section 1.1: Hello World 2
Section 1.2: Hello World on a Textbox upon Clicking of a Button 2
Section 1.3: Region 3
Section 1.4: Creating a simple Calculator to get familiar with the interface and code 4
Chapter 2: Declaring variables 8
Section 2.1: Declaring and assigning a variable using a primitive type 8
Section 2.2: Levels of declaration – Local and Member variables 10
Section 2.3: Example of Access Modifiers 11
Chapter 3: Introduction to Syntax 14
Section 3.1: Intellisense Helper 14
Section 3.2: Declaring a Variable 14
Section 3.3: Comments 15
Section 3.4: Modifiers 15
Section 3.5: Object Initializers 16
Section 3.6: Collection Initializer 17
Section 3.7: Writing a function 19
Chapter 4: Operators 21
Section 4.1: String Concatenation 21
Section 4.2: Math 21
Section 4.3: Assignment 22
Section 4.4: Comparison 23
Section 4.5: Bitwise 23
Chapter 5: Conditions 25
Section 5.1: If operator 25
Section 5.2: IF...Then...Else 25
Chapter 6: Short-Circuiting Operators (AndAlso - OrElse) 27
Section 6.1: OrElse Usage 27
Section 6.2: AndAlso Usage 27
Section 6.3: Avoiding NullReferenceException 27
Chapter 7: Date 30
Section 7.1: Converting (Parsing) a String to a Date 30
Section 7.2: Converting a Date To A String 30
Chapter 8: Array 31
Section 8.1: Array definition 31
Section 8.2: Null Array Variables 31
Section 8.3: Array initialization 32
Section 8.4: Declare a single-dimension array and set array element values 32
Section 8.5: Jagged Array Initialization 32
Section 8.6: Non-zero lower bounds 32
Section 8.7: Referencing Same Array from Two Variables 33
Section 8.8: Multidimensional Array initialization 33
Chapter 9: Lists 34
Section 9.1: Add items to a List 34
Section 9.2: Check if item exists in a List 34

Section 9.3: Loop through items in list ... 34
Section 9.4: Create a List ... 35
Section 9.5: Remove items from a List ... 36
Section 9.6: Retrieve items from a List ... 36
Chapter 10: Enum ... 38
Section 10.1: GetNames() ... 38
Section 10.2: HasFlag() ... 38
Section 10.3: Enum definition ... 39
Section 10.4: Member initialization ... 39
Section 10.5: The Flags attribute ... 39
Section 10.6: GetValues() ... 40
Section 10.7: String parsing ... 40
Section 10.8: ToString() ... 41
Section 10.9: Determine whether a Enum has FlagsAttribute specified or not ... 41
Section 10.10: For-each flag (flag iteration) ... 42
Section 10.11: Determine the amount of flags in a flag combination ... 42
Section 10.12: Find the nearest value in a Enum ... 43
Chapter 11: Dictionaries ... 45
Section 11.1: Create a dictionary filled with values ... 45
Section 11.2: Loop through a dictionary and print all entries ... 45
Section 11.3: Checking for key already in dictionary - data reduction ... 45
Section 11.4: Getting a dictionary value ... 46
Chapter 12: Looping ... 47
Section 12.1: For...Next ... 47
Section 12.2: For Each...Next loop for looping through collection of items ... 48
Section 12.3: Short Circuiting ... 48
Section 12.4: While loop to iterate while some condition is true ... 50
Section 12.5: Nested Loop ... 50
Section 12.6: Do...Loop ... 51
Chapter 13: File Handling ... 53
Section 13.1: Write Data to a File ... 53
Section 13.2: Read All Contents of a File ... 53
Section 13.3: Write Lines Individually to a Text File using StreamWriter ... 53
Chapter 14: File/Folder Compression ... 54
Section 14.1: Adding File Compression to your project ... 54
Section 14.2: Creating zip archive from directory ... 54
Section 14.3: Extracting zip archive to directory ... 54
Section 14.4: Create zip archive dynamicaly ... 54
Chapter 15: Connection Handling ... 55
Section 15.1: Public connection property ... 55
Chapter 16: Data Access ... 56
Section 16.1: Read field from Database ... 56
Section 16.2: Simple Function to read from Database and return as DataTable ... 57
Chapter 17: Type conversion ... 58
Section 17.1: Converting Text of The Textbox to an Integer ... 58
Chapter 18: ByVal and ByRef keywords ... 59
Section 18.1: ByRef keyword ... 59
Section 18.2: ByVal keyword ... 59
Chapter 19: Console ... 61

www.dbooks.org

Section 19.1: Console.ReadLine() 61
Section 19.2: Console.Read() 61
Section 19.3: Console.ReadKey() 61
Section 19.4: Prototype of command line prompt 61
Section 19.5: Console.WriteLine() 62
Chapter 20: Functions 63
Section 20.1: Defining a Function 63
Section 20.2: Defining a Function #2 63
Chapter 21: Recursion 64
Section 21.1: Compute nth Fibonacci number 64
Chapter 22: Random 65
Section 22.1: Declaring an instance 65
Section 22.2: Generate a random number from an instance of Random 65
Chapter 23: Classes 67
Section 23.1: Abstract Classes 67
Section 23.2: Creating classes 67
Chapter 24: Generics 69
Section 24.1: Create a generic class 69
Section 24.2: Instance of a Generic Class 69
Section 24.3: Define a 'generic' class 69
Section 24.4: Use a generic class 69
Section 24.5: Limit the possible types given 70
Section 24.6: Create a new instance of the given type 70
Chapter 25: Disposable objects 71
Section 25.1: Basic concept of IDisposable 71
Section 25.2: Declaring more objects in one Using 71
Chapter 26: NullReferenceException 73
Section 26.1: Empty Return 73
Section 26.2: Uninitialized variable 73
Chapter 27: Using Statement 74
Section 27.1: See examples under Disposable objects 74
Chapter 28: Option Strict 75
Section 28.1: Why Use It? 75
Section 28.2: How to Switch It On 75
Chapter 29: Option Explicit 77
Section 29.1: What is it? 77
Section 29.2: How to switch it on? 77
Chapter 30: Option Infer 78
Section 30.1: How to enable/disable it 78
Section 30.2: What is it? 78
Section 30.3: When to use type inference 79
Chapter 31: Error Handling 81
Section 31.1: Try...Catch...Finally Statement 81
Section 31.2: Creating custom exception and throwing 81
Section 31.3: Try Catch in Database Operation 82
Section 31.4: The Un-catchable Exception 82
Section 31.5: Critical Exceptions 82
Chapter 32: OOP Keywords 84

Section 32.1: Defining a class 84
Section 32.2: Inheritance Modifiers (on classes) 84
Section 32.3: Inheritance Modifiers (on properties and methods) 85
Section 32.4: MyBase 86
Section 32.5: Me vs MyClass 87
Section 32.6: Overloading 88
Section 32.7: Shadows 88
Section 32.8: Interfaces 90
Chapter 33: Extension methods 91
Section 33.1: Creating an extension method 91
Section 33.2: Making the language more functional with extension methods 91
Section 33.3: Getting Assembly Version From Strong Name 91
Section 33.4: Padding Numerics 92
Chapter 34: Reflection 94
Section 34.1: Retrieve Properties for an Instance of a Class 94
Section 34.2: Get a method and invoke it 94
Section 34.3: Create an instance of a generic type 94
Section 34.4: Get the members of a type 94
Chapter 35: Visual Basic 14.0 Features 96
Section 35.1: Null conditional operator 96
Section 35.2: String interpolation 96
Section 35.3: Read-Only Auto-Properties 97
Section 35.4: NameOf operator 97
Section 35.5: Multiline string literals 98
Section 35.6: Partial Modules and Interfaces 98
Section 35.7: Comments after implicit line continuation 99
Section 35.8: #Region directive improvements 99
Chapter 36: LINQ 101
Section 36.1: Selecting from array with simple condition 101
Section 36.2: Mapping array by Select clause 101
Section 36.3: Ordering output 101
Section 36.4: Generating Dictionary From IEnumerable 101
Section 36.5: Projection 102
Section 36.6: Getting distinct values (using the Distinct method) 102
Chapter 37: FTP server 103
Section 37.1: Download file from FTP server 103
Section 37.2: Download file from FTP server when login required 103
Section 37.3: Upload file to FTP server 103
Section 37.4: Upload file to FTP server when login required 103
Chapter 38: Working with Windows Forms 104
Section 38.1: Using the default Form instance 104
Section 38.2: Passing Data From One Form To Another 104
Chapter 39: Google Maps in a Windows Form 106
Section 39.1: How to use a Google Map in a Windows Form 106
Chapter 40: GDI+ 115
Section 40.1: Draw Shapes 115
Section 40.2: Fill Shapes 115
Section 40.3: Text 116
Section 40.4: Create Graphic Object 116

www.dbooks.org

Chapter 41: WinForms SpellCheckBox 118
Section 41.1: ElementHost WPF TextBox 118
Chapter 42: Using axWindowsMediaPlayer in VB.Net 122
Section 42.1: Adding the axWindowsMediaPlayer 122
Section 42.2: Play a Multimedia File 123
Chapter 43: WPF XAML Data Binding 124
Section 43.1: Binding a String in the ViewModel to a TextBox in the View 124
Chapter 44: Reading compressed textfile on-the-fly 126
Section 44.1: Reading .gz textfile line after line 126
Chapter 45: Threading 127
Section 45.1: Performing thread-safe calls using Control.Invoke() 127
Section 45.2: Performing thread-safe calls using Async/Await 127
Chapter 46: Multithreading 129
Section 46.1: Multithreading using Thread Class 129
Chapter 47: BackgroundWorker 131
Section 47.1: Using BackgroundWorker 131
Section 47.2: Accessing GUI components in BackgroundWorker 132
Chapter 48: Using BackgroundWorker 133
Section 48.1: Basic implementation of Background worker class 133
Chapter 49: Task-based asynchronous pattern 134
Section 49.1: Basic usage of Async/Await 134
Section 49.2: Using TAP with LINQ 134
Chapter 50: Debugging your application 135
Section 50.1: Debug in the console 135
Section 50.2: Indenting your debug output 135
Section 50.3: Debug in a text file 136
Chapter 51: Unit Testing in VB.NET 137
Section 51.1: Unit Testing for Tax Calculation 137
Section 51.2: Testing Employee Class assigned and derived Properties 138
Credits 141
You may also like 143

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/VisualBasic_NETBook

This *Visual Basic® .NET Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Visual Basic® .NET group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Visual Basic .NET Language

VB.NET Version	Visual Studio Version	.NET Framework Version	Release Date
7.0	2002	1.0	2002-02-13
7.1	2003	1.1	2003-04-24
8.0	2005	2.0 / 3.0	2005-10-18
9.0	2008	3.5	2007-11-19
10.0	2010	4.0	2010-04-12
11.0	2012	4.5	2012-08-15
12.0	2013	4.5.1 / 4.5.2	2013-10-17
14.0	2015	4.6.0 ~ 4.6.2	2015-07-20
15.0	2017	4.7	2017-03-07

Section 1.1: Hello World

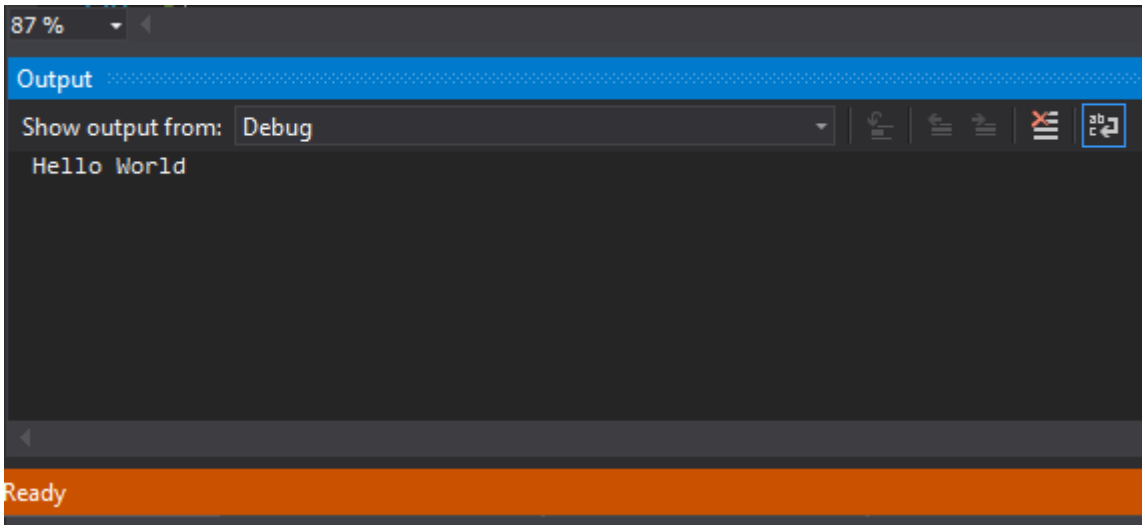
First, install a version of Microsoft Visual Studio, including the free Community edition. Then, create a Visual Basic Console Application project of type *Console Application*, and the following code will print the string `'Hello World'` to the Console:

```
Module Module1

    Sub Main()
        Console.WriteLine("Hello World")
    End Sub

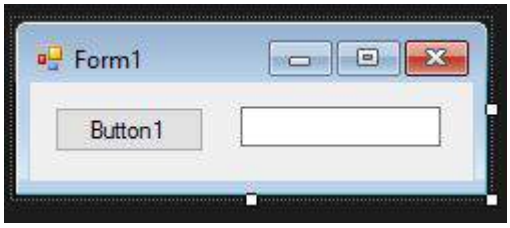
End Module
```

Then, save and press F5 on the keyboard (or go to the *Debug* menu, then click *Run without Debug* or *Run*) to compile and run the program. `'Hello World'` should appear in the console window.



Section 1.2: Hello World on a Textbox upon Clicking of a Button

Drag 1 textbox and 1 button



Double click the button1 and you will be transferred to the `Button1_Click` **event**

```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

    End Sub
End Class
```

Type the name of the object that you want to target, in our case it is the `textbox1`. `.Text` is the property that we want to use if we want to put a text on it.

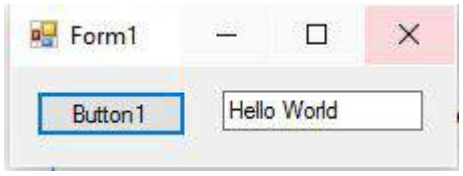
Property `Textbox.Text, gets` **or** `sets the current text` **in** `the TextBox`. Now, we have `Textbox1.Text`

We need to set the value of that `Textbox1.Text` so we will use the `=` sign. The value that we want to put in the `Textbox1.Text` is `Hello World`. Overall, this is the total code for putting a value of `Hello World` to the `Textbox1.Text`

```
TextBox1.Text = "Hello World"
```

Adding that code to the `clicked` **event** of `button1`

```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        TextBox1.Text = "Hello World"
    End Sub
End Class
```



Section 1.3: Region

For the sake of readability, which will be useful for beginners when reading VB code as well for full time developers to maintain the code, we can use "Region" to set a region of the same set of events, functions, or variables:

```
#Region "Events"
    Protected Sub txtPrice_TextChanged(...) Handles txtPrice.TextChanged
        'Do the ops here...
    End Sub

    Protected Sub txtTotal_TextChanged(...) Handles txtTotal.TextChanged
        'Do the ops here...
    End Sub

    'Some other events....

#End Region
```

This region block could be collapsed to gain some visual help when the code row goes to 1000+. It is also save your scroll efforts.

```
Imports System.Data
Imports System.Data.SqlClient
Imports ClassFunction
Imports CrystalDecisions.CrystalReports.Engine
Imports CrystalDecisions.Shared
Imports CrystalDecisions.ReportSource
Imports CrystalDecisions.Reporting
Partial Class transaction_trnBPB_PCH_JPS_Tekhnik
    Inherits System.Web.UI.Page
Variables

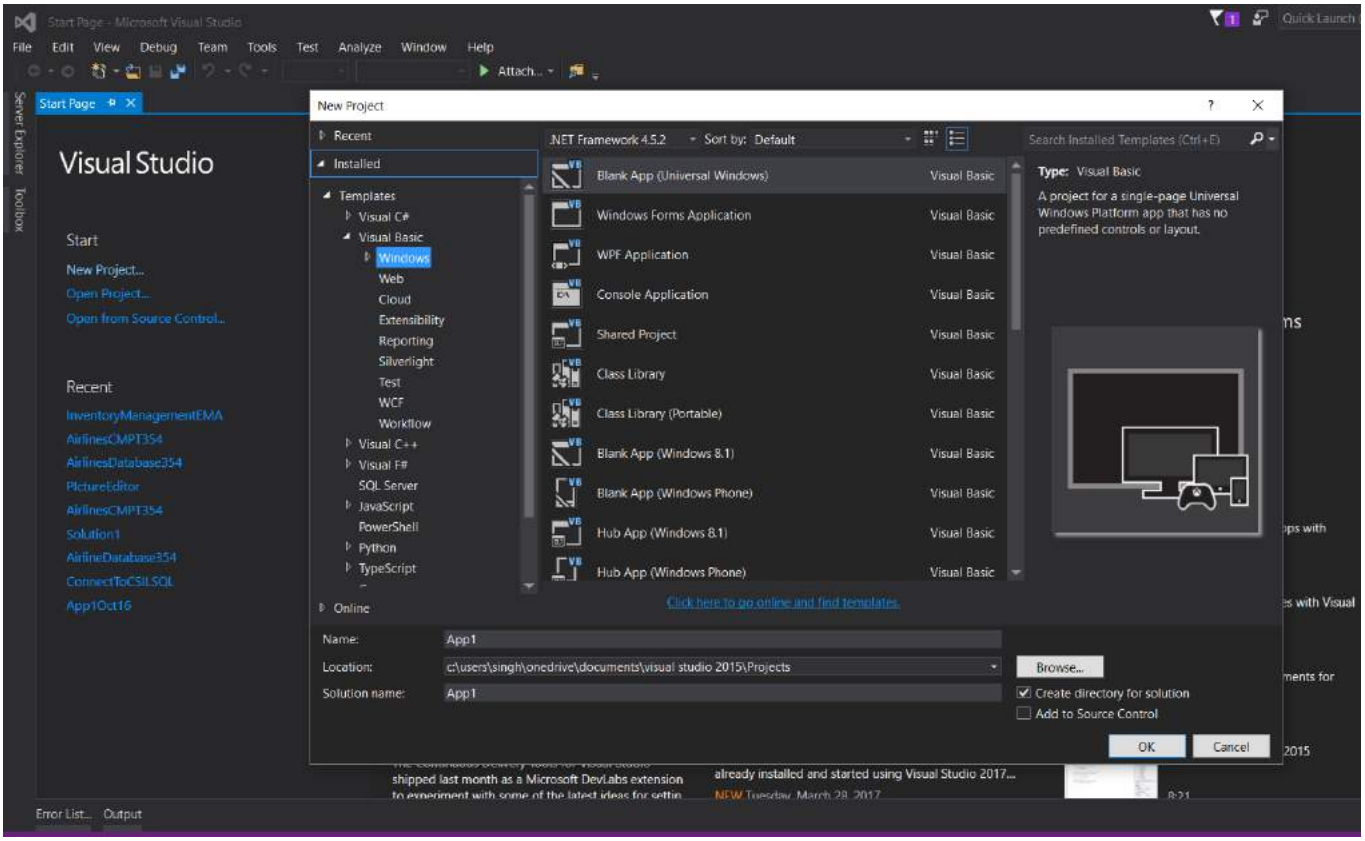
#Region "Functions"
    Private Function GenerateOrderNo ...
    Sub CreateTableTBLDTL ...
    Protected Function getDateToPeriodAcctg ...
#End Region

Procedures

Event
End Class
```

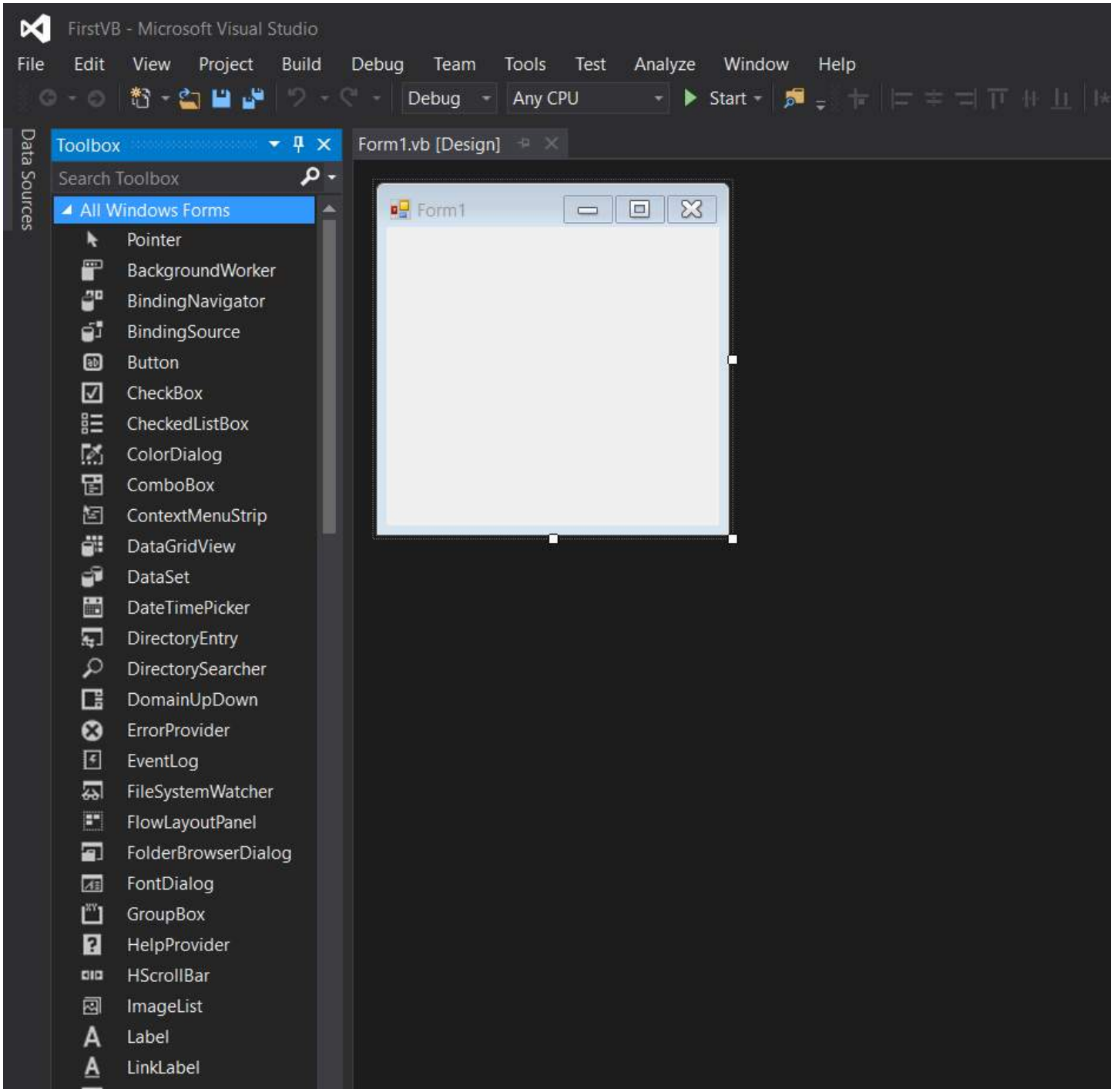
Tested on VS 2005, 2008 2010, 2015 and 2017.

Section 1.4: Creating a simple Calculator to get familiar with the interface and code

1. Once you have installed Visual Studio from https://www.visualstudio.com/downloads/, start a new project.
2. 

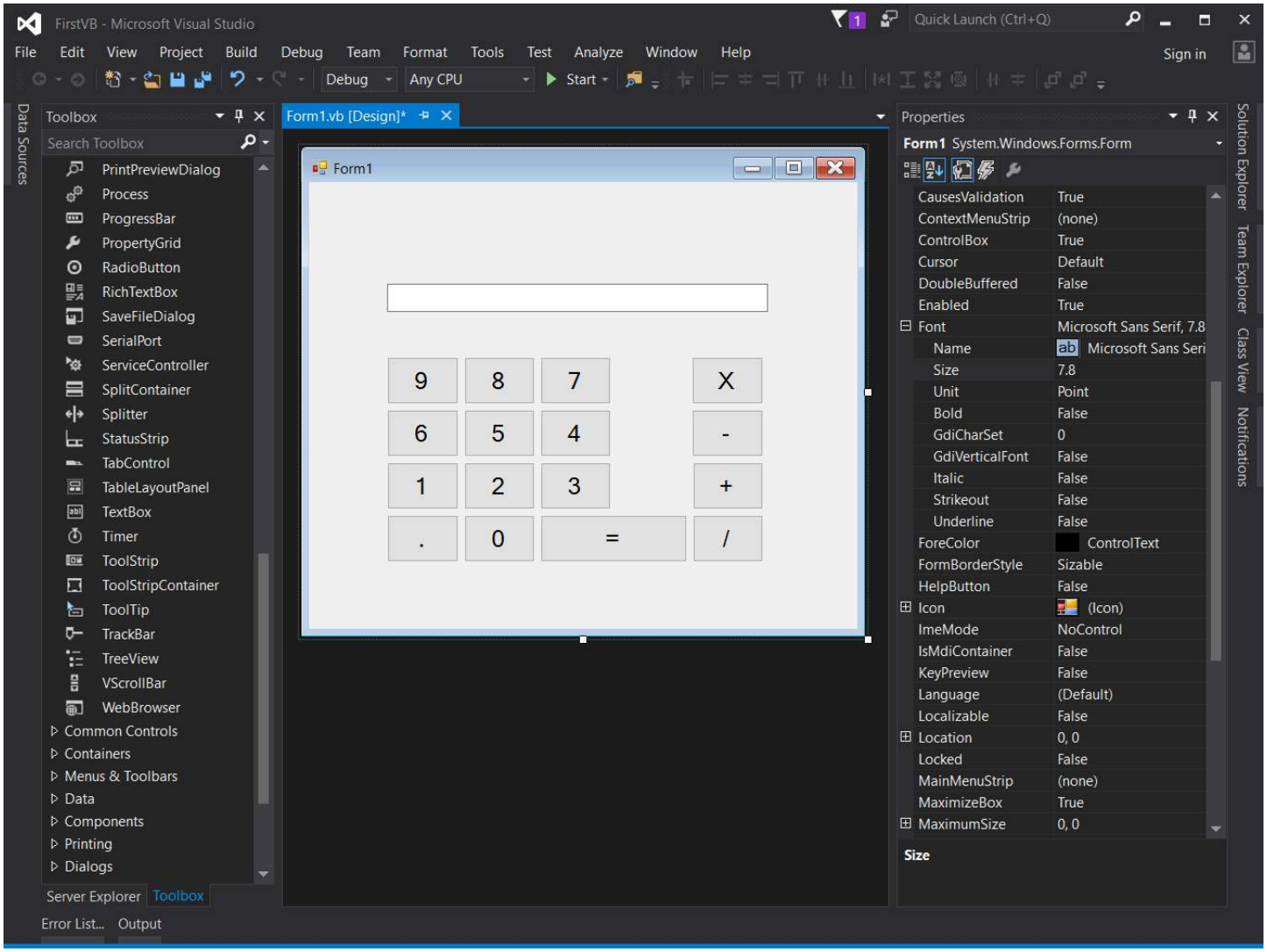
3. Select 'Windows Forms Application' from Visual Basic Tab. You can rename it here if you need to.

4. Once you click 'OK', you will see this window:



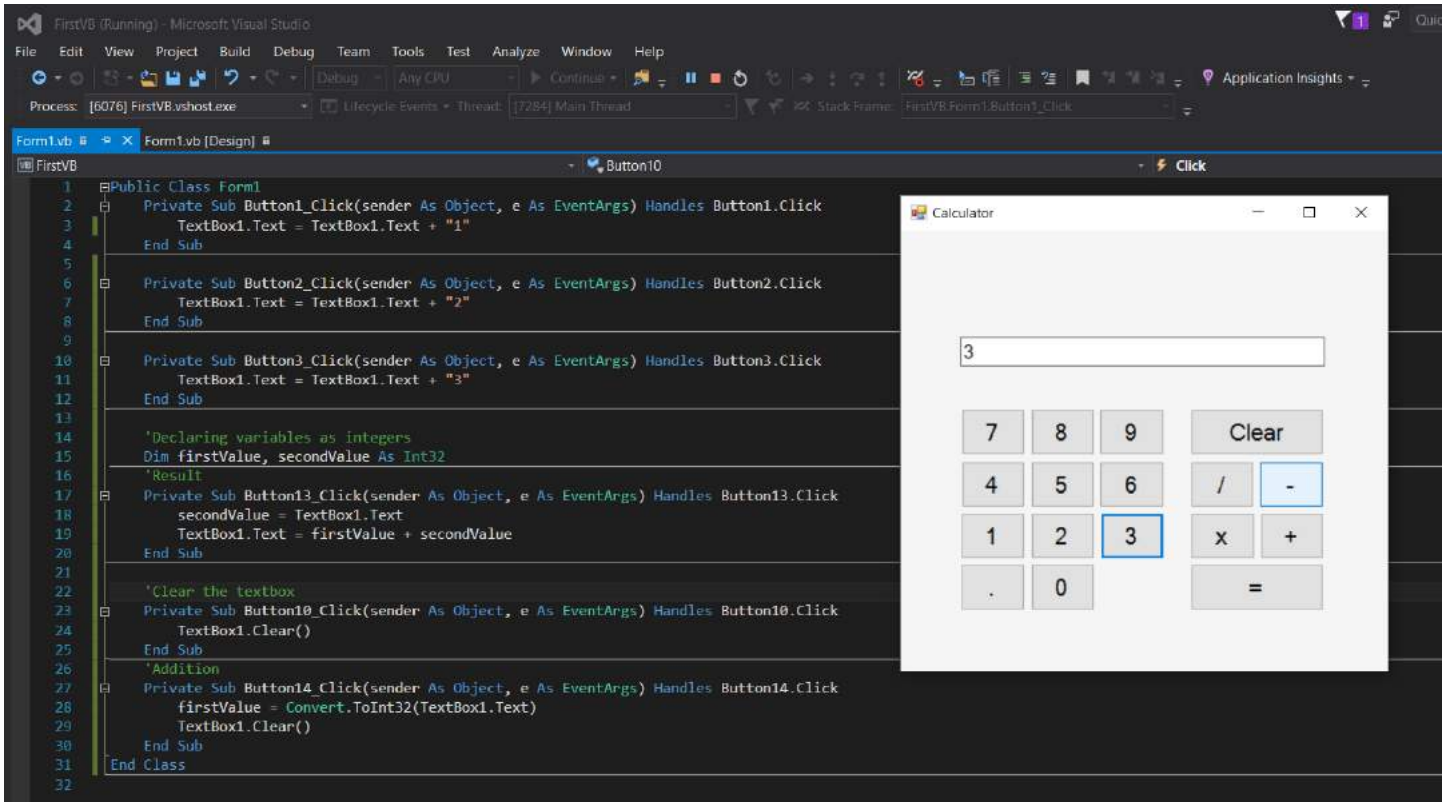
5. Click on the 'Toolbox' tab on the left. The toolbar has 'auto-hide' option enabled by default. To disable this option, click the small symbol between the 'down arrow' symbol and the 'x' symbol, on the top-right corner of Toolbox window.

6. Get yourself familiar with the tools provided in the box. I have made a calculator interface by using buttons and a Textbox.



7. Click on the *Properties* tab (It is on the right side of the editor). You can change the *Text* property of a button, and the textbox to rename them. *Font* property can be used to alter the font of the controls.

8. To write the specific action for an event(eg. clicking on a button), double click on the control. Code window will open.



9. VB.Net is a powerful language designed for fast development. High encapsulation and abstraction is cost for it. You do not need to add *semicolon* to indicate the end of a statement, there are no brackets, and most of the time, it auto-corrects the case of the alphabets.
10. Code provided in the picture should be simple to understand. *Dim* is the keyword used to initialize a variable, and *new* allocates memory. Anything you type in the textbox is of type *string* by default. Casting is required to use the value as a different type.

Enjoy your first creation in VB.Net!

www.dbooks.org

Chapter 2: Declaring variables

Section 2.1: Declaring and assigning a variable using a primitive type

Variables in Visual Basic are declared using the `Dim` keyword. For example, this declares a new variable called `counter` with the data type `Integer`:

```
Dim counter As Integer
```

A variable declaration can also include an access modifier, such as `Public`, `Protected`, `Friend`, or `Private`. This works in conjunction with the variable's scope to determine its accessibility.

Access Modifier	Meaning
Public	All types which can access the enclosing type
Protected	Only the enclosing class and those that inherit from it
Friend	All types in the same assembly that can access the enclosing type
Protected Friend	The enclosing class and its inheritors, *or* the types in the same assembly that can access the enclosing class
Private	Only the enclosing type
Static	Only on local variables and only initializes once.

As a shorthand, the `Dim` keyword can be replaced with the access modifier in the variable's declaration:

```
Public TotalItems As Integer
Private counter As Integer
```

The supported data types are outlined in the table below:

Type	Alias	Memory allocation	Example
SByte	N/A	1 byte	`Dim example As SByte = 10`
Int16	Short	2 bytes	`Dim example As Short = 10`
Int32	Integer	4 bytes	`Dim example As Integer = 10`
Int64	Long	8 bytes	`Dim example As Long = 10`
Single	N/A	4 bytes	`Dim example As Single = 10.95`
Double	N/A	8 bytes	`Dim example As Double = 10.95`
Decimal	N/A	16 bytes	`Dim example As Decimal = 10.95`
Boolean	N/A	Dictated by implementing platform	`Dim example As Boolean = True`
Char	N/A	2 Bytes	`Dim example As Char = "A"C`
String	N/A	source	`Dim example As String = "Stack Overflow"`
DateTime	Date	8 Bytes	`Dim example As Date = Date.Now`
Byte	N/A	1 byte	`Dim example As Byte = 10`
UInt16	UShort	2 bytes	`Dim example As UShort = 10`
UInt32	UInteger	4 bytes	`Dim example As UInteger = 10`
UInt64	ULong	8 bytes	`Dim example As ULong = 10`
Object	N/A	4 bytes 32 bit architecture, 8 bytes 64 bit architecture	`Dim example As Object = Nothing`

There also exist data identifier and literal type characters usable in replacement for the textual type and or to force literal type:

Type (or Alias)	Identifier type character	Literal type character
Short	N/A	`example = 10S`
Integer	`Dim example%`	`example = 10%` or `example = 10I`
Long	`Dim example&`	`example = 10&` or `example = 10L`
Single	`Dim example!`	`example = 10!` or `example = 10F`
Double	`Dim example#`	`example = 10#` or `example = 10R`
Decimal	`Dim example@`	`example = 10@` or `example = 10D`
Char	N/A	`example = "A"C`
String	`Dim example$`	N/A
UShort	N/A	`example = 10US`
UInteger	N/A	`example = 10UI`
ULong	N/A	`example = 10UL`

The integral suffixes are also usable with hexadecimal (&H) or octal (&O) prefixes:
`example = &H8000S` or `example = &O77&`

Date(Time) objects can also be defined using literal syntax:
```
Dim example As Date = #7/26/2016 12:8 PM#
```

Once a variable is declared it will exist within the Scope of the containing type, **`Sub`** or **`Function`** declared, as an example:

```
Public Function IncrementCounter() As Integer
    Dim counter As Integer = 0
    counter += 1

    Return counter
End Function
```

The counter variable will only exist until the **`End Function`** and then will be out of scope. If this counter variable is needed outside of the function you will have to define it at class/structure or module level.

```
Public Class ExampleClass

    Private _counter As Integer

    Public Function IncrementCounter() As Integer
       _counter += 1
       Return _counter
    End Function

End Class
```

Alternatively, you can use the **`Static`** (not to be confused with **`Shared`**) modifier to allow a local variable to retain it's value between calls of its enclosing method:

```
Function IncrementCounter() As Integer
    Static counter As Integer = 0
    counter += 1

    Return counter
End Function
```


www.dbooks.org

Section 2.2: Levels of declaration – Local and Member variables

Local variables - Those declared within a procedure (subroutine or function) of a class (or other structure). In this example, `exampleLocalVariable` is a local variable declared within `ExampleFunction()`:

```
Public Class ExampleClass1

    Public Function ExampleFunction() As Integer
        Dim exampleLocalVariable As Integer = 3
        Return exampleLocalVariable
    End Function

End Class
```

The `Static` keyword allows a local variable to be retained and keep its value after termination (where usually, local variables cease to exist when the containing procedure terminates).

In this example, the console is `024`. On each call to `ExampleSub()` from `Main()` the static variable retains the value it had at the end of the previous call:

```
Module Module1

    Sub Main()
        ExampleSub()
        ExampleSub()
        ExampleSub()
    End Sub

    Public Sub ExampleSub()
        Static exampleStaticLocalVariable As Integer = 0
        Console.Write(exampleStaticLocalVariable.ToString)
        exampleStaticLocalVariable += 2
    End Sub

End Module
```

Member variables - Declared outside of any procedure, at the class (or other structure) level. They may be **instance variables**, in which each instance of the containing class has its own distinct copy of that variable, or `Shared` **variables**, which exist as a single variable associated with the class itself, independent of any instance.

Here, `ExampleClass2` contains two member variables. Each instance of the `ExampleClass2` has an individual `ExampleInstanceVariable` which can be accessed via the class reference. The shared variable `ExampleSharedVariable` however is accessed using the class name:

```
Module Module1

    Sub Main()

        Dim instance1 As ExampleClass4 = New ExampleClass4
        instance1.ExampleInstanceVariable = "Foo"

        Dim instance2 As ExampleClass4 = New ExampleClass4
        instance2.ExampleInstanceVariable = "Bar"

        Console.WriteLine(instance1.ExampleInstanceVariable)
        Console.WriteLine(instance2.ExampleInstanceVariable)
        Console.WriteLine(ExampleClass4.ExampleSharedVariable)
```

```
    End Sub

    Public Class ExampleClass4

        Public ExampleInstanceVariable As String
        Public Shared ExampleSharedVariable As String = "FizzBuzz"

    End Class

End Module
```

Section 2.3: Example of Access Modifiers

In the following example consider you have a solution hosting two projects: **ConsoleApplication1** and **SampleClassLibrary**. The first project will have the classes **SampleClass1** and **SampleClass2**. The second one will have **SampleClass3** and **SampleClass4**. In other words we have two assemblies with two classes each. **ConsoleApplication1** has a reference to **SampleClassLibrary**.

See how **SampleClass1.MethodA** interacts with other classes and methods.

SampleClass1.vb:

```
Imports SampleClassLibrary

Public Class SampleClass1
    Public Sub MethodA()
        'MethodA can call any of the following methods because
        'they all are in the same scope.
        MethodB()
        MethodC()
        MethodD()
        MethodE()

        'Sample2 is defined as friend. It is accessible within
        'the type itself and all namespaces and code within the same assembly.
        Dim class2 As New SampleClass2()
        class2.MethodA()
        'class2.MethodB() 'SampleClass2.MethodB is not accessible because
                          'this method is private. SampleClass2.MethodB
                          'can only be called from SampleClass2.MethodA,
                          'SampleClass2.MethodC, SampleClass2.MethodD
                          'and SampleClass2.MethodE
        class2.MethodC()
        'class2.MethodD() 'SampleClass2.MethodD is not accessible because
                          'this method is protected. SampleClass2.MethodD
                          'can only be called from any class that inherits
                          'SampleClass2, SampleClass2.MethodA, SampleClass2.MethodC,
                          'SampleClass2.MethodD and SampleClass2.MethodE
        class2.MethodE()

        Dim class3 As New SampleClass3() 'SampleClass3 resides in other
                                         'assembly and is defined as public.
                                         'It is accessible anywhere.
        class3.MethodA()
        'class3.MethodB() 'SampleClass3.MethodB is not accessible because
                          'this method is private. SampleClass3.MethodB can
                          'only be called from SampleClass3.MethodA,
                          'SampleClass3.MethodC, SampleClass3.MethodD
                          'and SampleClass3.MethodE
```


www.dbooks.org

```
        'class3.MethodC() 'SampleClass3.MethodC is not accessible because
                          'this method is friend and resides in another assembly.
                          'SampleClass3.MethodC can only be called anywhere from the
                          'same assembly, SampleClass3.MethodA, SampleClass3.MethodB,
                          'SampleClass3.MethodD and SampleClass3.MethodE

        'class4.MethodD() 'SampleClass3.MethodE is not accessible because
                          'this method is protected friend. SampleClass3.MethodD
                          'can only be called from any class that resides inside
                          'the same assembly and inherits SampleClass3,
                          'SampleClass3.MethodA, SampleClass3.MethodB,
                          'SampleClass3.MethodC and SampleClass3.MethodD

        'Dim class4 As New SampleClass4() 'SampleClass4 is not accessible because
                                          'it is defined as friend and resides in
                                          'other assembly.
    End Sub

    Private Sub MethodB()
        'Doing MethodB stuff...
    End Sub

    Friend Sub MethodC()
        'Doing MethodC stuff...
    End Sub

    Protected Sub MethodD()
        'Doing MethodD stuff...
    End Sub

    Protected Friend Sub MethodE()
        'Doing MethodE stuff...
    End Sub
End Class
```

SampleClass2.vb:

```
Friend Class SampleClass2
    Public Sub MethodA()
        'Doing MethodA stuff...
    End Sub

    Private Sub MethodB()
        'Doing MethodB stuff...
    End Sub

    Friend Sub MethodC()
        'Doing MethodC stuff...
    End Sub

    Protected Sub MethodD()
        'Doing MethodD stuff...
    End Sub

    Protected Friend Sub MethodE()
        'Doing MethodE stuff...
    End Sub
End Class
```

SampleClass3.vb:

```
Public Class SampleClass3
    Public Sub MethodA()
        'Doing MethodA stuff...
    End Sub
    Private Sub MethodB()
        'Doing MethodB stuff...
    End Sub

    Friend Sub MethodC()
        'Doing MethodC stuff...
    End Sub

    Protected Sub MethodD()
        'Doing MethodD stuff...
    End Sub

    Protected Friend Sub MethodE()
        'Doing MethodE stuff...
    End Sub
End Class
```

SampleClass4.vb:

```
Friend Class SampleClass4
    Public Sub MethodA()
        'Doing MethodA stuff...
    End Sub
    Private Sub MethodB()
        'Doing MethodB stuff...
    End Sub

    Friend Sub MethodC()
        'Doing MethodC stuff...
    End Sub

    Protected Sub MethodD()
        'Doing MethodD stuff...
    End Sub

    Protected Friend Sub MethodE()
        'Doing MethodE stuff...
    End Sub
End Class
```

Chapter 3: Introduction to Syntax

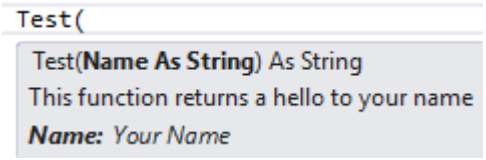
Section 3.1: Intellisense Helper

One interesting thing is the ability to add you own comments into Visual Studio Intellisense. So you can make your own written functions and classes self-explanatory. To do so, you must type the comment symbol three times the line above your function.

Once done, Visual Studio will automatically add an XML documentation :

```
''' <summary>
''' This function returns a hello to your name
''' </summary>
''' <param name="Name">Your Name</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Test(Name As String) As String
    Return "Hello " & Name
End Function
```

After that, if you type in your Test function somewhere in your code, this little help will show up :



Section 3.2: Declaring a Variable

In VB.NET, every variable must be declared before it is used (If Option Explicit is set to **On**). There are two ways of declaring variables:

- Inside a `Function` or a `Sub`:

```
Dim w 'Declares a variable named w of type Object (invalid if Option Strict is On)
Dim x As String 'Declares a variable named x of type String
Dim y As Long = 45 'Declares a variable named y of type Long and assigns it the value 45
Dim z = 45 'Declares a variable named z whose type is inferred
           'from the type of the assigned value (Integer here) (if Option Infer is On)
           'otherwise the type is Object (invalid if Option Strict is On)
           'and assigns that value (45) to it
```

See this answer for full details about `Option Explicit`, `Strict` and `Infer`.

- Inside a `Class` or a `Module`:

These variables (also called fields in this context) will be accessible for each instance of the `Class` they are declared in. They might be accessible from outside the declared `Class` depending on the modifier (`Public`, `Private`, `Protected`, `Protected Friend` or `Friend`)

```
Private x 'Declares a private field named x of type Object (invalid if Option Strict is On)
Public y As String 'Declares a public field named y of type String
Friend z As Integer = 45 'Declares a friend field named z of type Integer and assigns it the value
45
```

These fields can also be declared with `Dim` but the meaning changes depending on the enclosing type:

```
Class SomeClass
    Dim z As Integer = 45 ' Same meaning as Private z As Integer = 45
End Class

Structure SomeStructure
    Dim y As String ' Same meaning as Public y As String
End Structure
```

Section 3.3: Comments

The first interesting thing to know is how to write comments.

In VB .NET, you write a comment by writing an apostrophe ' or writing `REM`. This means the rest of the line will not be taken into account by the compiler.

```
'This entire line is a comment
Dim x As Integer = 0 'This comment is here to say we give 0 value to x

REM There are no such things as multiline comments
'So we have to start everyline with the apostrophe or REM
```

Section 3.4: Modifiers

Modifiers are a way to indicate how external objects can access an object's data.

- Public

Means any object can access this without restriction

- Private

Means only the declaring object can access and view this

- Protected

Means only the declaring object and any object that inherits from it can access and view this.

- Friend

Means only the delcaring object, any object that inherits from it and any object in the same namespace can access and view this.

```
Public Class MyClass
    Private x As Integer

    Friend Property Hello As String

    Public Sub New()
    End Sub

    Protected Function Test() As Integer
        Return 0
    End Function
End Class
```

Section 3.5: Object Initializers

- Named Types

```
Dim someInstance As New SomeClass(argument) With {
    .Member1 = value1,
    .Member2 = value2
    '...
}
```

Is equivalent to

```
Dim someInstance As New SomeClass(argument)
someInstance.Member1 = value1
someInstance.Member2 = value2
'...
```

- Anonymous Types *(Option Infer must be On)*

```
Dim anonymousInstance = New With {
    .Member1 = value1,
    .Member2 = value2
    '...
}
```

Although similar `anonymousInstance` doesn't have same type as `someInstance`

Member name must be unique in the anonymous type, and can be taken from a variable or another object member name

```
Dim anonymousInstance = New With {
    value1,
    value2,
    foo.value3
    '...
}
' usage : anonymousInstance.value1 or anonymousInstance.value3
```

Each member can be preceded by the `Key` keyword. Those members will be **`ReadOnly`** properties, those without will be read/write properties

```
Dim anonymousInstance = New With {
    Key value1,
    .Member2 = value2,
    Key .Member3 = value3
    '...
}
```

Two anonymous instance defined with the same members (name, type, presence of `Key` and order) will have the same anonymous type.

```
Dim anon1 = New With { Key .Value = 10 }
Dim anon2 = New With { Key .Value = 20 }

anon1.GetType Is anon2.GetType ' True
```

Anonymous types are structurally equatable. Two instance of the same anonymous types having at least one `Key` property with the same `Key` values will be equal. You have to use `Equals` method to test it, using `=` won't compile and `Is` will compare the object reference.

```
Dim anon1 = New With { Key .Name = "Foo", Key .Age = 10, .Salary = 0 }
Dim anon2 = New With { Key .Name = "Bar", Key .Age = 20, .Salary = 0 }
Dim anon3 = New With { Key .Name = "Foo", Key .Age = 10, .Salary = 10000 }

anon1.Equals(anon2) ' False
anon1.Equals(anon3) ' True although non-Key Salary isn't the same
```

Both Named and Anonymous types initializer can be nested and mixed

```
Dim anonymousInstance = New With {
    value,
    Key .someInstance = New SomeClass(argument) With {
        .Member1 = value1,
        .Member2 = value2
        '...
    }
    '...
}
```

Section 3.6: Collection Initializer

- Arrays

```
Dim names = {"Foo", "Bar"} ' Inferred as String()
Dim numbers = {1, 5, 42} ' Inferred as Integer()
```

- Containers (`List(Of T)`, `Dictionary(Of TKey, TValue)`, etc.)

```
Dim names As New List(Of String) From {
    "Foo",
    "Bar"
    '...
}

Dim indexedDays As New Dictionary(Of Integer, String) From {
    {0, "Sun"},
    {1, "Mon"}
    '...
}
```

Is equivalent to

```
Dim indexedDays As New Dictionary(Of Integer, String)
indexedDays.Add(0, "Sun")
indexedDays.Add(1, "Mon")
'...
```

Items can be the result of a constructor, a method call, a property access. It can also be mixed with Object initializer.

```
Dim someList As New List(Of SomeClass) From {
    New SomeClass(argument),
```

```
    New SomeClass With { .Member = value },
    otherClass.PropertyReturningSomeClass,
    FunctionReturningSomeClass(arguments)
    '...
}
```

It is not possible to use Object initializer syntax **AND** collection initializer syntax for the same object at the same time. For example, these **won't** work

```
Dim numbers As New List(Of Integer) With {.Capacity = 10} _
                                   From { 1, 5, 42 }

Dim numbers As New List(Of Integer) From {
    .Capacity = 10,
    1, 5, 42
}

Dim numbers As New List(Of Integer) With {
    .Capacity = 10,
    1, 5, 42
}
```

- Custom Type

We can also allow collection initializer syntax by providing for a custom type.
It must implement `IEnumerable` and have an accessible and compatible by overload rules `Add` method (instance, Shared or even extension method)

Contrived example :

```
Class Person
    Implements IEnumerable(Of Person) ' Inherits from IEnumerable

    Private ReadOnly relationships As List(Of Person)

    Public Sub New(name As String)
        relationships = New List(Of Person)
    End Sub

    Public Sub Add(relationName As String)
        relationships.Add(New Person(relationName))
    End Sub

    Public Iterator Function GetEnumerator() As IEnumerator(Of Person) _
        Implements IEnumerable(Of Person).GetEnumerator

        For Each relation In relationships
            Yield relation
        Next
    End Function

    Private Function IEnumerable_GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        Return GetEnumerator()
    End Function
End Class

' Usage
```

```
Dim somePerson As New Person("name") From {
    "FriendName",
    "CoWorkerName"
    '...
}
```

If we wanted to add Person object to a `List(Of Person)` by just putting the name in the collection initializer (but we can't modify the List(Of Person) class) we can use an Extension method

```
' Inside a Module
<Runtime.CompilerServices.Extension>
Sub Add(target As List(Of Person), name As String)
    target.Add(New Person(name))
End Sub

' Usage
Dim people As New List(Of Person) From {
    "Name1", ' no need to create Person object here
    "Name2"
}
```

Section 3.7: Writing a function

A function is a block of code that will be called several times during the execution. Instead of writing the same piece of code again and again, one can write this code inside a function and call that function whenever it is needed.

A function :

- Must be declared in a *class* or a *module*
- Returns a value (specified by the return type)
- Has a *modifier*
- Can take parameters to do its processing

```
Private Function AddNumbers(X As Integer, Y As Integer) As Integer
    Return X + Y
End Function
```

A Function Name, could be used as the return statement

```
Function sealBarTypeValidation() as Boolean
    Dim err As Boolean = False

    If rbSealBarType.SelectedValue = "" Then
        err = True
    End If

    Return err
End Function
```

is just the same as

```
Function sealBarTypeValidation() as Boolean
    sealBarTypeValidation = False

    If rbSealBarType.SelectedValue = "" Then
        sealBarTypeValidation = True
    End If
```

```
End Function
```

Chapter 4: Operators

Section 4.1: String Concatenation

String concatenation is when you combine two or more strings into a single string variable.

String concatenation is performed with the `&` symbol.

```
Dim one As String = "Hello "
Dim two As String = "there"
Dim result As String = one & two
```

Non-string values will be converted to string when using `&`.

```
Dim result as String = "2" & 10 ' result  = "210"
```

Always use `&` (ampersand) to perform string concatenation.

DON'T DO THIS
While it is possible, in the *simplest* of cases, to use the `+` symbol to do string concatenation, you should never do this. If one side of the plus symbol is not a string, when Option strict is off, the behavior becomes non-intuitive, when Option strict is on it will produce a compiler error. Consider:

```
Dim value = "2" + 10     ' result = 12  (data type Double)
Dim value = "2" + "10"   ' result = "210"  (data type String)
Dim value = "2g" + 10    ' runtime error
```

The problem here is that if the `+` operator sees any operand that is a numeric type, it will presume that the programmer wanted to perform an arithmetic operation and attempt to cast the other operand to the equivalent numeric type. In cases where the other operand is a string that contains a number (for example, "10"), the string is *converted to a number* and then *arithmetically* added to the other operand. If the other operand cannot be converted to a number (for example, "2g"), the operation will crash due to a data conversion error. The `+` operator will only perform string concatenation if *both* operands are of `String` type.

The `&` operator, however, is designed for string concatenation and will cast non-string types to strings.

Section 4.2: Math

If you have the following variables

```
Dim leftValue As Integer = 5
Dim rightValue As Integer = 2
Dim value As Integer = 0
```

Addition Performed by the plus sign `+`.

```
value  = leftValue + rightValue

'Output the following:
'7
```

Subtraction Performed by the minus sign `-`.

```
value = leftValue - rightValue
```


www.dbooks.org

```
'Output the following:
'3
```

Multiplication Performed by the star symbol `*`.

```
value = leftValue * rightValue

'Output the following:
'10
```

Division Performed by the forward slash symbol `/`.

```
value = leftValue / rightValue

'Output the following:
'2.5
```

Integer Division Performed by the backslash symbol `\`.

```
value = leftValue \ rightValue

'Output the following:
'2
```

Modulus Performed by the `Mod` keyword.

```
value = leftValue Mod rightValue

'Output the following:
'1
```

Raise to a Power of Performed by the `^` symbol.

```
value = leftValue ^ rightValue

'Output the following:
'25
```

Section 4.3: Assignment

There is a single assignment operator in VB.

- The equal sign `=` is used both for equality comparison and assignment.
  `Dim value = 5`

Notes
Watch out for assignment vs. equality comparison.

```
Dim result = leftValue = rightValue
```

In this example you can see the equal sign being used as both a comparison operator and an assignment operator, unlike other languages. In this case, `result` will be of type `Boolean` and will contain the value of the equality comparison between `leftValue` and `rightValue`.

Related: Using Option Strict On to declare variables properly

Section 4.4: Comparison

Comparison operators compare two values and return to you a boolean (`True` or `False`) as the result.

Equality

- The equal sign `=` is used both for equality comparison and assignment.
  `If leftValue = rightValue Then ...`

Inequality

- The left angle bracket nest to the right angle bracket `<>` performs an unequal comparison.
  `If leftValue <> rightValue Then ...`

Greater Than

- The left angle bracket `<` performs a greater than comparison.
  `If leftValue < rightValue Then ...`

Greater Than Or Equal

- The equal sign nest to the left angle bracket `=>` performs a greater than or equals comparison.
  `If leftValue =< rightValue Then ...`

Less Than

- The right angle bracket `>` performs a less than comparison.
  `If leftValue > rightValue Then ...`

Less Than Or Equal

- The equal sign nest to the right angle bracket `=>` performs a greater than or equals comparison.
  `If leftValue => rightValue Then ...`

Like

- The `Like` operator tests the equality of a string and a search pattern.
- The `Like` operator relies on the Option Compare Statement
- The following table lists the available patterns. Source: https://msdn.microsoft.com/en-us/library/swf8kaxw.aspx (Remarks section)

Characters in the *Pattern*	Matches in the *String*
?	Any single character
*	Zero or more characters
#	Any single digit (0 - 9)
[charlist]	Any single character in *charlist*
[!charlist]	Any single character not in *charlist*

- See further info on MSDN in the remarks section.
  `If string Like pattern Then ...`

Section 4.5: Bitwise

These are the bitwise operators in VB.NET : And, Or, Xor, Not

Example of And bitwise operation

```
Dim a as Integer
a = 3 And 5
```

The value of a will be 1. The result is obtained after comparing 3 and 5 in binary for. 3 in binary form is 011 and 5 in binary form is 101. The And operator places 1 if both bits are 1. If any of the bits are 0 then the value will be 0

```
3 And 5 will be  011
                 101
                 ---
                 001
```

So the binary result is 001 and when that is converted to decimal, the answer will be 1.

Or operator places 1 if both or one bit is 1

```
   3 Or 5 will be 011
                  101
                  ---
                  111
```

Xor operator places 1 if only one of the bit is 1 (not both)

```
   3 Xor 5 will be  011
                    101
                    ---
                    110
```

Not operator reverts the bits including sign

```
Not 5 will be - 010
```

Chapter 5: Conditions

Section 5.1: If operator

Version ≥ 9.0

```
If(condition > value, "True", "False")
```

We can use the **If** operator instead of **If...Then...Else..End If** statement blocks.

Consider the following example:

```
If 10 > 9 Then
     MsgBox("True")
Else
     MsgBox("False")
End If
```

is the same as

```
MsgBox(If(10 > 9, "True", "False"))
```

`If()` uses *short-circuit* evaluation, which means that it will only evaluate the arguments it uses. If the condition is false (or a `Nullable` that is `Nothing`), the first alternative will not be evaluated at all, and none of its side effects will be observed. This is effectively the same as C#'s ternary operator in the form of `condition?a:b`.

This is especially useful in avoiding exceptions:

```
Dim z As Integer = If(x = 0, 0, y/x)
```

We all know that dividing by zero will throw an exception, but `If()` here guards against this by short-circuiting to only the expression that the condition has already ensured is valid.

Another example:

```
Dim varDate as DateTime = If(varString <> "N/A", Convert.ToDateTime(varString), Now.Date)
```

If `varString <> "N/A"` evaluates to `False`, it will assign `varDate`'s value as `Now.Date` without evaluating the first expression.

Version < 9.0

Older versions of VB do not have the `If()` operator and have to make do with the `IIf()` built-in function. As it's a function, not an operator, it does *not* short-circuit; all expressions are evaluated, with all possible side-effects, including performance penalties, changing state, and throwing exceptions. (Both of the above examples that avoid exceptions would throw if converted to `IIf`.) If any of these side effects present a problem, there's no way to use an inline conditional; instead, rely on `If..Then` blocks as usual.

Section 5.2: IF...Then...Else

```
Dim count As Integer = 0
Dim message As String

If count = 0 Then
```


www.dbooks.org

```
    message = "There are no items."
ElseIf count = 1 Then
    message = "There is 1 item."
Else
    message = "There are " & count & " items."
End If
```

Chapter 6: Short-Circuiting Operators (AndAlso - OrElse)

Parameter	Details
result	Required. Any Boolean expression. The result is the Boolean result of comparison of the two expressions.
expression1	Required. Any Boolean expression.
expression2	Required. Any Boolean expression.

Section 6.1: OrElse Usage

```
' The OrElse operator is the homologous of AndAlso. It lets us perform a boolean
' comparison evaluating the second condition only if the first one is False

If testFunction(5) = True OrElse otherFunction(4) = True Then
    ' If testFunction(5) is True, otherFunction(4) is not called.
    ' Insert code to be executed.
End If
```

Section 6.2: AndAlso Usage

```
' Sometimes we don't need to evaluate all the conditions in an if statement's boolean check.

' Let's suppose we have a list of strings:

Dim MyCollection as List(Of String) = New List(of String)()

' We want to evaluate the first value inside our list:

If MyCollection.Count > 0 And MyCollection(0).Equals("Somevalue")
    Console.WriteLine("Yes, I've found Somevalue in the collection!")
End If

' If MyCollection is empty, an exception will be thrown at runtime.
' This because it evaluates both first and second condition of the
' if statement regardless of the outcome of the first condition.

' Now let's apply the AndAlso operator

If MyCollection.Count > 0 AndAlso MyCollection(0).Equals("Somevalue")
    Console.WriteLine("Yes, I've found Somevalue in the collection!")
End If

' This won't throw any exception because the compiler evaluates just the first condition.
' If the first condition returns False, the second expression isn't evaluated at all.
```

Section 6.3: Avoiding NullReferenceException

Version ≥ 7.0

OrElse

```
Sub Main()
    Dim elements As List(Of Integer) = Nothing

    Dim average As Double = AverageElementsOrElse(elements)
    Console.WriteLine(average) ' Writes 0 to Console
```


www.dbooks.org

```
    Try
        'Throws ArgumentNullException
        average = AverageElementsOr(elements)
    Catch ex As ArgumentNullException
        Console.WriteLine(ex.Message)
    End Try
End Sub

Public Function AverageElementsOrElse(ByVal elements As IEnumerable(Of Integer)) As Double
    ' elements.Count is not called if elements is Nothing so it cannot crash
    If (elements Is Nothing OrElse elements.Count = 0) Then
        Return 0
    Else
        Return elements.Average()
    End If
End Function

Public Function AverageElementsOr(ByVal elements As IEnumerable(Of Integer)) As Double
    ' elements.Count is always called so it can crash if elements is Nothing
    If (elements Is Nothing Or elements.Count = 0) Then
        Return 0
    Else
        Return elements.Average()
    End If
End Function
```

Version ≥ 7.0

AndAlso

```
Sub Main()
    Dim elements As List(Of Integer) = Nothing

    Dim average As Double = AverageElementsAndAlso(elements)
    Console.WriteLine(average) ' Writes 0 to Console

    Try
        'Throws ArgumentNullException
        average = AverageElementsAnd(elements)
    Catch ex As ArgumentNullException
        Console.WriteLine(ex.Message)
    End Try
End Sub

Public Function AverageElementsAndAlso(ByVal elements As IEnumerable(Of Integer)) As Double
    ' elements.Count is not called if elements is Nothing so it cannot crash
    If (Not elements Is Nothing AndAlso elements.Count > 0) Then
        Return elements.Average()
    Else
        Return 0
    End If
End Function

Public Function AverageElementsAnd(ByVal elements As IEnumerable(Of Integer)) As Double
    ' elements.Count is always called so it can crash if elements is Nothing
    If (Not elements Is Nothing And elements.Count > 0) Then
        Return elements.Average()
    Else
        Return 0
    End If
End Function
```

Version ≥ 14.0

Visual Basic 14.0 introduced the null conditional operator, allowing to rewrite the functions in a cleaner way,

mimicking the behavior of the `AndAlso` version of the example.

www.dbooks.org

Chapter 7: Date

Section 7.1: Converting (Parsing) a String to a Date

If you know the format of the string you are converting (parsing) you should use `DateTime.ParseExact`

```
Dim dateString As String = "12.07.2003"
Dim dateFormat As String = "dd.MM.yyyy"
Dim dateValue As Date

dateValue = DateTime.ParseExact(dateString, dateFormat, Globalization.CultureInfo.InvariantCulture)
```

If you are not certain for the format of the string, you can use `DateTime.TryParseExact` and test the result to see if parsed or not:

```
Dim dateString As String = "23-09-2013"
Dim dateFormat As String = "dd-MM-yyyy"
Dim dateValue As Date

If DateTime.TryParseExact(dateString, dateFormat, Globalization.CultureInfo.InvariantCulture,
DateTimeStyles.None, dateValue) Then
    'the parse worked and the dateValue variable now holds the datetime that was parsed as it is
passing in ByRef
Else
    'the parse failed
End If
```

Section 7.2: Converting a Date To A String

Simply use the `.ToString` overload of a `DateTime` object to get the format you require:

```
Dim dateValue As DateTime = New DateTime(2001, 03, 06)
Dim dateString As String = dateValue.ToString("yyyy-MM-dd") '2001-03-06
```

Chapter 8: Array

Section 8.1: Array definition

```
Dim array(9) As Integer ' Defines an array variable with 10 Integer elements (0-9).

Dim array = New Integer(10) {} ' Defines an array variable with 11 Integer elements (0-10)
                               'using New.

Dim array As Integer() = {1, 2, 3, 4} ' Defines an Integer array variable and populate it
                                      'using an array literal. Populates the array with
                                      '4 elements.

ReDim Preserve array(10) ' Redefines the size of an existing array variable preserving any
                         'existing values in the array. The array will now have 11 Integer
                         'elements (0-10).

ReDim array(10) ' Redefines the size of an existing array variable discarding any
                'existing values in the array. The array will now have 11 Integer
                'elements (0-10).
```

Zero-Based

All arrays in VB.NET are zero-based. In other words, the index of the first item (the lower bound) in a VB.NET array is always `0`. Older versions of VB, such as VB6 and VBA, were one-based by default, but they provided a way to override the default bounds. In those earlier versions of VB, the lower and upper bounds could be explicitly stated (e.g. `Dim array(5 To 10)`. In VB.NET, in order to maintain compatibility with other .NET languages, that flexibility was removed and the lower bound of `0` is now always enforced. However, the `To` syntax can still be used in VB.NET, which may make the range more explicitly clear. For instance, the following examples are all equivalent to the ones listed above:

```
Dim array(0 To 9) As Integer

Dim array = New Integer(0 To 10) {}

ReDim Preserve array(0 To 10)

ReDim array(0 To 10)
```

Nested Array Declarations

```
Dim myArray = {{1, 2}, {3, 4}}
```

Section 8.2: Null Array Variables

Since arrays are reference types, an array variable can be null. To declare a null array variable, you must declare it without a size:

```
Dim array() As Integer
```

Or

```
Dim array As Integer()
```


www.dbooks.org

To check if an array is null, test to see if it `Is Nothing`:

```
Dim array() As Integer
If array Is Nothing Then
    array = {1, 2, 3}
End If
```

To set an existing array variable to null, simply set it to `Nothing`:

```
Dim array() As Integer = {1, 2, 3}
array = Nothing
Console.WriteLine(array(0))  ' Throws a NullReferenceException
```

Or use `Erase`, which does the same thing:

```
Dim array() As Integer = {1, 2, 3}
Erase array
Console.WriteLine(array(0))  ' Throws a NullReferenceException
```

Section 8.3: Array initialization

```
Dim array() As Integer = {2, 0, 1, 6}                  ''Initialize an array of four Integers.
Dim strings() As String = {"this", "is", "an", "array"} ''Initialize an array of four Strings.
Dim floats() As Single = {56.2, 55.633, 1.2, 5.7743, 22.345}
              ''Initialize an array of five Singles, which are the same as floats in C#.
Dim miscellaneous() as Object = { New Object(), "Hello", New List(of String) }
              ''Initialize an array of three references to any reference type objects
              ''and point them to objects of three different types.
```

Section 8.4: Declare a single-dimension array and set array element values

```
Dim array = New Integer() {1, 2, 3, 4}
```

or

```
Dim array As Int32() = {1, 2, 3, 4}
```

Section 8.5: Jagged Array Initialization

Note the parenthesis to distinguish between a jagged array and a multidimensional array SubArrays can be of different length

```
Dim jaggedArray()() As Integer = { ({1, 2, 3}), ({4, 5, 6}), ({7}) }
' jaggedArray(0) is {1, 2, 3} and so jaggedArray(0)(0) is 1
' jaggedArray(1) is {4, 5, 6} and so jaggedArray(1)(0) is 4
' jaggedArray(2) is {7} and so jaggedArray(2)(0) is 7
```

Section 8.6: Non-zero lower bounds

With `Option Strict On`, although the .NET Framework allows the creation of single dimension arrays with non-zero lower bounds they are not "vectors" and so not compatible with VB.NET typed arrays. This means they can only be seen as `Array` and so cannot use normal array `(index)` references.

```
Dim a As Array = Array.CreateInstance(GetType(Integer), {4}, {-1})
For y = LBound(a) To UBound(a)
    a.SetValue(y * y, y)
Next
For y = LBound(a) To UBound(a)
    Console.WriteLine($"{y}: {a.GetValue(y)}")
Next
```

As well as by using `Option Strict Off`, you can get the `(index)` syntax back by treating the array as an `IList`, but then it's not an array, so you can't use `LBound` and `UBound` on that variable name (and you're still not avoiding boxing):

```
Dim nsz As IList = a
For y = LBound(a) To UBound(a)
    nsz(y) = 2 - CInt(nsz(y))
Next
For y = LBound(a) To UBound(a)
    Console.WriteLine($"{y}: {nsz(y)}")
Next
```

Multi-dimensional non-zero lower bounded arrays *are* compatible with VB.NET multi-dimensional typed arrays:

```
Dim nza(,) As Integer = DirectCast(Array.CreateInstance(GetType(Integer),
                                    {4, 3}, {1, -1}), Integer(,))
For y = LBound(nza) To UBound(nza)
    For w = LBound(nza, 2) To UBound(nza, 2)
        nza(y, w) = -y * w + nza(UBound(nza) - y + LBound(nza),
                                 UBound(nza, 2) - w + LBound(nza, 2))
    Next
Next
For y = LBound(nza) To UBound(nza)
    Dim ly = y
    Console.WriteLine(String.Join(" ",
        Enumerable.Repeat(ly & ":", 1).Concat(
            Enumerable.Range(LBound(nza, 2), UBound(nza, 2) - LBound(nza, 2) + 1) _
            .Select(Function(w) CStr(nza(ly, w))))))
Next
```

MSDN reference: Array.CreateInstance

Section 8.7: Referencing Same Array from Two Variables

Since arrays are reference types, it is possible to have multiple variables pointing to the same array object.

```
Dim array1() As Integer = {1, 2, 3}
Dim array2() As Integer = array1
array1(0) = 4
Console.WriteLine(String.Join(", ", array2))  ' Writes "4, 2, 3"
```

Section 8.8: Multidimensional Array initialization

```
Dim array2D(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
' array2D(0, 0) is 1 ; array2D(0, 1) is 2 ; array2D(1, 0) is 4

Dim array3D(,,) As Integer = {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}}}
' array3D(0, 0, 0) is 1 ; array3D(0, 0, 1) is 2
' array3D(0, 1, 0) is 4 ; array3D(1, 0, 0) is 7
```

Chapter 9: Lists

Section 9.1: Add items to a List

```
Dim aList as New List(Of Integer)
aList.Add(1)
aList.Add(10)
aList.Add(1001)
```

To add more than one item at a time use **AddRange**. Always adds to the end of the list

```
Dim blist as New List(of Integer)
blist.AddRange(alist)


Dim aList as New List(of String)
alist.AddRange({"one", "two", "three"})
```

In order to add items to the middle of the list use **Insert**

Insert will place the item at the index, and renumber the remaining items

```
Dim aList as New List(Of String)
aList.Add("one")
aList.Add("three")
alist(0) = "one"
alist(1) = "three"
alist.Insert(1,"two")
```

New Output:

```
alist(0) = "one"
alist(1) = "two"
alist(2) = "three"
```

Section 9.2: Check if item exists in a List

```
    Sub Main()
        Dim People = New List(Of String)({"Bob Barker", "Ricky Bobby", "Jeff Bridges"})
        Console.WriteLine(People.Contains("Rick James"))
        Console.WriteLine(People.Contains("Ricky Bobby"))
        Console.WriteLine(People.Contains("Barker"))
        Console.Read
    End Sub
```

Produces the following output:

```
False
True
False
```

Section 9.3: Loop through items in list

```
Dim aList as New List(Of String)
```

```
aList.Add("one")
aList.Add("two")
aList.Add("three")

For Each str As String in aList
    System.Console.WriteLine(str)
Next
```

Produces the following output:

```
one
two
three
```

Another option, would be to loop through using the index of each element:

```
Dim aList as New List(Of String)
aList.Add("one")
aList.Add("two")
aList.Add("three")

For i = 0 to aList.Count - 1 'We use "- 1" because a list uses 0 based indexing.
    System.Console.WriteLine(aList(i))
Next
```

Section 9.4: Create a List

Lists can populated with any data type as necessary, with the format

```
Dim aList as New List(Of Type)
```

For example:

Create a new, empty list of Strings

```
Dim aList As New List(Of String)
```

Create a new list of strings, and populate with some data

VB.NET 2005/2008:

```
Dim aList as New List(Of String)(New String() {"one", "two", "three"})
```

VB.NET 2010:

```
Dim aList as New List(Of String) From {"one", "two", "three"}
```

--

VB.NET 2015:

```
Dim aList as New List(Of String)(New String() {"one", "two", "three"})
```

NOTE:

If you are receiving the following when the code is ran:

www.dbooks.org

> Object reference not set to an instance of an object.

Make sure you either declare as `New` i.e. `Dim aList as New List(Of String)` or if declaring without the `New`, make sure you set the list to a new list - `Dim aList as List(Of String) = New List(Of String)`

Section 9.5: Remove items from a List

```
Dim aList As New List(Of String)
aList.Add("Hello")
aList.Add("Delete Me!")
aList.Add("World")

'Remove the item from the list at index 1
aList.RemoveAt(1)

'Remove a range of items from a list, starting at index 0, for a count of 1)
'This will remove index 0, and 1!
aList.RemoveRange(0, 1)

'Clear the entire list
alist.Clear()
```

Section 9.6: Retrieve items from a List

```
Dim aList as New List(Of String)
aList.Add("Hello, World")
aList.Add("Test")

Dim output As String = aList(0)
```

output:

> Hello, World

If you do not know the index of the item or only know part of the string then use the **Find** or **FindAll** method

```
Dim aList as New List(Of String)
aList.Add("Hello, World")
aList.Add("Test")

Dim output As String = aList.Find(Function(x) x.StartWith("Hello"))
```

output:

> Hello, World

The **FindAll** method returns a new List (of String)

```
Dim aList as New List(Of String)
aList.Add("Hello, Test")
aList.Add("Hello, World")
aList.Add("Test")
```

```
Dim output As String = aList.FindAll(Function(x) x.Contains("Test"))
```

> output(0) = "Hello, Test"

> output(1) = "Test"

www.dbooks.org

Chapter 10: Enum

Section 10.1: GetNames()

Returns the names of constants in the specified Enum as a string array:

```
Module Module1

  Enum Size
    Small
    Medium
    Large
  End Enum

  Sub Main()
    Dim sizes = [Enum].GetNames(GetType(Size))

    For Each size In sizes
      Console.WriteLine(size)
      Next
  End Sub

End Module
```

Output:

> Small
>
> Medium
>
> Large

Section 10.2: HasFlag()

The `HasFlag()` method can be used to check if a flag is set.

```
Module Module1

    <Flags>
    Enum Material
        Wood = 1
        Plastic = 2
        Metal = 4
        Stone = 8
    End Enum

    Sub Main()
        Dim houseMaterials As Material = Material.Wood Or Material.Stone

        If houseMaterials.HasFlag(Material.Stone) Then
            Console.WriteLine("the house is made of stone")
        Else
            Console.WriteLine("the house is not made of stone")
        End If
    End Sub
```

```
End Module
```

For more information about the Flags-attribute and how it should be used see the official Microsoft documentation.

Section 10.3: Enum definition

An enum is a set of logically related constants.

```
Enum Size
    Small
    Medium
    Large
End Enum

Public Sub Order(shirtSize As Size)
    Select Case shirtSize
        Case Size.Small
            ' ...
        Case Size.Medium
            ' ...
        Case Size.Large
            ' ...
    End Select
End Sub
```

Section 10.4: Member initialization

Each of the enum members may be initialized with a value. If a value is not specified for a member, by default it's initialized to 0 (if it's the first member in the member list) or to a value greater by 1 than the value of the preceding member.

```
Module Module1

    Enum Size
        Small
        Medium = 3
        Large
    End Enum

    Sub Main()
        Console.WriteLine(Size.Small)    ' prints 0
        Console.WriteLine(Size.Medium)   ' prints 3
        Console.WriteLine(Size.Large)    ' prints 4

        ' Waits until user presses any key
        Console.ReadKey()
    End Sub

End Module
```

Section 10.5: The Flags attribute

With the **<Flags>** attribute, the enum becomes a set of flags. This attribute enables assigning multiple values to an enum variable. The members of a flags enum should be initialized with powers of 2 (1, 2, 4, 8...).

```
Module Module1

    <Flags>
```


www.dbooks.org

```
    Enum Material
        Wood = 1
        Plastic = 2
        Metal = 4
        Stone = 8
    End Enum

    Sub Main()
        Dim houseMaterials As Material = Material.Wood Or Material.Stone
        Dim carMaterials as Material = Material.Plastic Or Material.Metal
        Dim knifeMaterials as Material = Material.Metal

        Console.WriteLine(houseMaterials.ToString()) 'Prints "Wood, Stone"
        Console.WriteLine(CType(carMaterials, Integer)) 'Prints 6
    End Sub

End Module
```

Section 10.6: GetValues()

' This method is useful for iterating Enum values '

```
Enum Animal
    Dog = 1
    Cat = 2
    Frog = 4
End Enum

Dim Animals = [Enum].GetValues(GetType(Animal))

For Each animal in Animals
    Console.WriteLine(animal)
Next
```

Prints:

> 1
>
> 2
>
> 4

Section 10.7: String parsing

An Enum instance can be created by parsing a string representation of the Enum.

```
Module Module1

    Enum Size
        Small
        Medium
        Large
    End Enum

    Sub Main()
        Dim shirtSize As Size =  DirectCast([Enum].Parse(GetType(Size), "Medium"), Size)
```

```
        ' Prints 'Medium'
        Console.WriteLine(shirtSize.ToString())

        ' Waits until user presses any key
        Console.ReadKey()
    End Sub

End Module
```

See also: Parse a string to an Enum value in VB.NET

Section 10.8: ToString()

The `ToString` method on an enum returns the string name of the enumeration. For instance:

```
Module Module1
    Enum Size
        Small
        Medium
        Large
    End Enum

    Sub Main()
        Dim shirtSize As Size = Size.Medium
        Dim output As String = shirtSize.ToString()
        Console.WriteLine(output)  ' Writes "Medium"
    End Sub
End Module
```

If, however, the string representation of the actual integer value of the enum is desired, you can cast the enum to an `Integer` and then call `ToString`:

```
Dim shirtSize As Size = Size.Medium
Dim output As String = CInt(shirtSize).ToString()
Console.WriteLine(output)  ' Writes "1"
```

Section 10.9: Determine whether a Enum has FlagsAttribute specified or not

The next example can be used to determine whether a enumeration has the **FlagsAttribute** specified. The methodology used is based on **Reflection**.

This example will give a **`True`** result:

```
Dim enu As [Enum] = New FileAttributes()
Dim hasFlags As Boolean = enu.GetType().GetCustomAttributes(GetType(FlagsAttribute),
inherit:=False).Any()
Console.WriteLine("{0} Enum has FlagsAttribute?: {1}", enu.GetType().Name, hasFlags)
```

This example will give a **`False`** result:

```
Dim enu As [Enum] = New ConsoleColor()
Dim hasFlags As Boolean = enu.GetType().GetCustomAttributes(GetType(FlagsAttribute),
inherit:=False).Any()
Console.WriteLine("{0} Enum has FlagsAttribute?: {1}", enu.GetType().Name, hasFlags)
```

We can design a generic usage extension method like this one:

```
<DebuggerStepThrough>
<Extension>
<EditorBrowsable(EditorBrowsableState.Always)>
Public Function HasFlagsAttribute(ByVal sender As [Enum]) As Boolean
    Return sender.GetType().GetCustomAttributes(GetType(FlagsAttribute), inherit:=False).Any()
End Function
```

Usage Example:

```
Dim result As Boolean = (New FileAttributes).HasFlagsAttribute()
```

Section 10.10: For-each flag (flag iteration)

In some very specific scenarios we would feel the need to perform a specific action for each flag of the source enumeration.

We can write a simple *Generic* extension method to realize this task.

```
<DebuggerStepThrough>
<Extension>
<EditorBrowsable(EditorBrowsableState.Always)>
Public Sub ForEachFlag(Of T)(ByVal sender As [Enum],
                             ByVal action As Action(Of T))

    For Each flag As T In sender.Flags(Of T)
        action.Invoke(flag)
    Next flag

End Sub
```

Usage Example:

```
Dim flags As FileAttributes = (FileAttributes.ReadOnly Or FileAttributes.Hidden)

flags.ForEachFlag(Of FileAttributes)(
      Sub(ByVal x As FileAttributes)
          Console.WriteLine(x.ToString())
      End Sub)
```

Section 10.11: Determine the amount of flags in a flag combination

The next example is intended to count the amount of flags in the specified flag combination.

The example is provided as a extension method:

```
<DebuggerStepThrough>
<Extension>
<EditorBrowsable(EditorBrowsableState.Always)>
Public Function CountFlags(ByVal sender As [Enum]) As Integer
    Return sender.ToString().Split(","c).Count()
End Function
```

Usage Example:

```
Dim flags As FileAttributes = (FileAttributes.Archive Or FileAttributes.Compressed)
Dim count As Integer = flags.CountFlags()
```

```
Console.WriteLine(count)
```

Section 10.12: Find the nearest value in a Enum

The next code illustrates how to find the nearest value of a **Enum**.

First we define this **Enum** that will serve to specify search criteria (search direction)

```
Public Enum EnumFindDirection As Integer
    Nearest = 0
    Less = 1
    LessOrEqual = 2
    Greater = 3
    GreaterOrEqual = 4
End Enum
```

And now we implement the search algorithm:

```
<DebuggerStepThrough>
Public Shared Function FindNearestEnumValue(Of T)(ByVal value As Long,
                                                  ByVal direction As EnumFindDirection) As T

    Select Case direction

        Case EnumFindDirection.Nearest
            Return (From enumValue As T In [Enum].GetValues(GetType(T)).Cast(Of T)()
                    Order By Math.Abs(value - Convert.ToInt64(enumValue))
                    ).FirstOrDefault

        Case EnumFindDirection.Less
            If value < Convert.ToInt64([Enum].GetValues(GetType(T)).Cast(Of T).First) Then
                Return [Enum].GetValues(GetType(T)).Cast(Of T).FirstOrDefault

            Else
                Return (From enumValue As T In [Enum].GetValues(GetType(T)).Cast(Of T)()
                        Where Convert.ToInt64(enumValue) < value
                        ).LastOrDefault
            End If

        Case EnumFindDirection.LessOrEqual
            If value < Convert.ToInt64([Enum].GetValues(GetType(T)).Cast(Of T).First) Then
                Return [Enum].GetValues(GetType(T)).Cast(Of T).FirstOrDefault

            Else
                Return (From enumValue As T In [Enum].GetValues(GetType(T)).Cast(Of T)()
                        Where Convert.ToInt64(enumValue) <= value
                        ).LastOrDefault
            End If

        Case EnumFindDirection.Greater
            If value > Convert.ToInt64([Enum].GetValues(GetType(T)).Cast(Of T).Last) Then
                Return [Enum].GetValues(GetType(T)).Cast(Of T).LastOrDefault

            Else
                Return (From enumValue As T In [Enum].GetValues(GetType(T)).Cast(Of T)()
                        Where Convert.ToInt64(enumValue) > value
                        ).FirstOrDefault
            End If

        Case EnumFindDirection.GreaterOrEqual
```


www.dbooks.org

```
            If value > Convert.ToInt64([Enum].GetValues(GetType(T)).Cast(Of T).Last) Then
                Return [Enum].GetValues(GetType(T)).Cast(Of T).LastOrDefault

            Else
                Return (From enumValue As T In [Enum].GetValues(GetType(T)).Cast(Of T)()
                        Where Convert.ToInt64(enumValue) >= value
                        ).FirstOrDefault
            End If

    End Select

End Function
```

Usage Example:

```
Public Enum Bitrate As Integer
    Kbps128 = 128
    Kbps192 = 192
    Kbps256 = 256
    Kbps320 = 320
End Enum

Dim nearestValue As Bitrate = FindNearestEnumValue(Of Bitrate)(224,
EnumFindDirection.GreaterOrEqual)
```

Chapter 11: Dictionaries

A dictionary represents a collection of keys and values. See MSDN Dictionary(Tkey, TValue) Class.

Section 11.1: Create a dictionary filled with values

```
Dim extensions As New Dictionary(Of String, String) _
  from { { "txt", "notepad" },
  { "bmp", "paint" },
  { "doc", "winword" } }
```

This creates a dictionary and immediately fills it with three KeyValuePairs.

You can also add new values later on by using the Add method:

```
extensions.Add("png", "paint")
```

Note that the key (the first parameter) needs to be unique in the dictionary, otherwise an Exception will be thrown.

Section 11.2: Loop through a dictionary and print all entries

Each pair in the dictionary is an instance of `KeyValuePair` with the same type parameters as the Dictionary. When you loop through the dictionary with `For Each`, each iteration will give you one of the Key-Value Pairs stored in the dictionary.

```
For Each kvp As KeyValuePair(Of String, String) In currentDictionary
  Console.WriteLine("{0}: {1}", kvp.Key, kvp.Value)
Next
```

Section 11.3: Checking for key already in dictionary - data reduction

The `ConstainsKey` method is the way to know if a key already exists in the Dictionary.

This come in handy for data reduction. In the sample below, each time we encountner a new word, we add it as a key in the dictionary, else we increment the counter for this specific word.

```
Dim dic As IDictionary(Of String, Integer) = New Dictionary(Of String, Integer)

Dim words As String() = Split(<big text source>," ", -1, CompareMethod.Binary)

For Each str As String In words
    If dic.ContainsKey(str) Then
        dic(str) += 1
    Else
        dic.Add(str, 1)
    End If
Next
```

XML reduction example : getting all the child nodes names and occurrence in an branch of an XML document

```
Dim nodes As IDictionary(Of String, Integer) = New Dictionary(Of String, Integer)
Dim xmlsrc = New XmlDocument()
xmlsrc.LoadXml(<any text stream source>)
```


www.dbooks.org

```
For Each xn As XmlNode In xmlsrc.FirstChild.ChildNodes 'selects the proper parent
    If nodes.ContainsKey(xn.Name) Then
        nodes(xn.Name) += 1
    Else
        nodes.Add(xn.Name, 1)
    End If
Next
```

Section 11.4: Getting a dictionary value

You can get the value of an entry in the dictionary using the 'Item' property:

```
Dim extensions As New Dictionary(Of String, String) From {
    { "txt", "notepad" },
    { "bmp", "paint" },
    { "doc", "winword" }
}

Dim program As String = extensions.Item("txt") 'will be "notepad"

' alternative syntax as Item is the default property (a.k.a. indexer)
Dim program As String = extensions("txt") 'will be "notepad"

' other alternative syntax using the (rare)
' dictionary member access operator (a.k.a. bang operator)
Dim program As String = extensions!txt 'will be "notepad"
```

If the key is not present in the dictionary, a KeyNotFoundException will be thrown.

Chapter 12: Looping

Section 12.1: For...Next

`For`...`Next` loop is used for repeating the same action for a finite number of times. The statements inside the following loop will be executed 11 times. The first time, `i` will have the value 0, the second time it will have the value 1, the last time it will have the value 10.

```
For i As Integer = 0 To 10
    'Execute the action
    Console.Writeline(i.ToString)
Next
```

Any integer expression can be used to parameterize the loop. It is permitted, but not required, for the control variable (in this case `i`) to also be stated after the `Next`. It is permitted for the control variable to be declared in advance, rather than within the `For` statement.

```
Dim StartIndex As Integer = 3
Dim EndIndex As Integer = 7
Dim i As Integer

For i = StartIndex To EndIndex - 1
    'Execute the action
    Console.Writeline(i.ToString)
Next i
```

Being able to define the Start and End integers allows loops to be created that directly reference other objects, such as:

```
For i = 0 to DataGridView1.Rows.Count - 1
    Console.Writeline(DataGridView1.Rows(i).Cells(0).Value.ToString)
Next
```

This would then loop through every row in `DataGridView1` and perform the action of writing the value of Column 1 to the Console. *(The -1 is because the first row of the counted rows would be 1, not 0)*

It is also possible to define how the control variable must increment.

```
For i As Integer = 1 To 10 Step 2
    Console.Writeline(i.ToString)
Next
```

This outputs:

> 1 3 5 7 9

It is also possible to decrement the control variable (count down).

```
For i As Integer = 10 To 1 Step -1
    Console.Writeline(i.ToString)
Next
```

This outputs:

www.dbooks.org

> 10 9 8 7 6 5 4 3 2 1

You should not attempt to use (read or update) the control variable outside the loop.

Section 12.2: For Each...Next loop for looping through collection of items

You can use a `For Each...Next` loop to iterate through any `IEnumerable` type. This includes arrays, lists, and anything else that may be of type IEnumerable or returns an IEnumerable.

An example of looping through a DataTable's Rows property would look like this:

```
For Each row As DataRow In DataTable1.Rows
    'Each time this loops, row will be the next item out of Rows
    'Here we print the first column's value from the row variable.
    Debug.Print(Row.Item(0))
Next
```

An important thing to note is that the collection must not be modified while in a `For Each` loop. Doing so will cause a `System.InvalidOperationException` with the message:

> Collection was modified; enumeration operation may not execute.

Section 12.3: Short Circuiting

Any loop may be terminated or continued early at any point by using the `Exit` or `Continue` statements.

Exiting

You can stop any loop by exiting early. To do this, you can use the keyword `Exit` along with the name of the loop.

Loop	Exit Statement
For	`Exit For`
For Each	`Exit For`
Do While	`Exit Do`
While	`Exit While`

Exiting a loop early is a great way to boost performance by only looping the necessary number of times to satisfy the application's needs. Below is example where the loop will exit once it finds the number 2.

```
Dim Numbers As Integer() = {1,2,3,4,5}
Dim SoughtValue As Integer = 2
Dim SoughtIndex
For Each i In Numbers
    If i = 2 Then
        SoughtIndex = i
        Exit For
    End If
Next
Debug.Print(SoughtIndex)
```

Continuing

Along with exiting early, you can also decide that you need to just move on to the next loop iteration. This is easily done by using the `Continue` statement. Just like `Exit`, it is proceeded by the loop name.

Loop	Continue Statement
For	`Continue For`
For Each	`Continue For`
Do While	`Continue Do`
While	`Continue While`

Here's an example of preventing even numbers from being added to the sum.

```
Dim Numbers As Integer() = {1,2,3,4,5}
Dim SumOdd As Integer = 0
For Each i In Numbers
    If Numbers(i) \ 2 = 0 Then Continue For
    SumOdd += Numbers(i)
Next
```

Usage Advice

There are two alternative techniques that can be used instead of using `Exit` or `Continue`.

You can declare a new Boolean variable, initializing it to one value and conditionally setting it to the other value inside the loop; you then use a conditional statement (e.g. `If`) based on that variable to avoid execution of the statements inside the loop in subsequent iterations.

```
Dim Found As Boolean = False
Dim FoundIndex As Integer
For i As Integer = 0 To N - 1
    If Not Found AndAlso A(i) = SoughtValue Then
        FoundIndex = i
        Found = True
    End If
Next
```

One of the objections to this technique is that it may be inefficient. For example, if in the above example `N` is 1000000 and the first element of the array `A` is equal to `SoughtValue`, the loop will iterate a further 999999 times without doing anything useful. However, this technique can have the advantage of greater clarity in some cases.

You can use the `GoTo` statement to jump out of the loop. Note that you cannot use `GoTo` to jump *into* a loop.

```
    Dim FoundIndex As Integer
    For i As Integer = 0 To N - 1
        If A(i) = SoughtValue Then
            FoundIndex = i
            GoTo Found
        End If
    Next
    Debug.Print("Not found")
Found:
    Debug.Print(FoundIndex)
```

This technique can sometimes be the neatest way to jump out of the loop and avoid one or more statements that are executed just after the natural end of the loop.

You should consider all of the alternatives, and use whichever one best fits your requirements, considering such

www.dbooks.org

things as efficiency, speed of writing the code, and readability (thus maintainability).

Do not be put off using `GoTo` on those occasions when it is the best alternative.

Section 12.4: While loop to iterate while some condition is true

A `While` loop starts by evaluating a condition. If it is true, the body of the loop is executed. After the body of the loop is executed, the `While` condition is evaluated again to determine whether to re-execute the body.

```
Dim iteration As Integer = 1
While iteration <= 10
  Console.Writeline(iteration.ToString() & " ")

  iteration += 1
End While
```

This outputs:

> 1 2 3 4 5 6 7 8 9 10

Warning: A `While` loop can lead to an *infinite loop*. Consider what would happen if the line of code that increments `iteration` were removed. In such a case the condition would never be True and the loop would continue indefinitely.

Section 12.5: Nested Loop

> A nested loop is a loop within a loop, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. a break within either the inner or outer loop would interrupt this process.

The Structure of a For Next nested loop is :

```
For counter1=startNumber to endNumber (Step increment)

    For counter2=startNumber to endNumber (Step increment)

        One or more VB statements

    Next counter2

Next  counter1
```

Example :

```
    For  firstCounter = 1 to 5

      Print “First Loop of ” + firstCounter

    For   secondCounter= 1 to 4

      Print “Second Loop of ” + secondCounter

    Next secondCounter
```

```
Next firstCounter
```

Section 12.6: Do...Loop

Use `Do...Loop` to repeat a block of statements `While` or `Until` a condition is true, checking the condition either at the beginning or at the end of the loop.

```
Dim x As Integer = 0
Do
    Console.Write(x & " ")
    x += 1
Loop While x < 10
```

or

```
Dim x As Integer = 0
Do While x < 10
    Console.Write(x & " ")
    x += 1
Loop
```

> 0 1 2 3 4 5 6 7 8 9

```
Dim x As Integer = 0
Do
    Console.Write(x & " ")
    x += 1
Loop Until x = 10
```

or

```
Dim x As Integer = 0
Do Until x = 10
    Console.Write(x & " ")
    x += 1
Loop
```

> 0 1 2 3 4 5 6 7 8 9

`Continue Do` can be used to skip to the next iteration of the loop:

```
Dim x As Integer = 0
Do While x < 10
    x += 1
    If x Mod 2 = 0 Then
        Continue Do
    End If
    Console.Write(x & " ")
Loop
```

> 1 3 5 7 9

You can terminate the loop with `Exit Do` - note that in this example, the lack of any condition would otherwise cause an infinite loop:

```
Dim x As Integer = 0
Do
    Console.Write(x & " ")
    x += 1
    If x = 10 Then
        Exit Do
    End If
Loop
```

> 0 1 2 3 4 5 6 7 8 9

Chapter 13: File Handling

Section 13.1: Write Data to a File

To write the contents of a string to a file:

```
Dim toWrite As String = "This will be written to the file."
System.IO.File.WriteAllText("filename.txt", toWrite)
```

`WriteAllText` will open the specified file, write the data, and then close the file. If the target file exists, it is overwritten. If the target file does not exist, it is created.

To write the contents of an array to a file:

```
Dim toWrite As String() = {"This", "Is", "A", "Test"}
System.IO.File.WriteAllLines("filename.txt", toWrite)
```

`WriteAllLines` will open the specified file, write each value of the array on a new line, and then close the file. If the target file exists, it is overwritten. If the target file does not exist, it is created.

Section 13.2: Read All Contents of a File

To read the contents to a file into a string variable:

```
Dim fileContents As String = System.IO.File.ReadAllText("filename.txt")
```

`ReadAllText` will open the specified file, read data to the end, then close the file.

To read a file, separating it into an array element for each line:

```
Dim fileLines As String() = System.IO.File.ReadAllLines("filename.txt")
```

`ReadAllLines` will open the specified file, read each line of the file into a new index in an array until the end of the file, then close the file.

Section 13.3: Write Lines Individually to a Text File using StreamWriter

```
Using sw As New System.IO.StreamWriter("path\to\file.txt")
    sw.WriteLine("Hello world")
End Using
```

The use of a `Using` block is recommended good practice when using an object that Implements `IDisposable`

www.dbooks.org

Chapter 14: File/Folder Compression

Section 14.1: Adding File Compression to your project

1. In *Solution Explorer* go to your project, right click on *References* then *Add reference*…
2. Search for Compression and select *System.IO.Compression.FileSystem* then press OK.
3. Add `Imports System.IO.Compression` to the top of your code file (before any class or module, with the other `Imports` statements).

```
Option Explicit On
Option Strict On

Imports System.IO.Compression

Public Class Foo

    ...

End Class
```

Plese note that this class (ZipArchive) is only available from .NET verison 4.5 onwards

Section 14.2: Creating zip archive from directory

```
System.IO.Compression.ZipFile.CreateFromDirectory("myfolder", "archive.zip")
```

Create archive.zip file containing files which are in `myfolder`. In example paths are relative to program working directory. You can specify absolute paths.

Section 14.3: Extracting zip archive to directory

```
System.IO.Compression.ZipFile.ExtractToDirectory("archive.zip", "myfolder")
```

Extracts archive.zip to myfolder directory. In example paths are relative to program working directory. You can specify absolute paths.

Section 14.4: Create zip archive dynamicaly

```
' Create filestream to file
Using fileStream = New IO.FileStream("archive.zip", IO.FileMode.Create)
    ' open zip archive from stream
    Using archive = New System.IO.Compression.ZipArchive(fileStream,
IO.Compression.ZipArchiveMode.Create)
        ' create file_in_archive.txt in archive
        Dim zipfile = archive.CreateEntry("file_in_archive.txt")

        ' write Hello world to file_in_archive.txt in archive
        Using sw As New IO.StreamWriter(zipfile.Open())
            sw.WriteLine("Hello world")
        End Using

    End Using
End Using
```

Chapter 15: Connection Handling

Section 15.1: Public connection property

```
Imports System.Data.OleDb

Private WithEvents _connection As OleDbConnection
Private _connectionString As String = "myConnectionString"

Public ReadOnly Property Connection As OleDbConnection
    Get
        If _connection Is Nothing Then
            _connection = New OleDbConnection(_connectionString)
            _connection.Open()
        Else
            If _connection.State <> ConnectionState.Open Then
                _connection.Open()
            End If
        End If
        Return _connection
    End Get
End Property
```


www.dbooks.org

Chapter 16: Data Access

Section 16.1: Read field from Database

```
Public Function GetUserFirstName(UserName As String) As String
    Dim Firstname As String = ""

    'Specify the SQL that you want to use including a Parameter
    Dim SQL As String = "select firstname from users where username=@UserName"

    'Provide a Data Source
    Dim DBDSN As String = "Data Source=server.address;Initial Catalog=DatabaseName;Persist Security
Info=True;User ID=UserName;Password=UserPassword"

    Dim dbConn As New SqlConnection(DBDSN)

    Dim dbCommand As New SqlCommand(SQL, dbConn)

    'Provide one or more Parameters
    dbCommand.Parameters.AddWithValue("@UserName", UserName)

    'An optional Timeout
    dbCommand.CommandTimeout = 600

    Dim reader As SqlDataReader
    Dim previousConnectionState As ConnectionState = dbConn.State
    Try
        If dbConn.State = ConnectionState.Closed Then
            dbConn.Open()
        End If
        reader = dbCommand.ExecuteReader
        Using reader
            With reader
                If .HasRows Then
                    'Read the 1st Record
                    reader.Read()
                    'Read required field/s
                    Firstname = .Item("FirstName").ToString
                End If

            End With

        End Using

    Catch
        'Handle the error here
    Finally
        If previousConnectionState = ConnectionState.Closed Then
            dbConn.Close()
        End If
        dbConn.Dispose()
        dbCommand.Dispose()

    End Try
    'Pass the data back from the function
    Return Firstname

End Function
```

Using the above function is simply:

```
    Dim UserFirstName as string=GetUserFirstName(UserName)
```

Section 16.2: Simple Function to read from Database and return as DataTable

This simple function will execute the specified Select SQL command and return the result as data set.

```
Public Function ReadFromDatabase(ByVal DBConnectionString As String, ByVal SQL As String) As
DataTable
    Dim dtReturn As New DataTable
    Try
        'Open the connection using the connection string
        Using conn As New SqlClient.SqlConnection(DBConnectionString)
            conn.Open()

            Using cmd As New SqlClient.SqlCommand()
                cmd.Connection = conn
                cmd.CommandText = SQL
                Dim da As New SqlClient.SqlDataAdapter(cmd)
                da.Fill(dtReturn)
            End Using
        End Using
    Catch ex As Exception
        'Handle the exception
    End Try


    'Return the result data set
    Return dtReturn
End Function
```

Now you can execute the above function from below codes

```
Private Sub MainFunction()
    Dim dtCustomers As New DataTable
    Dim dtEmployees As New DataTable
    Dim dtSuppliers As New DataTable


    dtCustomers = ReadFromDatabase("Server=MYDEVPC\SQLEXPRESS;Database=MyDatabase;User
Id=sa;Password=pwd22;", "Select * from [Customers]")
    dtEmployees = ReadFromDatabase("Server=MYDEVPC\SQLEXPRESS;Database=MyDatabase;User
Id=sa;Password=pwd22;", "Select * from [Employees]")
    dtSuppliers = ReadFromDatabase("Server=MYDEVPC\SQLEXPRESS;Database=MyDatabase;User
Id=sa;Password=pwd22;", "Select * from [Suppliers]")

End Sub
```

The above example expects that your SQL Express instance "SQLEXPRESS" is currently installed on "MYDEVPC" and your database "MyDatabase" contains "Customers", "Suppliers" and "Employees" tables and the "sa" user password is "pwd22". Please change these values as per your setup to get the desired results.

www.dbooks.org

Chapter 17: Type conversion

Function name	Range for Expression argument
CBool	Any valid Char or String or numeric expression
CByte	0 through 255 (unsigned); fractional parts are rounded.
CChar	Any valid Char or String expression; only first character of a String is converted; value can be 0 through 65535 (unsigned).

Section 17.1: Converting Text of The Textbox to an Integer

From MSDN

> Use the CInt function to provide conversions from any other data type to an Integer subtype. For example, CInt forces integer arithmetic when currency, single-precision, or double-precision arithmetic would normally occur.

Assuming that you have 1 button and 2 textbox. If you type on textbox1.text `5.5` and on textbox2.text `10`.

If you have this code:

```
Dim result = textbox1.text + textbox2.text
MsgBox("Result: " & result)
'It will output
5.510
```

In order to add the values of the 2 textboxes you need to convert their values to `Int` by using the `CInt(expression)`.

```
Dim result = CInt(textbox1.text) + CInt(textbox2.text)
MsgBox("Result: " & result)
'It will output
16
```

> Note: When the fractional part of a value is exactly 0.5, the CInt function rounds to the closest even number. For example, **0.5 rounds to 0**, while **1.5 rounds to 2, and 3.5 rounds to 4**. The purpose of rounding to the closest even number is to compensate for a bias that could accumulate when many numbers are added together.

Chapter 18: ByVal and ByRef keywords

Section 18.1: ByRef keyword

ByRef keyword before method parameter says that parameter will be sent in a way allowing the method to change (assign a new value) the variable underlying the parameter.

```
Class SomeClass
    Public Property Member As Integer
End Class

Module Program
    Sub Main()
        Dim someInstance As New SomeClass With {.Member = 42}

        Foo (someInstance)
        ' here someInstance is not Nothing
        ' but someInstance.Member is -42

        Bar(someInstance)
        ' here someInstance is Nothing
    End Sub

    Sub Foo(ByVal arg As SomeClass)
        arg.Member = -arg.Member ' change argument content
        arg = Nothing ' change (re-assign) argument
    End Sub

    Sub Bar(ByRef param As Integer)
        arg.Member = -arg.Member ' change argument content
        arg = Nothing ' change (re-assign) argument
    End Sub
End Module
```

Section 18.2: ByVal keyword

ByVal keyword before method parameter (or no keyword as ByVal is assumed by default) says that parameter will be sent in a way **not** allowing the method to change (assign a new value) the variable underlying the parameter. It doesn't prevent the content (or state) of the argument to be changed if it's a class.

```
Class SomeClass
    Public Property Member As Integer
End Class

Module Program
    Sub Main()
        Dim someInstance As New SomeClass With {.Member = 42}

        Foo (someInstance)
        ' here someInstance is not Nothing (still the same object)
        ' but someInstance.Member is -42 (internal state can still be changed)

        Dim number As Integer = 42
        Foo(number)
        ' here number is still 42
    End Sub

    Sub Foo(ByVal arg As SomeClass)
```


www.dbooks.org

```
        arg.Member = -arg.Member ' change argument content
        arg = Nothing ' change (re-assign) argument
    End Sub

    Sub Foo(arg As Integer) ' No ByVal or ByRef keyword, ByVal is assumed
        arg = -arg
    End Sub
End Module
```

Chapter 19: Console

Section 19.1: Console.ReadLine()

```
Dim input as String = Console.ReadLine()
```

`Console.ReadLine()` will read the console input from the user, up until the next newline is detected (usually upon pressing the Enter or Return key). Code execution is paused in the current thread until a newline is provided. Afterwards, the next line of code will be executed.

Section 19.2: Console.Read()

```
Dim inputCode As Integer = Console.Read()
```

`Console.Read()` awaits input from the user and, upon receipt, returns an integer value corresponding with the character code of the entered character. If the input stream is ended in some way before input can be obtained, -1 is returned instead.

Section 19.3: Console.ReadKey()

```
Dim inputChar As ConsoleKeyInfo = Console.ReadKey()
```

`Console.ReadKey()` awaits input from the user and, upon receipt, returns an object of class `ConsoleKeyInfo`, which holds information relevant to the character which the user provided as input. For detail regarding the information provided, visit the MSDN documentation.

Section 19.4: Prototype of command line prompt

```
Module MainPrompt
Public Const PromptSymbol As String = "TLA > "
Public Const ApplicationTitle As String = GetType(Project.BaseClass).Assembly.FullName
REM Or you can use a custom string
REM Public Const ApplicationTitle As String = "Short name of the application"

Sub Main()
    Dim Statement As String
    Dim BrokenDownStatement As String()
    Dim Command As String
    Dim Args As String()
    Dim Result As String

    Console.ForegroundColor = ConsoleColor.Cyan
    Console.Title = ApplicationTitle & " command line console"

    Console.WriteLine("Welcome to " & ApplicationTitle & "console frontend")
    Console.WriteLine("This package is version " &
GetType(Project.BaseClass).Assembly.GetName().Version.ToString)
    Console.WriteLine()
    Console.Write(PromptSymbol)

    Do While True
        Statement = Console.ReadLine()
        BrokenDownStatement = Statement.Split(" ")
        ReDim Args(BrokenDownStatement.Length - 1)
        Command = BrokenDownStatement(0)
```


www.dbooks.org

```
        For i = 1 To BrokenDownStatement.Length - 1
            Args(i - 1) = BrokenDownStatement(i)
        Next

        Select Case Command.ToLower
            Case "example"
                Result = DoSomething(Example)
            Case "exit", "quit"
                Exit Do
            Case "ver"
                Result = "This package is version " &
GetType(Project.BaseClass).Assembly.GetName().Version.ToString
            Case Else
                Result = "Command not acknowldged: -" & Command & "-"
        End Select
        Console.WriteLine(" " & Result)
        Console.Write(PromptSymbol)
    Loop

    Console.WriteLine("I am exiting, time is " & DateTime.Now.ToString("u"))
    Console.WriteLine("Goodbye")
    Environment.Exit(0)
End Sub
End Module
```

This prototype generate a basic command line interpreter.

It automatically get the application name and version to communicate to the user. For each input line, it recognize the command and an arbitrary list of arguments, all separated by space.

As a basic example, this code understand *ver*, *quit* and *exit* commands.

The parameter *Project.BaseClass* is a class of your project where the Assembly details are set.

Section 19.5: Console.WriteLine()

```
Dim x As Int32 = 128
Console.WriteLine(x) ' Variable '
Console.WriteLine(3) ' Integer '
Console.WriteLine(3.14159) ' Floating-point number '
Console.WriteLine("Hello, world") ' String '
Console.WriteLine(myObject) ' Outputs the value from calling myObject.ToString()
```

The `Console.WriteLine()` method will print the given argument(s) **with** a newline attached at the end. This will print any object supplied, including, but not limited to, strings, integers, variables, floating-point numbers.

When writing objects that are not explicitly called out by the various `WriteLine` overloads (that is, you are using the overload that expects a value of type `Object`, WriteLine will use the `.ToString()` method to generate a `String` to actually write. Your custom objects should OverRide the `.ToString` method and produce something more meaningful than the default implementation (which typically just writes the fully qualified type name).

Chapter 20: Functions

The function is just like sub. But function returns a value. A function can accept single or multiple parameters.

Section 20.1: Defining a Function

It's really easy to define the functions.

```
Function GetAreaOfARectangle(ByVal Edge1 As Integer, ByVal Edge2 As Integer) As Integer
    Return Edge1 * Edge2
End Function
```

```
Dim Area As Integer = GetAreaOfARectangle(5, 8)
Console.Writeline(Area) 'Output: 40
```

Section 20.2: Defining a Function #2

```
Function Age(ByVal YourAge As Integer) As String
    Select Case YourAge
        Case Is < 18
            Return("You are younger than 18! You are teen!")
        Case 18 to 64
            Return("You are older than 18 but younger than 65! You are adult!")
        Case Is >= 65
            Return("You are older than 65! You are old!")
    End Select
End Function
```

```
Console.WriteLine(Age(48)) 'Output: You are older than 18 but younger than 65! You are adult!
```


www.dbooks.org

Chapter 21: Recursion

Section 21.1: Compute nth Fibonacci number

Visual Basic.NET, like most languages, permits recursion, a process by which a function calls *itself* under certain conditions.

Here is a basic function in Visual Basic .NET to compute Fibonacci numbers.

```
''' <summary>
''' Gets the n'th Fibonacci number
''' </summary>
''' <param name="n">The 1-indexed ordinal number of the Fibonacci sequence that you wish to receive.
Precondition: Must be greater than or equal to 1.</param>
''' <returns>The nth Fibonacci number. Throws an exception if a precondition is violated.</returns>
Public Shared Function Fibonacci(ByVal n as Integer) as Integer
    If n<1
        Throw New ArgumentOutOfRangeException("n must be greater than or equal to one.")
    End If
    If (n=1) or (n=2)
        ''Base case. The first two Fibonacci numbers (n=1 and n=2) are both 1, by definition.
        Return 1
    End If
    ''Recursive case.
    ''Get the two previous Fibonacci numbers via recursion, add them together, and return the result.
    Return Fibonacci(n-1) + Fibonacci(n-2)
End Function
```

This function works by first checking if the function has been called with the parameter `n` equal to `1` or `2`. By definition, the first two values in the Fibonacci sequence are 1 and 1, so no further computation is necessary to determine this. If `n` is greater than 2, we cannot look up the associated value as easily, but we know that any such Fibonacci number is equal to the sum of the prior two numbers, so we request those via *recursion* (calling our own Fibonacci function). Since successive recursive calls get called with smaller and smaller numbers via decrements of -1 and -2, we know that eventually they will reach numbers that are smaller than 2. Once those conditions (called *base cases*) are reached, the stack unwinds and we get our final result.

Chapter 22: Random

The Random class is used to generate non-negative pseudo-random integers that are not truly random, but are for general purposes close enough.

The sequence is calculated using an initial number (called the **Seed**) In earlier versions of .net, this seed number was the same every time an application was run. So what would happen was that you would get the same sequence of pseudo-random numbers every time the application was executed. Now, the seed is based on the time the object is declared.

Section 22.1: Declaring an instance

```
Dim rng As New Random()
```

This declares an instance of the Random class called `rng`. In this case, the current time at the point where the object is created is used to calculate the seed. This is the most common usage, but has its own problems as we shall see later in the remarks

Instead of allowing the program to use the current time as part of the calculation for the initial seed number, you can specify the initial seed number. This can be any 32 bit integer literal, constant or variable. See below for examples. Doing this means that your instance will generate the same sequence of pseudo-random numbers, which can be useful in certain situations.

```
Dim rng As New Random(43352)
```

or

```
Dim rng As New Random(x)
```

where `x` has been declared elsewhere in your program as an Integer constant or variable.

Section 22.2: Generate a random number from an instance of Random

The following example declares a new instance of the Random class and then uses the method `.Next` to generate the next number in the sequence of pseudo-random numbers.

```
Dim rnd As New Random
Dim x As Integer
x = rnd.Next
```

The last line above will generate the next pseudo-random number and assign it to `x`. This number will be in the range of 0 - 2147483647. However, you can also specify the range of numbers to be generated as in the example below.

```
x = rnd.Next(15, 200)
```

Please note however, that using these parameters, range of numbers will be between 15 or above and 199 or below.

You can also generate floating point numbers of the type Double by using `.NextDouble` e.g

www.dbooks.org

```
Dim rnd As New Random
Dim y As Double
y = rnd.NextDouble()
```

You cannot however specify a range for this. It will always be in the range of 0.0 to less than 1.0.

Chapter 23: Classes

A class groups different functions, methods, variables, and properties, which are called its members. A class encapsulates the members, which can be accessed by an instance of the class, called an object. Classes are extremely useful for the programmer, as they make the task convenient and fast, with characteristics such as modularity, re-usability, maintainability, and readability of the code.

Classes are the building blocks of object-oriented programming languages.

Section 23.1: Abstract Classes

If classes share common functionality you can group this in a base or abstract class. Abstract classes can contain partial or no implementation at all and allow the derived type to override the base implementation.

Abstract classes within VisualBasic.NET must be declared as `MustInherit` and cannot be instantiated.

```
Public MustInherit Class Vehicle
     Private Property _numberOfWheels As Integer
     Private Property _engineSize As Integer

     Public Sub New(engineSize As Integer, wheels As Integer)
         _numberOfWheels = wheels
         _engineSize = engineSize
     End Sub

     Public Function DisplayWheelCount() As Integer
         Return _numberOfWheels
     End Function
End Class
```

A sub type can then `inherit` this abstract class as shown below:

```
Public Class Car
    Inherits Vehicle
End Class
```

Car will inherit all of the declared types within vehicle, but can only access them based upon the underlying access modifier.

```
Dim car As New Car()
car.DisplayWheelCount()
```

In the above example a new Car instance is created. The `DisplayWheelCount()` method is then invoked which will call the base class `Vehicles` implementation.

Section 23.2: Creating classes

Classes provide a way of creating your own types within the .NET framework. Within a class definition you may include the following:

- Fields
- Properties
- Methods
- Constructors

www.dbooks.org

- Events

To declare a class you use the following syntax:

```
Public Class Vehicle
End Class
```

Other .NET types can be encapsulated within the class and exposed accordingly, as shown below:

```
Public Class Vehicle
     Private Property _numberOfWheels As Integer
     Private Property _engineSize As Integer

     Public Sub New(engineSize As Integer, wheels As Integer)
         _numberOfWheels = wheels
         _engineSize = engineSize
     End Sub

     Public Function DisplayWheelCount() As Integer
         Return _numberOfWheels
     End Function
End Class
```

Chapter 24: Generics

Section 24.1: Create a generic class

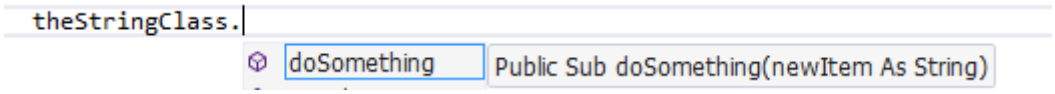
A generic type is created to adapt so that the same functionallity can be accessible for different data types.

```
Public Class SomeClass(Of T)
    Public Sub doSomething(newItem As T)
        Dim tempItem As T
        ' Insert code that processes an item of data type t.
    End Sub
End Class
```

Section 24.2: Instance of a Generic Class

By creating an instance of the same class with a different type given, the interface of the class changes depending on the given type.

```
Dim theStringClass As New SomeClass(Of String)
Dim theIntegerClass As New SomeClass(Of Integer)
```



Section 24.3: Define a 'generic' class

A generic class is a class who adapts to a later-given type so that the same functionality can be offered to different types.

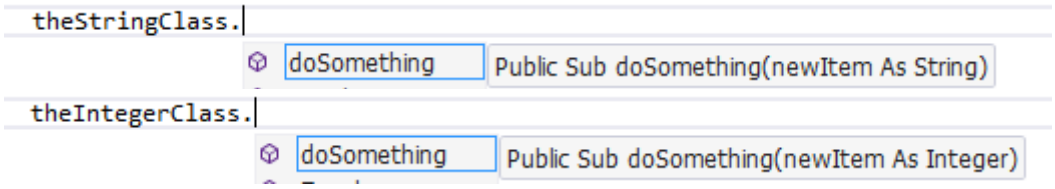
In this basic example a generic class is created. It has a sub who uses the generic type T. While programming this class, we don't know the type of T. In this case T has all the characteristics of Object.

```
Public Class SomeClass(Of T)
    Public Sub doSomething(newItem As T)
        Dim tempItem As T
        ' Insert code that processes an item of data type t.
    End Sub
End Class
```

Section 24.4: Use a generic class

In this example there are 2 instances created of the SomeClass Class. Depending on the type given the 2 instances have a different interface:

```
Dim theStringClass As New SomeClass(Of String)
Dim theIntegerClass As New SomeClass(Of Integer)
```



The most famous generic class is List(of)

Section 24.5: Limit the possible types given

The possible types passed to a new instance of SomeClass must inherit SomeBaseClass. This can also be an interface. The characteristics of SomeBaseClass are accessible within this class definition.

```
Public Class SomeClass(Of T As SomeBaseClass)
    Public Sub DoSomething(newItem As T)
        newItem.DoSomethingElse()
        ' Insert code that processes an item of data type t.
    End Sub
End Class

Public Class SomeBaseClass
    Public Sub DoSomethingElse()
    End Sub
End Class
```

Section 24.6: Create a new instance of the given type

Creating a new intance of a generic type can be done/checed at compile time.

```
Public Class SomeClass(Of T As {New})
    Public Function GetInstance() As T
        Return New T
    End Function
End Class
```

Or with limited types:

```
Public Class SomeClass(Of T As {New, SomeBaseClass})
    Public Function GetInstance() As T
        Return New T
    End Function
End Class

Public Class SomeBaseClass
End Class
```

The baseClass (if none given it is Object) must have a parameter less constructor.

This can also be done at runtime through reflection

Chapter 25: Disposable objects

Section 25.1: Basic concept of IDisposable

Any time you instantiate a class that Implements `IDisposable`, you should call `.Dispose`1 on that class when you have finished using it. This allows the class to clean up any managed or unmanaged dependencies that it may be using. Not doing this could cause a memory leak.

The `Using` keyword ensures that `.Dispose` is called, without you having to *explicitly* call it.

For example without `Using`:

```
Dim sr As New StreamReader("C:\foo.txt")
Dim line = sr.ReadLine
sr.Dispose()
```

Now with `Using`:

```
Using sr As New StreamReader("C:\foo.txt")
    Dim line = sr.ReadLine
End Using '.Dispose is called here for you
```

One major advantage `Using` has is when an exception is thrown, because it *ensures* `.Dispose` is called.

Consider the following. If an exception is thrown, you need to need to remember to call .Dispose but you might also have to check the state of the object to ensure you don't get a null reference error, etc.

```
    Dim sr As StreamReader = Nothing
    Try
        sr = New StreamReader("C:\foo.txt")
        Dim line = sr.ReadLine
    Catch ex As Exception
        'Handle the Exception
    Finally
        If sr IsNot Nothing Then sr.Dispose()
    End Try
```

A using block means you don't have to remember to do this and you can declare your object inside the `try`:

```
    Try
        Using sr As New StreamReader("C:\foo.txt")
            Dim line = sr.ReadLine
        End Using
    Catch ex As Exception
        'sr is disposed at this point
    End Try
```

1 Do I always have to call Dispose() on my DbContext objects? Nope

Section 25.2: Declaring more objects in one Using

Sometimes, you have to create two `Disposable` objects in a row. There is an easy way to avoid nesting `Using` blocks.

This code

www.dbooks.org

```
Using File As New FileStream("MyFile", FileMode.Append)
    Using Writer As New BinaryWriter(File)
        'You code here
        Writer.Writer("Hello")
    End Using
End Using
```

can be shortened into this one. The main advantage is that you gain one indentation level:

```
Using File As New FileStream("MyFile", FileMode.Append), Writer As New BinaryWriter(File)
    'You code here
    Writer.Writer("Hello")
End Using
```

Chapter 26: NullReferenceException

Section 26.1: Empty Return

```
Function TestFunction() As TestClass
    Return Nothing
End Function
```

BAD CODE

```
TestFunction().TestMethod()
```

GOOD CODE

```
Dim x = TestFunction()
If x IsNot Nothing Then x.TestMethod()
```

```
Version = 14.0
```

[Null Conditional Operator]

```
TestFunction()?.TestMethod()
```

Section 26.2: Uninitialized variable

BAD CODE

```
Dim f As System.Windows.Forms.Form
f.ShowModal()
```

GOOD CODE

```
Dim f As System.Windows.Forms.Form = New System.Windows.Forms.Form
' Dim f As New System.Windows.Forms.Form ' alternative syntax
f.ShowModal()
```

EVEN BETTER CODE (Ensure proper disposal of IDisposable object more info)

```
Using f As System.Windows.Forms.Form = New System.Windows.Forms.Form
' Using f As New System.Windows.Forms.Form ' alternative syntax
    f.ShowModal()
End Using
```

Chapter 27: Using Statement

Section 27.1: See examples under Disposable objects

Basic concept of IDisposable

Chapter 28: Option Strict

Section 28.1: Why Use It?

`Option Strict On` prevents three things from happening:

1. Implicit Narrowing Conversion Errors

It prevents you from assigning to a variable that has *less precision or smaller capacity* (a narrowing conversion) without an explicit cast. Doing so would result in data loss.

```
Dim d As Double = 123.4
Dim s As Single = d 'This line does not compile with Option Strict On
```

2. Late Binding Calls

Late binding is not allowed. This is to prevent typos that would compile, but fail at runtime

```
Dim obj As New Object
obj.Foo 'This line does not compile with Option Strict On
```

3. Implicit Object Type Errors

This prevents variable being inferred as an Object when in fact they should have been declared as a type

```
Dim something = Nothing. 'This line does not compile with Option Strict On
```

Conclusion

Unless you need to do late binding, you should always have `Option Strict On` as it will cause the mentioned errors to generate compile time errors instead of runtime exceptions.

If you *have* to do late binding, you can *either*

- Wrap all your late binding calls into one class/module and use `Option Strict Off` at the top of the code file (this is the preferred method as it reduces the likelihood of a typos in other files), *or*
- Specify that Late Binding does not cause a compilation failure (`Project Properties > Compile Tab > Warning Configuration`)

Section 28.2: How to Switch It On

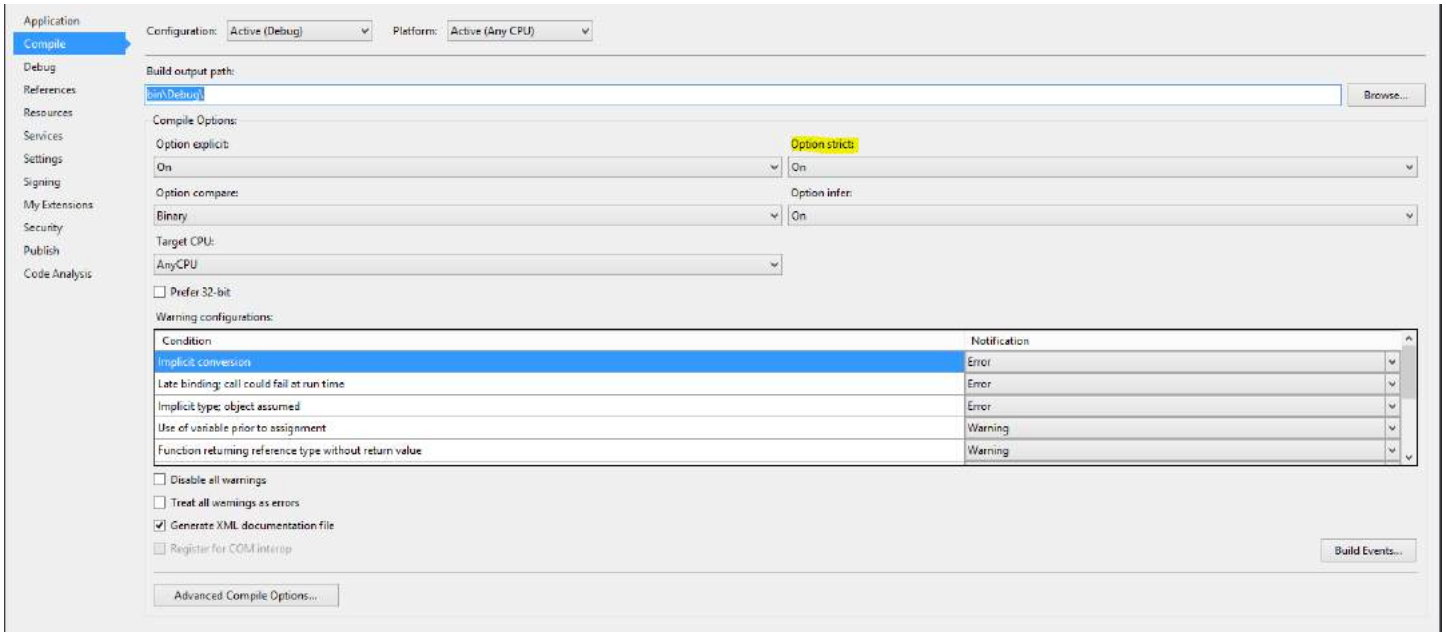
- You can switch it On at the Module/Class Level by placing the directive at the top of the code file.

```
Option Strict On
```

- You can switch it on at the project level via the menu in Visual Studio

> Project > [Project] Properties > Compile Tab > Option Strict > On

www.dbooks.org



- You can switch it On by default for all new Projects by selecting:

> Tools > Options > Projects and Solutions > VB defaults > Option Strict
> Set it to On.

Chapter 29: Option Explicit

Section 29.1: What is it?

It forces you to explicitly declare all variables.

What is the difference between explicitly declaring and implicitly declaring a variable?

Explicitly declaring a variable:

```
Dim anInteger As Integer = 1234
```

Implicitly declaring a variable:

```
'Did not declare aNumber using Dim
aNumber = 1234
```

Conclusion

Therefore, you should always have `Option Explicit On` as you could misspel a variable during assignment, which cause your program to behave unexpectedly.

Section 29.2: How to switch it on?

Document level

It is on by default, but you can have an extra layer of protection by placing `Option Explicit On` at the top of the code file. The option will apply to the whole document.

Project level

You can switch it on via the menu in Visual Studio:

> Project > [Project] Properties > Compile Tab > Option Explicit

Choose `On` in the drop-down menu. The option will apply to the whole document.

All new projects

You can switch it On by default for all new Projects by selecting:

> Tools > Options > Projects and Solutions > VB defaults > Option Explicit

Choose `On` in the drop-down menu.

Chapter 30: Option Infer

Section 30.1: How to enable/disable it

Document level

It is on by default, but you can set it by placing `Option Infer On|Off` at the top of the code file. The option will apply to the whole document.

Project level

You can switch it on/off via the menu in Visual Studio:

> Project > [Project] Properties > Compile Tab > Option infer

Choose `On|Off` in the drop-down menu. The option will apply to the whole document.

All new projects

You can switch it On by default for all new Projects by selecting:

> Tools > Options > Projects and Solutions > VB defaults > Option Infer

Choose `On|Off` in the drop-down menu.

Section 30.2: What is it?

Enables the use of local type inference in declaring variables.

What is type inference?

You can declare local variables without explicitly stating a data type. The compiler infers the data type of a variable from the type of its initialization expression.

Option Infer On:

```
Dim aString  = "1234" '--> Will be treated as String by the compiler
Dim aNumber  = 4711   '--> Will be treated as Integer by the compiler
```

vs. explicit type declaration:

```
'State a type explicitly
Dim aString as String = "1234"
Dim aNumber as Integer = 4711
```

Option Infer Off:
The compiler behavior with `Option Infer Off` depends on the `Option Strict` setting which is already documented here.

- **Option Infer Off - Option Strict Off**
  All variables without explicit type declarations are declared as `Object`.

```
Dim aString  = "1234" '--> Will be treated as Object by the compiler
```

- **Option Infer Off - Option Strict On**
  The compiler won´t let you declare a variable without an explicit type.

```
'Dim aString  = "1234" '--> Will not compile due to missing type in declaration
```

Section 30.3: When to use type inference

Basically you can use type inference whenever it is possible.
However, be careful when combining `Option Infer Off` and `Option Strict Off`, as this can lead to undesired run-time behavior:

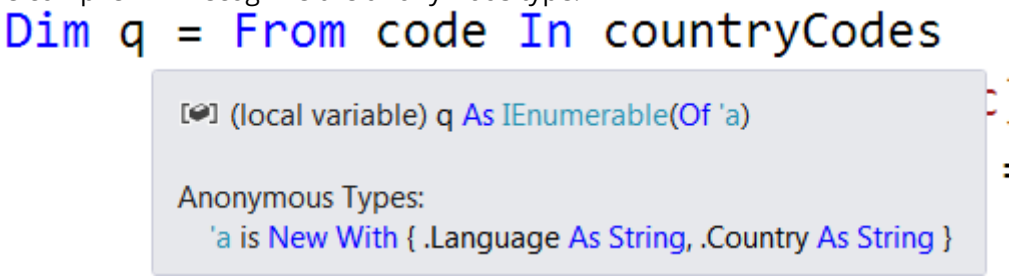
```
Dim someVar = 5
someVar.GetType.ToString() '--> System.Int32
someVar = "abc"
someVar.GetType.ToString() '--> System.String
```

Anonymous Type
Anonymous types can **only** be declared with `Option Infer On`.
They are often used when dealing with LINQ:

```
Dim countryCodes = New List(Of String)
countryCodes.Add("en-US")
countryCodes.Add("en-GB")
countryCodes.Add("de-DE")
countryCodes.Add("de-AT")

Dim q = From code In countryCodes
        Let split = code.Split("-"c)
        Select New With {.Language = split(0), .Country = split(1)}
```

- **Option Infer On**
  The compiler will recognize the anonymous type:



- **Option Infer Off**
  The compiler will either throw an error (with `Option Strict On`)
  or will consider `q` as type `object` (with `Option Strict Off`).
  Both cases will produce the outcome that you cannot use the anonymous type.

Doubles/Decimals
Numeric variables with decimal places will be infered as `Double` by default:

```
Dim aNumber = 44.11 '--> Will be treated as type `Double` by the compiler
```

If another type like `Decimal` is desired the value which initialized the variable needs to be marked:

```vb
Dim mDecimal = 47.11D '--> Will be treated as type `Decimal` by the compiler
```

Chapter 31: Error Handling

Section 31.1: Try...Catch...Finally Statement

Structure:

```
Try
    'Your program will try to run the code in this block.
    'If any exceptions are thrown, the code in the Catch Block will be executed,
    'without executing the lines after the one which caused the exception.
Catch ex As System.IO.IOException
    'If an exception occurs when processing the Try block, each Catch statement
    'is examined in textual order to determine which handles the exception.
    'For example, this Catch block handles an IOException.
Catch ex As Exception
    'This catch block handles all Exception types.
    'Details of the exception, in this case, are in the "ex" variable.
    'You can show the error in a MessageBox with the below line.
    MessageBox.Show(ex.Message)
Finally
    'A finally block is always executed, regardless of if an Exception occurred.
End Try
```

Example Code:

```
Try
    Dim obj = Nothing
    Dim prop = obj.Name 'This line will throw a NullReferenceException

    Console.WriteLine("Test.") ' This line will NOT be executed
Catch ex As System.IO.IOException
    ' Code that reacts to IOException.
Catch ex As NullReferenceException
    ' Code that reacts to a NullReferenceException
    Console.WriteLine("NullReferenceException: " & ex.Message)
    Console.WriteLine("Stack Trace: " & ex.StackTrace)
Catch ex As Exception
    ' Code that reacts to any other exception.
Finally
    ' This will always be run, regardless of if an exception is thrown.
    Console.WriteLine("Completed")
End Try
```

Section 31.2: Creating custom exception and throwing

You can create a custom exception and throw them during the execution of your function. As a general practice you should only throw an exception when your function could not achieve its defined functionality.

```
Private Function OpenDatabase(Byval Server as String, Byval User as String, Byval Pwd as String)
    if Server.trim="" then
        Throw new Exception("Server Name cannot be blank")
    elseif User.trim ="" then
        Throw new Exception("User name cannot be blank")
    elseif Pwd.trim="" then
        Throw new Exception("Password cannot be blank")
    endif

    'Here add codes for connecting to the server
```


www.dbooks.org

```
End function
```

Section 31.3: Try Catch in Database Operation

You can use Try..Catch to rollback database operation by placing the rollback statement at the Catch Segment.

```
    Try
        'Do the database operation...
        xCmd.CommandText = "INSERT into ...."
        xCmd.ExecuteNonQuery()

        objTrans.Commit()
        conn.Close()
    Catch ex As Exception
        'Rollback action when something goes off
        objTrans.Rollback()
        conn.Close()
    End Try
```

Section 31.4: The Un-catchable Exception

Although `Catch ex As Exception` claims that it can handle all exceptions - there are one exception (no pun intended).

```
Imports System
Static Sub StackOverflow() ' Again no pun intended
    StackOverflow()
End Sub
Static Sub Main()
    Try
        StackOverflow()
    Catch ex As Exception
        Console.WriteLine("Exception caught!")
    Finally
        Console.WriteLine("Finally block")
    End Try
End Sub
```

Oops... There is an un-caught `System.StackOverflowException` while the console didn't even print out anything! According to MSDN,

> Starting with the .NET Framework 2.0, you can't catch a StackOverflowException object with a try/catch block, and the corresponding process is terminated by default. Consequently, you should write your code to detect and prevent a stack overflow.

So, `System.StackOverflowException` is un-catchable. Beware of that!

Section 31.5: Critical Exceptions

Generally most of the exceptions are not that critical, but there are some really serious exceptions that you might not be capable to handle, such as the famous `System.StackOverflowException`. However, there are others that might get hidden by `Catch ex As Exception`, such as `System.OutOfMemoryException`, `System.BadImageFormatException` and `System.InvalidProgramException`. It is a good programming practice to leave these out if you cannot correctly handle them. To filter out these exceptions, we need a helper method:

```
Public Shared Function IsCritical(ex As Exception) As Boolean
    Return TypeOf ex Is OutOfMemoryException OrElse
           TypeOf ex Is AppDomainUnloadedException OrElse
           TypeOf ex Is AccessViolationException OrElse
           TypeOf ex Is BadImageFormatException OrElse
           TypeOf ex Is CannotUnloadAppDomainException OrElse
           TypeOf ex Is ExecutionEngineException OrElse ' Obsolete one, but better to include
           TypeOf ex Is InvalidProgramException OrElse
           TypeOf ex Is System.Threading.ThreadAbortException
End Function
```

Usage:

```
Try
    SomeMethod()
Catch ex As Exception When Not IsCritical(ex)
    Console.WriteLine("Exception caught: " & ex.Message)
End Try
```

Chapter 32: OOP Keywords

Section 32.1: Defining a class

Classes are vital aspects of OOP. A class is like the "blueprint" of an object. An object has the properties of a class, but the characteristics are not defined within the class itself. As each object can be different, they define their own characteristics.

```
Public Class Person
End Class

Public Class Customer
End Class
```

A class can also contain *subclasses*. A subclass inherits the same properties and behaviors as its parent class, but can have its own unique properties and classes.

Section 32.2: Inheritance Modifiers (on classes)

Inherits

Specifies the base (or parent) class

```
Public Class Person
End Class

Public Class Customer
    Inherits Person

End Class

'One line notation
Public Class Student : Inherits Person
End Class
```

Possible objects:

```
Dim p As New Person
Dim c As New Customer
Dim s As New Student
```

NotInheritable

Prevents programmers from using the class as a base class.

```
Public NotInheritable Class Person
End Class
```

Possible objects:

```
Dim p As New Person
```

MustInherit

Specifies that the class is intended for use as a base class only. (Abstract class)

```
Public MustInherit Class Person
```

```
End Class

Public Class Customer
    Inherits Person
End Class
```

Possible objects:

```
Dim c As New Customer
```

Section 32.3: Inheritance Modifiers (on properties and methods)

Overridable

Allows a property or method in a class to be overridden in a derived class.

```
Public Class Person
    Public Overridable Sub DoSomething()
        Console.WriteLine("Person")
    End Sub
End Class
```

Overrides

Overrides an Overridable property or method defined in the base class.

```
Public Class Customer
    Inherits Person

    'Base Class must be Overridable
    Public Overrides Sub DoSomething()
        Console.WriteLine("Customer")
    End Sub
End Class
```

NotOverridable

Prevents a property or method from being overridden in an inheriting class. Default behaviour. Can only be declared on **overrides methods**

```
Public Class Person

    Public Overridable Sub DoSomething()
        Console.WriteLine("Person")
    End Sub

End Class

Public Class Customer
    Inherits Person

    Public NotOverridable Overrides Sub DoSomething()
        Console.WriteLine("Customer")
    End Sub

End Class

Public Class DetailedCustomer
    Inherits Customer
```


www.dbooks.org

```
    'DoSomething can't be overridden
End Class
```

Example Usage:

```
Dim p As New Person
p.DoSomething()

Dim c As New Customer
c.DoSomething()

Dim d As New DetailedCustomer
d.DoSomething()
```

Output:

```
Person
Customer
Customer
```

MustOverride

Requires that a derived class override the property or method.

MustOverride methods must be declared in **MustInherit classes.**

```
Public MustInherit Class Person

    Public MustOverride Sub DoSomething()
    'No method definition here

End Class

Public Class Customer
    Inherits Person

    'DoSomething must be overridden
    Public Overrides Sub DoSomething()
        Console.WriteLine("Customer")
    End Sub

End Class
```

Example Usage:

```
Dim c As New Customer
c.DoSomething()
```

Output:

```
Customer
```

Section 32.4: MyBase

The MyBase keyword behaves like an object variable that refers to the base class of the current instance of a class.

```
Public Class Person
    Public Sub DoSomething()
```

```
        Console.WriteLine("Person")
    End Sub
End Class

Public Class Customer
    Inherits Person

    Public Sub DoSomethingElse()
        MyBase.DoSomething()
    End Sub

End Class
```

Usage example:

```
Dim p As New Person
p.DoSomething()

Console.WriteLine("----")

Dim c As New Customer
c.DoSomething()
c.DoSomethingElse()
```

Output:

```
Person
----
Person
Person
```

Section 32.5: Me vs MyClass

Me uses the current object instance.

MyClass uses the memberdefinition in the class where the member is called

```
Class Person
    Public Overridable Sub DoSomething()
        Console.WriteLine("Person")
    End Sub

    Public Sub useMe()
        Me.DoSomething()
    End Sub

    Public Sub useMyClass()
        MyClass.DoSomething()
    End Sub
End Class

Class Customer
    Inherits Person

    Public Overrides Sub DoSomething()
        Console.WriteLine("Customer")
    End Sub
End Class
```

Example Usage:

```
Dim c As New Customer
c.useMe()
c.useMyClass()
```

Output:

```
Customer
Person
```

Section 32.6: Overloading

Overloading is the creation of more than one procedure, instance constructor, or property in a class with the same name but different argument types.

```
Class Person
    Overloads Sub Display(ByVal theChar As Char)
        ' Add code that displays Char data.
    End Sub

    Overloads Sub Display(ByVal theInteger As Integer)
        ' Add code that displays Integer data.
    End Sub

    Overloads Sub Display(ByVal theDouble As Double)
        ' Add code that displays Double data.
    End Sub
End Class
```

Section 32.7: Shadows

It redeclares a member that is not overridable. Only calls to the instance will be affected. Code inside the base classes will not be affected by this.

```
Public Class Person
    Public Sub DoSomething()
        Console.WriteLine("Person")
    End Sub


    Public Sub UseMe()
        Me.DoSomething()
    End Sub
End Class
Public Class Customer
    Inherits Person
    Public Shadows Sub DoSomething()
        Console.WriteLine("Customer")
    End Sub

End Class
```

Example usage:

```
Dim p As New Person
Dim c As New Customer
```

```
p.UseMe()
c.UseMe()
Console.WriteLine("----")
p.DoSomething()
c.DoSomething()
```

Output:

```
Person
Person
----
Person
Customer
```

Pitfalls:

Example1, Creating a new object through a generic. Which function will be used??

```
Public Sub CreateAndDoSomething(Of T As {Person, New})()
    Dim obj As New T
    obj.DoSomething()
End Sub
```

example usage:

```
Dim p As New Person
p.DoSomething()
Dim s As New Student
s.DoSomething()
Console.WriteLine("----")
CreateAndDoSomething(Of Person)()
CreateAndDoSomething(Of Student)()
```

Output: By intuition the result should be the same. Yet that is not true.

```
Person
Student
----
Person
Person
```

Example 2:

```
Dim p As Person
Dim s As New Student
p = s
p.DoSomething()
s.DoSomething()
```

Output: By intuition you could think that p and s are equal and will behave equal. Yet that is not true.

```
Person
Student
```

In this simple examples it is easy to learn the strange behaviour of Shadows. But in real-life it brings a lot of

www.dbooks.org

surprises. It is advisably to prevent the usage of shadows. One should use other alternatives as much as possible (overrides etc..)

Section 32.8: Interfaces

```
Public Interface IPerson
    Sub DoSomething()
End Interface

Public Class Customer
    Implements IPerson
    Public Sub DoSomething() Implements IPerson.DoSomething
        Console.WriteLine("Customer")
    End Sub

End Class
```

Chapter 33: Extension methods

Section 33.1: Creating an extension method

Extension methods are useful to extend the behaviour of libraries we don't own.

They are used similar to instance methods thanks to the compiler's syntactic sugar:

```
Sub Main()
    Dim stringBuilder = new StringBuilder()

    'Extension called directly on the object.
    stringBuilder.AppendIf(true, "Condition was true")

    'Extension called as a regular method. This defeats the purpose
    'of an extension method but should be noted that it is possible.
    AppendIf(stringBuilder, true, "Condition was true")

End Sub

<Extension>
Public Function AppendIf(stringBuilder As StringBuilder, condition As Boolean, text As String) As
StringBuilder
    If(condition) Then stringBuilder.Append(text)

    Return stringBuilder
End Function
```

To have a usable extension method, the method needs the `Extension` attribute and needs to be declared in a `Module`.

Section 33.2: Making the language more functional with extension methods

A good use of extension method is to make the language more functional

```
Sub Main()
    Dim strings = { "One", "Two", "Three" }

    strings.Join(Environment.NewLine).Print()
End Sub

<Extension>
Public Function Join(strings As IEnumerable(Of String), separator As String) As String
    Return String.Join(separator, strings)
End Function

<Extension>
Public Sub Print(text As String)
    Console.WriteLine(text)
End Sub
```

Section 33.3: Getting Assembly Version From Strong Name

Example of calling an extension method as an extension and as a regular method.

```
public Class MyClass
```


www.dbooks.org

```
  Sub Main()

        'Extension called directly on the object.
        Dim Version = Assembly.GetExecutingAssembly.GetVersionFromAssembly()

        'Called as a regular method.
        Dim Ver = GetVersionFromAssembly(SomeOtherAssembly)

    End Sub
End Class
```

The Extension Method in a Module. Make the Module Public if extensions are compiled to a dll and will be referenced in another assembly.

```
Public Module Extensions
    ''' <summary>
    ''' Returns the version number from the specified assembly using the assembly's strong name.
    ''' </summary>
    ''' <param name="Assy">[Assembly] Assembly to get the version info from.</param>
    ''' <returns>[String]</returns>
    <Extension>
    Friend Function GetVersionFromAssembly(ByVal Assy As Assembly) As String
        Return Split(Split(Assy.FullName,",")(1),"=")(1)
    End Function
End Module
```

Section 33.4: Padding Numerics

```
Public Module Usage
  Public Sub LikeThis()
    Dim iCount As Integer
    Dim sCount As String

    iCount = 245
    sCount = iCount.PadLeft(4, "0")

    Console.WriteLine(sCount)
    Console.ReadKey()
  End Sub
End Module



Public Module Padding
  <Extension>
  Public Function PadLeft(Value As Integer, Length As Integer) As String
    Return Value.PadLeft(Length, Space(Length))
  End Function



  <Extension>
  Public Function PadRight(Value As Integer, Length As Integer) As String
    Return Value.PadRight(Length, Space(Length))
  End Function



  <Extension>
  Public Function PadLeft(Value As Integer, Length As Integer, Character As Char) As String
```

```
    Return CStr(Value).PadLeft(Length, Character)
  End Function



  <Extension>
  Public Function PadRight(Value As Integer, Length As Integer, Character As Char) As String
    Return CStr(Value).PadRight(Length, Character)
  End Function
End Module
```

Chapter 34: Reflection

Section 34.1: Retrieve Properties for an Instance of a Class

```
Imports System.Reflection

Public Class PropertyExample

    Public Function GetMyProperties() As PropertyInfo()
        Dim objProperties As PropertyInfo()
        objProperties = Me.GetType.GetProperties(BindingFlags.Public Or BindingFlags.Instance)
        Return objProperties
    End Function

    Public Property ThisWillBeRetrieved As String = "ThisWillBeRetrieved"

    Private Property ThisWillNot As String = "ThisWillNot"

    Public Shared Property NeitherWillThis As String = "NeitherWillThis"

    Public Overrides Function ToString() As String
        Return String.Join(",", GetMyProperties.Select(Function(pi) pi.Name).ToArray)
    End Function
End Class
```

The Parameter of GetProperties defines which kinds of Properties will be returned by the function. Since we pass Public and Instance, the method will return only properties that are both public and non-shared. See The Flags attribute for and explanation on how Flag-enums can be combined.

Section 34.2: Get a method and invoke it

Static method:

```
Dim parseMethod = GetType(Integer).GetMethod("Parse",{GetType(String)})
Dim result = DirectCast(parseMethod.Invoke(Nothing,{"123"}), Integer)
```

Instance method:

```
Dim instance = "hello".ToUpper
Dim method = Gettype(String).GetMethod("ToUpper",{})
Dim result = method.Invoke(instance,{})
Console.WriteLine(result) 'HELLO
```

Section 34.3: Create an instance of a generic type

```
    Dim openListType = GetType(List(Of ))
    Dim typeParameters = {GetType(String)}
    Dim stringListType = openListType.MakeGenericType(typeParameters)
    Dim instance = DirectCast(Activator.CreateInstance(stringListType), List(Of String))
    instance.Add("Hello")
```

Section 34.4: Get the members of a type

```
Dim flags = BindingFlags.Static Or BindingFlags.Public Or BindingFlags.Instance
Dim members = GetType(String).GetMembers(flags)
```

```vb
For Each member In members
    Console.WriteLine($"{member.Name}, ({member.MemberType})")
Next
```

Chapter 35: Visual Basic 14.0 Features

Visual Basic 14 is the version of Visual Basic that was shipped as part of Visual Studio 2015.

This version was rewritten from scratch in about 1.3 million lines of VB. Many features were added to remove common irritations and to make common coding patterns cleaner.

The version number of Visual Basic went straight from 12 to 14, skipping 13. This was done to keep VB in line with the version numbering of Visual Studio itself.

Section 35.1: Null conditional operator

To avoid verbose null checking, the `?.` operator has been introduced in the language.

The old verbose syntax:

```
If myObject IsNot Nothing AndAlso myObject.Value >= 10 Then
```

Can be now replaced by the concise:

```
If myObject?.Value >= 10 Then
```

The `?` operator is particularly powerful when you have a chain of properties. Consider the following:

```
Dim fooInstance As Foo = Nothing
Dim s As String
```

Normally you would have to write something like this:

```
If fooInstance IsNot Nothing AndAlso fooInstance.BarInstance IsNot Nothing Then
    s = fooInstance.BarInstance.Baz
Else
    s = Nothing
End If
```

But with the `?` operator this can be replaced with just:

```
s = fooInstance?.BarInstance?.Baz
```

Section 35.2: String interpolation

This new feature makes the string concatenation more readable. This syntax will be compiled to its equivalent `String.Format` call.

Without string interpolation:

```
String.Format("Hello, {0}", name)
```

With string interpolation:

```
$"Hello, {name}"
```

The two lines are equivalent and both get compiled to a call to `String.Format`.

As in `String.Format`, the brackets can contain any single expression (call to a method, property, a null coalescing operator et cetera).

String Interpolation is the preferred method over `String.Format` because it prevents some runtime errors from occurring. Consider the following `String.Format` line:

```
String.Format("The number of people is {0}/{1}", numPeople)
```

This will compile, but will cause a runtime error as the compiler does not check that the number of arguments match the placeholders.

Section 35.3: Read-Only Auto-Properties

Read-only properties were always possible in VB.NET in this format:

```
Public Class Foo

  Private _MyProperty As String = "Bar"

  Public ReadOnly Property MyProperty As String
    Get
      Return _MyProperty
    End Get
  End Property

End Class
```

The new version of Visual Basic allows a short hand for the property declaration like so:

```
Public Class Foo

  Public ReadOnly Property MyProperty As String = "Bar"

End Class
```

The actual implementation that is generated by the compiler is exactly the same for both examples. The new method to write it is just a short hand. The compiler will still generate a private field with the format: `_<PropertyName>` to back the read-only property.

Section 35.4: NameOf operator

The `NameOf` operator resolves namespaces, types, variables and member names at compile time and replaces them with the string equivalent.

One of the use cases:

```
Sub MySub(variable As String)
    If variable Is Nothing Then Throw New ArgumentNullException("variable")
End Sub
```

The old syntax will expose the risk of renaming the variable and leaving the hard-coded string to the wrong value.

```
Sub MySub(variable As String)
    If variable Is Nothing Then Throw New ArgumentNullException(NameOf(variable))
End Sub
```

With `NameOf`, renaming the variable only will raise a compiler error. This will also allow the renaming tool to rename both with a single effort.

The `NameOf` operator only uses the last component of the reference in the brackets. This is important when handling something like namespaces in the `NameOf` operator.

```
Imports System

Module Module1
    Sub WriteIO()
        Console.WriteLine(NameOf(IO)) 'displays "IO"
        Console.WriteLine(NameOf(System.IO)) 'displays "IO"
    End Sub
End Module
```

The operator also uses the name of the reference that is typed in without resolving any name changing imports. For example:

```
Imports OldList = System.Collections.ArrayList

Module Module1
    Sub WriteList()
        Console.WriteLine(NameOf(OldList)) 'displays "OldList"
        Console.WriteLine(NameOf(System.Collections.ArrayList)) 'displays "ArrayList"
    End Sub
End Module
```

Section 35.5: Multiline string literals

VB now allows string literals that split over multiple lines.

Old syntax:

```
Dim text As String = "Line1" & Environment.NewLine & "Line2"
```

New syntax:

```
Dim text As String = "Line 1
Line 2"
```

Section 35.6: Partial Modules and Interfaces

Similar to partial classes the new version of Visual Basic is now able to handle partial modules and partial interfaces. The syntax and behaviour is exactly the same as it would be for partial classes.

A partial module example:

```
Partial Module Module1
  Sub Main()
    Console.Write("Ping -> ")
    TestFunktion()
  End Sub
End Module

Partial Module Module1
  Private Sub TestFunktion()
    Console.WriteLine("Pong")
```

```
  End Sub
End Module
```

And a partial interface:

```
Partial Interface Interface1
  Sub Methode1()
End Interface

Partial Interface Interface1
  Sub Methode2()
End Interface

Public Class Class1
  Implements Interface1
  Public Sub Methode1() Implements Interface1.Methode1
    Throw New NotImplementedException()
  End Sub

  Public Sub Methode2() Implements Interface1.Methode2
    Throw New NotImplementedException()
  End Sub
End Class
```

Just like for partial classes the definitions for the partial modules and interfaces have to be located in the same namespace and the same assembly. This is because the partial parts of the modules and interfaces are merged during the compilation and the compiled assembly does not contain any indication that the original definition of the module or interface was split.

Section 35.7: Comments after implicit line continuation

VB 14.0 introduces the ability to add comments after implicit line continuation.

```
Dim number =
    From c As Char 'Comment
    In "dj58kwd92n4" 'Comment
    Where Char.IsNumber(c) 'Comment
    Select c 'Comment
```

Section 35.8: #Region directive improvements

#Region directive can now be placed inside methods and can even span over methods, classes and modules.

```
#Region "A Region Spanning A Class and Ending Inside Of A Method In A Module"
   Public Class FakeClass
    'Nothing to see here, just a fake class.
   End Class

   Module Extensions

    ''' <summary>
    ''' Checks the path of files or directories and returns [TRUE] if it exists.
    ''' </summary>
    ''' <param name="Path">[Sting] Path of file or directory to check.</param>
    ''' <returns>[Boolean]</returns>
    <Extension>
    Public Function PathExists(ByVal Path As String) As Boolean
        If My.Computer.FileSystem.FileExists(Path) Then Return True
```


www.dbooks.org

```
        If My.Computer.FileSystem.DirectoryExists(Path) Then Return True
        Return False
    End Function

    ''' <summary>
    ''' Returns the version number from the specified assembly using the assembly's strong name.
    ''' </summary>
    ''' <param name="Assy">[Assembly] Assembly to get the version info from.</param>
    ''' <returns>[String]</returns>
    <Extension>
    Friend Function GetVersionFromAssembly(ByVal Assy As Assembly) As String
#End Region
        Return Split(Split(Assy.FullName, ",")(1), "=")(1)
    End Function
End Module
```

Chapter 36: LINQ

LINQ (Language Integrated Query) is an expression that retrieves data from a data source. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

Section 36.1: Selecting from array with simple condition

```
Dim sites() As String = {"Stack Overflow", "Super User", "Ask Ubuntu", "Hardware
Recommendations"}
Dim query = From x In sites Where x.StartsWith("S")
' result = "Stack Overflow", "Super User"
```

Query will be enumerable object containing `Stack Overflow` and `Super User`. `x` in the query is iterating variable where will be stored each object checked by `Where` clause.

Section 36.2: Mapping array by Select clause

```
Dim sites() As String = {"Stack Overflow",
                         "Super User",
                         "Ask Ubuntu",
                         "Hardware Recommendations"}
Dim query = From x In sites Select x.Length
' result = 14, 10, 10, 24
```

Query result will be enumerable object containing lengths of strings in input array. In this example this would be values 14, 10, 10, 24. x in the query is iterating variable where will be stored each object from the input array.

Section 36.3: Ordering output

```
Dim sites() As String = {"Stack Overflow",
                         "Super User",
                         "Ask Ubuntu",
                         "Hardware Recommendations"}

Dim query = From x In sites
            Order By x.Length

' result = "Super User", "Ask Ubuntu", "Stack Overflow", "Hardware Recommendations"
```

OrderBy clause orders the output by the value returned from the clause. In this example it is Length of each string. Default output order is ascending. If you need descending you could specify `Descending` keyword after clause.

```
Dim query = From x In sites
            Order By x.Length Descending
```

Section 36.4: Generating Dictionary From IEnumerable

```
' Just setting up the example
Public Class A
    Public Property ID as integer
    Public Property Name as string
```


www.dbooks.org

```
    Public Property OtherValue as Object
End Class

Public Sub Example()
    'Setup the list of items
    Dim originalList As New List(Of A)
    originalList.Add(New A() With {.ID = 1, .Name = "Item 1", .OtherValue = "Item 1 Value"})
    originalList.Add(New A() With {.ID = 2, .Name = "Item 2", .OtherValue = "Item 2 Value"})
    originalList.Add(New A() With {.ID = 3, .Name = "Item 3", .OtherValue = "Item 3 Value"})

    'Convert the list to a dictionary based on the ID
    Dim dict As Dictionary(Of Integer, A) = originalList.ToDictionary(function(c) c.ID, function(c)
c)

    'Access Values From The Dictionary
    console.Write(dict(1).Name) ' Prints "Item 1"
    console.Write(dict(1).OtherValue) ' Prints "Item 1 Value"
End Sub
```

Section 36.5: Projection

```
' sample data
Dim sample = {1, 2, 3, 4, 5}

' using "query syntax"
Dim squares = From number In sample Select number * number

' same thing using "method syntax"
Dim squares = sample.Select (Function (number) number * number)
```

We can project multiple result at once too

```
Dim numbersAndSquares =
    From number In sample Select number, square = number * number

Dim numbersAndSquares =
    sample.Select (Function (number) New With {Key number, Key .square = number * number})
```

Section 36.6: Getting distinct values (using the Distinct method)

```
Dim duplicateFruits = New List(Of String) From {"Grape", "Apple", "Grape", "Apple", "Grape"}
'At this point, duplicateFruits.Length = 5

Dim uniqueFruits = duplicateFruits.Distinct();
'Now, uniqueFruits.Count() = 2
'If iterated over at this point, it will contain 1 each of "Grape" and "Apple"
```

Chapter 37: FTP server

Section 37.1: Download file from FTP server

```
My.Computer.Network.DownloadFile("ftp://server.my/myfile.txt", "donwloaded_file.txt")
```

This command download `myfile.txt` file from server named `server.my` and saves it as `donwloaded_file.txt` into working directory. You can specify absolute path for downloaded file.

Section 37.2: Download file from FTP server when login required

```
My.Computer.Network.DownloadFile("ftp://srv.my/myfile.txt", "donwload.txt", "Peter", "1234")
```

This command download `myfile.txt` file from server named `srv.my` and saves it as `donwload.txt` into working directory. You can specify absolute path for downloaded file. File is download by user Peter with password 1234.

Section 37.3: Upload file to FTP server

```
My.Computer.Network.UploadFile("example.txt", "ftp://server.my/server_example.txt")
```

This command upload `example.txt` file from working directory (you could specify absolute path if you want) to server named `server.my`. File stored on the server will be named `server_example.txt`.

Section 37.4: Upload file to FTP server when login required

```
My.Computer.Network.UploadFile("doc.txt", "ftp://server.my/on_server.txt", "Peter", "1234")
```

This command upload `doc.txt` file from working directory (you could specify absolute path if you want) to server named `server.my`. File stored on the server will be named `server_example.txt`. Fill is send on the server by user Peter and password 1234.

www.dbooks.org

Chapter 38: Working with Windows Forms

Section 38.1: Using the default Form instance

VB.NET offers default Form instances. The developer does not need to create the instance as it is created behind the scenes. However, *it is **not** preferable* to use the default instance all but the simplest programs.

```
Public Class Form1

    Public Sub Foo()
        MessageBox.Show("Bar")
    End Sub

End Class

Module Module1

    Public Sub Main()
        ' Default instance
        Form1.Foo()
        ' New instance
        Dim myForm1 As Form1 = New Form1()
        myForm1.Foo()

    End Sub

End Module
```

See also:

- Do you have to explicitly create instance of form in VB.NET?
- Why is there a default instance of every form in VB.Net but not in C#?

Section 38.2: Passing Data From One Form To Another

Sometimes you might want to pass information that has been generated in one form, to another form for additional use. This is useful for forms that display a search tool, or a settings page among many other uses.

Let's say you want to pass a `DataTable` between a form that is already open *(MainForm)* and a new form *(NewForm)*:

In The MainForm:

```
  Private Sub Open_New_Form()
      Dim NewInstanceOfForm As New NewForm(DataTable1)
      NewInstanceOfForm.ShowDialog()
    End Sub
```

In The NewForm

```
Public Class NewForm
    Dim NewDataTable as Datatable

    Public Sub New(PassedDataTable As Datatable)
       InitializeComponent()
       NewDataTable= PassedDataTable
    End Sub
```

```
End Class
```

Now when the *NewForm* is opened, it is passed `DataTable1` from *MainForm* and stored as `NewDataTable` in *NewForm* for use by that form.

This can be extremely useful when trying to pass large amounts of information between forms, especially when combining all of the information in to a single `ArrayList` and passing the `ArrayList` to the new form.

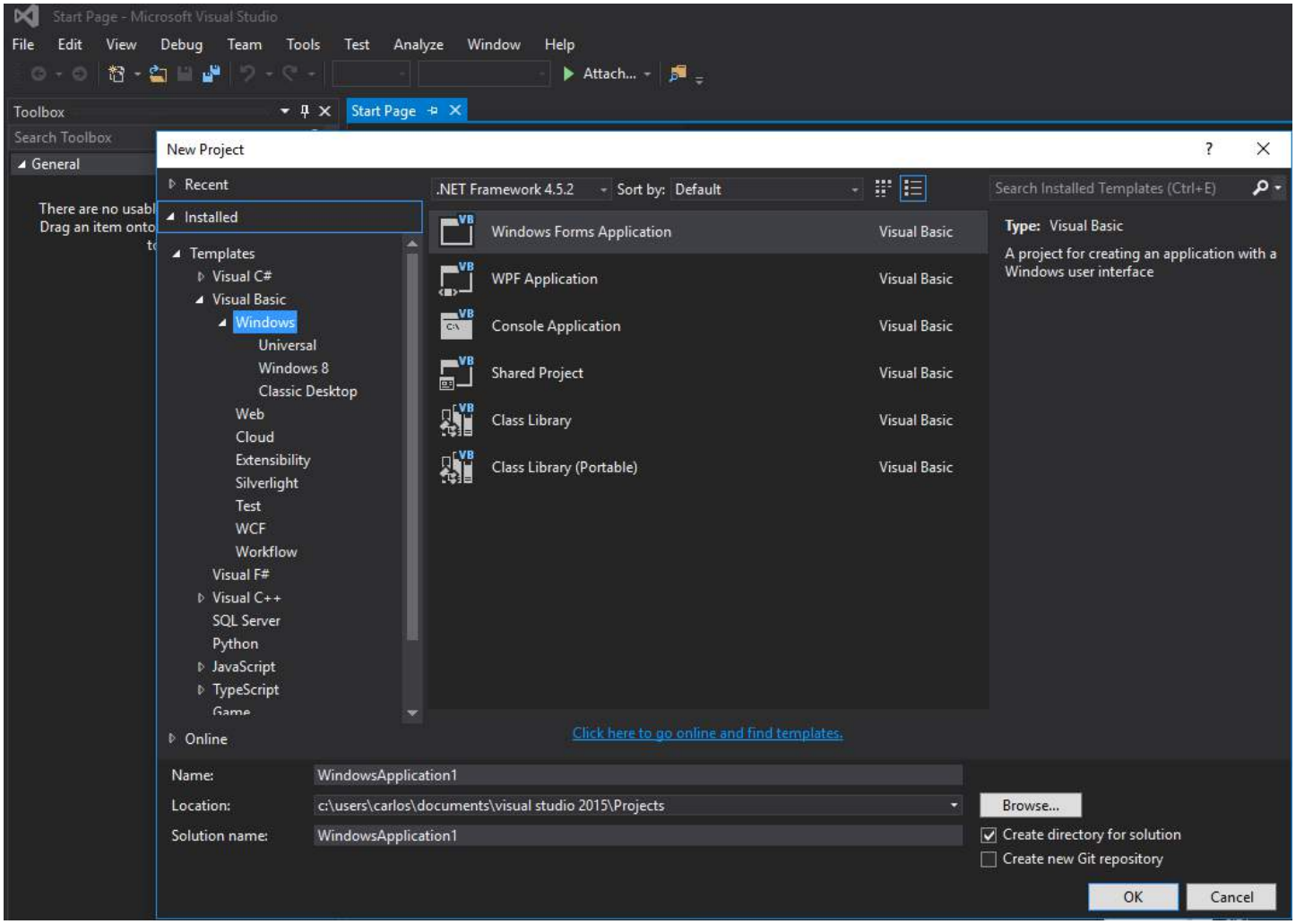
Chapter 39: Google Maps in a Windows Form

Section 39.1: How to use a Google Map in a Windows Form

The first part of this example explains how to implement it. In the second, I will explain how it works. This tries to be a general example. The template for the map (see step 3) and the example functions are fully customizable.

########## IMPLEMENTATION ##########

Step 1. Firstly, create a new project and select Windows Form Application. Let's leave its name as "Form1".



Step 2. Add a WebBrowser control (which will hold your map) to your Form1. Let's call it "wbmap"

Step 3. Create a .html file named "googlemap_template.html" with your favourite text editor and paste the following code:

googlemap_template.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
     <style type="text/css">
      html, body {
```

```
        height: 100%;
        margin: 0;
        padding: 0;
      }
      #gmap {
        height: 100%;
      }
    </style>
    <script type="text/javascript" src="http://maps.google.com/maps/api/js?sensor=false"></script>
    <script type="text/javascript">
        function initialize() {
            //Use window.X instead of var X to make a variable globally available
            window.markers = new Array();
            window.marker_data = [[MARKER_DATA]];
            window.gmap = new google.maps.Map(document.getElementById('gmap'), {
            zoom: 15,
            center: new google.maps.LatLng(marker_data[0][0], marker_data[0][1]),
            mapTypeId: google.maps.MapTypeId.ROADMAP
          });
          var infowindow = new google.maps.InfoWindow();
          var newmarker, i;
          for (i = 0; i < marker_data.length; i++) {
              if (marker_data[0].length == 2) {
                  newmarker = new google.maps.Marker({
                      position: new google.maps.LatLng(marker_data[i][0], marker_data[i][1]),
                      map: gmap
                  });
              } else if (marker_data[0].length == 3) {
                  newmarker = new google.maps.Marker({
                      position: new google.maps.LatLng(marker_data[i][0], marker_data[i][1]),
                      map: gmap,
                      title: (marker_data[i][2])
                  });
              } else {
                  newmarker = new google.maps.Marker({
                      position: new google.maps.LatLng(marker_data[i][0], marker_data[i][1]),
                      map: gmap,
                      title: (marker_data[i][2]),
                      icon: (marker_data[i][3])
                  });
              }
            google.maps.event.addListener(newmarker, 'click', (function (newmarker, i) {
                return function () {
                    if (newmarker.title) {
                        infowindow.setContent(newmarker.title);
                        infowindow.open(gmap, newmarker);
                    }
                    gmap.setCenter(newmarker.getPosition());
                    // Calling functions written in the WF
                    window.external.showVbHelloWorld();
                    window.external.getMarkerDataFromJavascript(newmarker.title,i);
                }
            })(newmarker, i));
            markers[i] = newmarker;
          }
        }
        google.maps.event.addDomListener(window, 'load', initialize);
    </script>
    <script type="text/javascript">
        // Function triggered from the WF with no arguments
        function showJavascriptHelloWorld() {
            alert("Hello world in HTML from WF");
```


www.dbooks.org

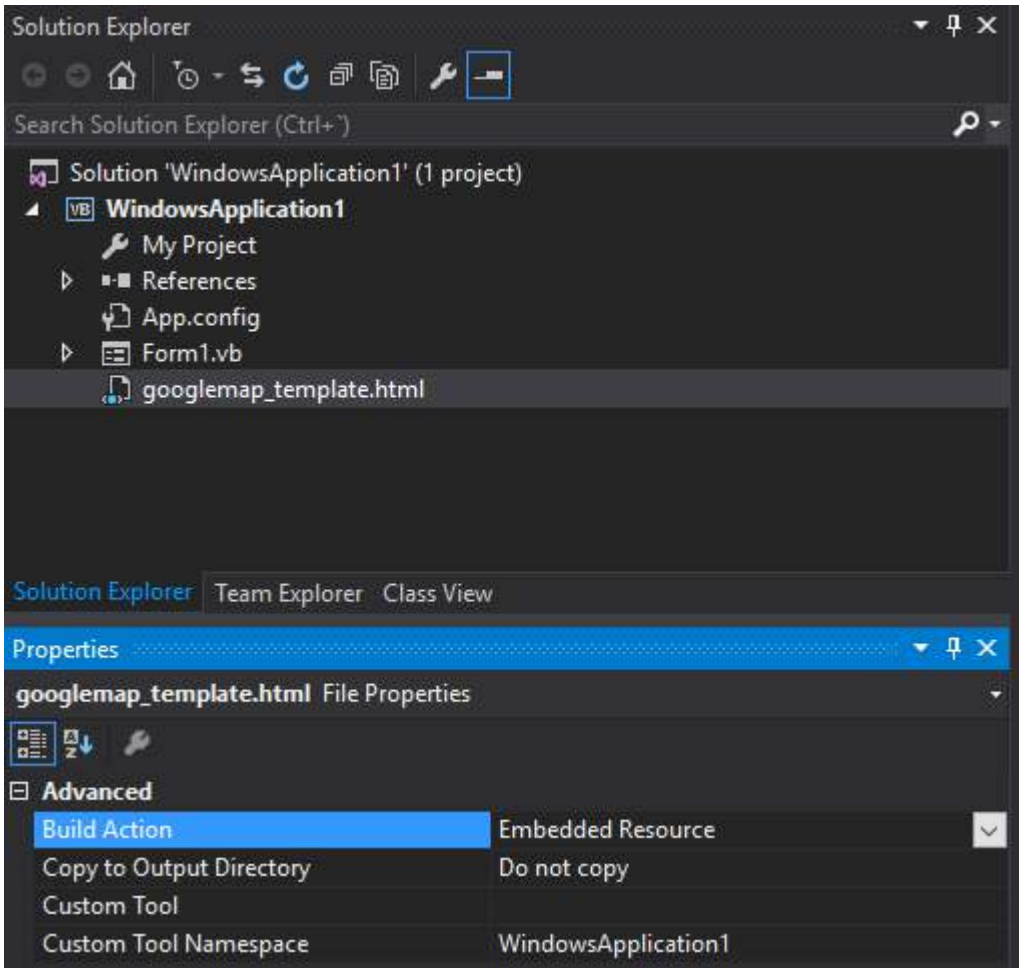
```
        }
    </script>
     <script type="text/javascript">
       // Function triggered from the WF with a String argument
       function focusMarkerFromIdx(idx) {
           google.maps.event.trigger(markers[idx], 'click');
       }
     </script>
  </head>
  <body>
    <div id="gmap"></div>
  </body>
</html>
```

This will serve as our map template. I will explain how it works later.

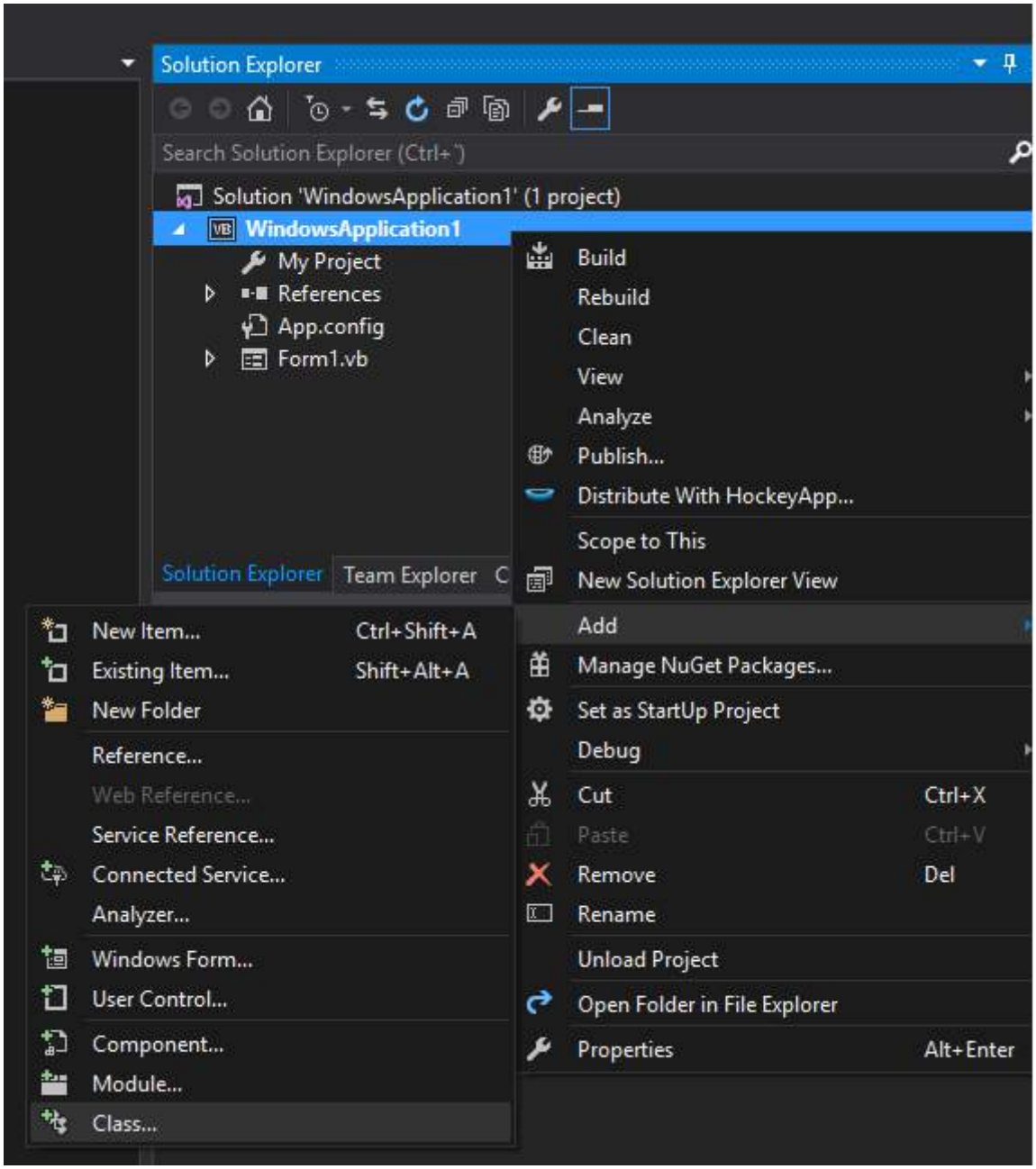
Step 4. Add the googlemap_template.hmtl file to your project (right click on your project->add->existing item)

Step 5. Once it appears in your Solution Explorer, set its properties to:

- Build Action -> Embedded Resource
- Custom Tool Namespace -> write the name of the project



Step 6. Add a new class (right click on your project->add->class). In my example I'll call it GoogleMapHelper.



Step 7. Paste the following code into your class:

GoogleMapHelper.vb

```
    Imports System.IO
    Imports System.Reflection
    Imports System.Text

    Public Class GoogleMapHelper

    ' 1- googlemap_template.html must be copied in the main project folder
    ' 2- add the file into the Visual Studio Solution Explorer (add existing file)
    ' 3- set the properties of the file to:
    '                                   Build Action -> Embedded Resource
    '                                   Custom Tool Namespace -> write the name of the project

    Private Const ICON_FOLDER As String = "marker_icons/" 'images must be stored in a folder inside
Debug/Release folder
    Private Const MAP_TEMPLATE As String = "WindowsApplication1.googlemap_template.html"
    Private Const TEXT_TO_REPLACE_MARKER_DATA As String = "[[MARKER_DATA]]"
    Private Const TMP_NAME As String = "tmp_map.html"
```

```
    Private mWebBrowser As WebBrowser

    'MARKER POSITIONS
    Private mPositions As Double(,) 'lat, lon
    ' marker data allows different formats to include lat,long and optionally title and icon:
    ' op1: mMarkerData = New String(N-1, 1) {{lat1, lon1}, {lat2, lon2}, {latN, lonN}}
    ' op2: mMarkerData = New String(N-1, 2) {{lat1, lon1,'title1'}, {lat2, lon2,'title2'}, {latN,
lonN, 'titleN'}}
    ' op3: mMarkerData = New String(N-1, 3) {{lat1, lon1,'title1','image1.png'}, {lat2,
lon2,'title2','image2.png'}, {latN, lonN, 'titleN','imageN.png'}}
    Private mMarkerData As String(,) = Nothing


    Public Sub New(ByRef wb As WebBrowser, pos As Double(,))
        mWebBrowser = wb
        mPositions = pos
        mMarkerData = getMarkerDataFromPositions(pos)
    End Sub

    Public Sub New(ByRef wb As WebBrowser, md As String(,))
        mWebBrowser = wb
        mMarkerData = md
    End Sub

    Public Sub loadMap()
        mWebBrowser.Navigate(getMapTemplate())
    End Sub

    Private Function getMapTemplate() As String

        If mMarkerData Is Nothing Or mMarkerData.GetLength(1) > 4 Then
            MessageBox.Show("Marker data has not the proper size. It must have 2, 3 o 4 columns")
            Return Nothing
        End If

        Dim htmlTemplate As New StringBuilder()
        Dim tmpFolder As String = Environment.GetEnvironmentVariable("TEMP")
        Dim dataSize As Integer = mMarkerData.GetLength(1) 'number of columns
        Dim mMarkerDataAsText As String = String.Empty
        Dim myresourcePath As String = My.Resources.ResourceManager.BaseName
        Dim myresourcefullPath As String = Path.GetFullPath(My.Resources.ResourceManager.BaseName)
        Dim localPath = myresourcefullPath.Replace(myresourcePath, "").Replace("\", "/") &
ICON_FOLDER

        htmlTemplate.AppendLine(getStringFromResources(MAP_TEMPLATE))
        mMarkerDataAsText = "["

        For i As Integer = 0 To mMarkerData.GetLength(0) - 1
            If i <> 0 Then
                mMarkerDataAsText += ","
            End If
            If dataSize = 2 Then 'lat,lon
                mMarkerDataAsText += "[" & mMarkerData(i, 0) & "," + mMarkerData(i, 1) & "]"
            ElseIf dataSize = 3 Then 'lat,lon and title
                mMarkerDataAsText += "[" & mMarkerData(i, 0) & "," + mMarkerData(i, 1) & ",'" &
mMarkerData(i, 2) & "']"
            ElseIf dataSize = 4 Then 'lat,lon,title and image
                mMarkerDataAsText += "[" & mMarkerData(i, 0) & "," + mMarkerData(i, 1) & ",'" &
mMarkerData(i, 2) & "','" & localPath & mMarkerData(i, 3) & "']" 'Ojo a las comillas simples en las
columnas 3 y 4
```

```
            End If
        Next

        mMarkerDataAsText += "]"
        htmlTemplate.Replace(TEXT_TO_REPLACE_MARKER_DATA, mMarkerDataAsText)

        Dim tmpHtmlMapFile As String = (tmpFolder & Convert.ToString("\")) + TMP_NAME
        Dim existsMapFile As Boolean = False
        Try
            existsMapFile = createTxtFile(tmpHtmlMapFile, htmlTemplate)
        Catch ex As Exception
            MessageBox.Show("Error writing temporal file", "Writing Error", MessageBoxButtons.OK,
MessageBoxIcon.[Error])
        End Try

        If existsMapFile Then
            Return tmpHtmlMapFile
        Else
            Return Nothing
        End If
    End Function

    Private Function getMarkerDataFromPositions(pos As Double(,)) As String(,)
        Dim md As String(,) = New String(pos.GetLength(0) - 1, 1) {}
        For i As Integer = 0 To pos.GetLength(0) - 1
            md(i, 0) = pos(i, 0).ToString("g", New System.Globalization.CultureInfo("en-US"))
            md(i, 1) = pos(i, 1).ToString("g", New System.Globalization.CultureInfo("en-US"))
        Next
        Return md
    End Function

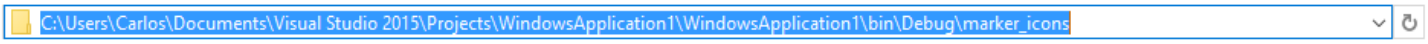
    Private Function getStringFromResources(resourceName As String) As String
        Dim assem As Assembly = Me.[GetType]().Assembly

        Using stream As Stream = assem.GetManifestResourceStream(resourceName)
            Try
                Using reader As New StreamReader(stream)
                    Return reader.ReadToEnd()
                End Using
            Catch e As Exception
                Throw New Exception((Convert.ToString("Error de acceso al Recurso '") &
resourceName) + "'" & vbCr & vbLf + e.ToString())
            End Try
        End Using
    End Function

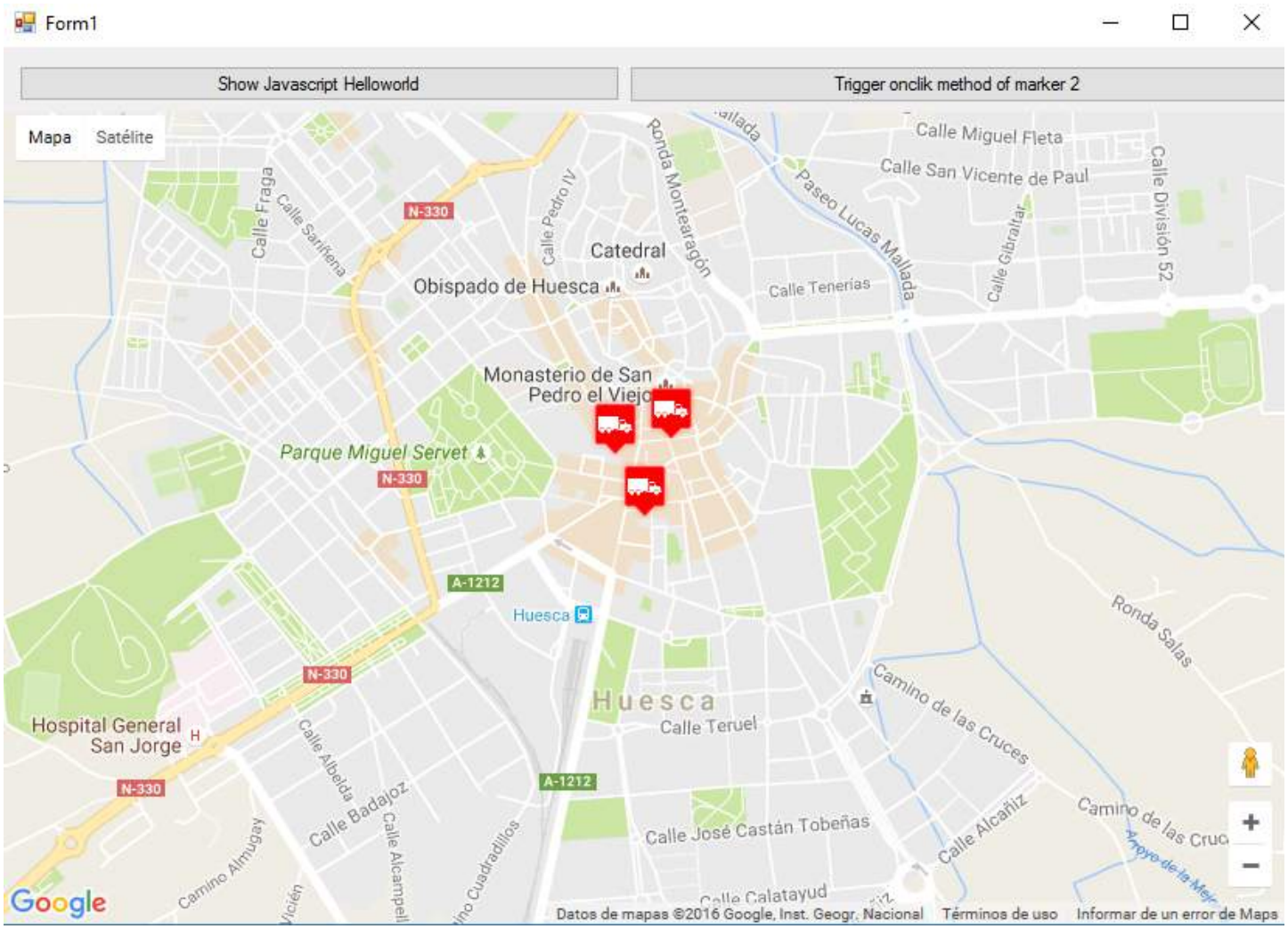
    Private Function createTxtFile(mFile As String, content As StringBuilder) As Boolean
        Dim mPath As String = Path.GetDirectoryName(mFile)
        If Not Directory.Exists(mPath) Then
            Directory.CreateDirectory(mPath)
        End If
        If File.Exists(mFile) Then
            File.Delete(mFile)
        End If
        Dim sw As StreamWriter = File.CreateText(mFile)
        sw.Write(content.ToString())
        sw.Close()
        Return True
    End Function
    End Class
```

Note: The MAP_TEMPLATE constant must include the name of your project

Step 8. Now we can use our GoogleMapHelper class to load the map into our webbrowser by simply creating and instance and calling its loadMap() method. How you build your markerData is up to you. In this example, for clarification, I write them by hand. There are 3 options to define the marker data (see GoogleMapHelper class comments). Note that if you use the third option (including title and icons) you must create a folder called "marker_icons" (or whatever you define in the GoogleMapHelper constant ICON_FOLDER) in your Debug/Release folder and place there your .png files. In my case:



I created two buttons in my Form1 to illustrate how the map and the WF interact. Here is how it looks:



And here is the code:

Form1.vb

```
Imports System.IO
Imports System.Reflection
Imports System.Security.Permissions
Imports System.Text
<PermissionSet(SecurityAction.Demand, Name:="FullTrust")>
<System.Runtime.InteropServices.ComVisible(True)>
Public Class Form1

Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load

    Me.wbmap.ObjectForScripting = Me
```

```
    Dim onlyPositions As Double(,) = New Double(2, 1) {{42.13557, -0.40806}, {42.13684, -0.40884},
{42.13716, -0.40729}}
    Dim positonAndTitles As String(,) = New String(2, 2) {{"42.13557", "-0.40806", "marker0"},
{"42.13684", "-0.40884", "marker1"}, {"42.13716", "-0.40729", "marker2"}}
    Dim positonTitlesAndIcons As String(,) = New String(2, 3) {{"42.13557", "-0.40806", "marker0",
"truck_red.png"}, {"42.13684", "-0.40884", "marker1", "truck_red.png"}, {"42.13716", "-0.40729",
"marker2", "truck_red.png"}}

    'Dim gmh As GoogleMapHelper = New GoogleMapHelper(wbmap, onlyPositions)
    'Dim gmh As GoogleMapHelper = New GoogleMapHelper(wbmap, positonAndTitles)
    Dim gmh As GoogleMapHelper = New GoogleMapHelper(wbmap, positonTitlesAndIcons)
    gmh.loadMap()
End Sub

'################################ CALLING JAVASCRIPT METHODS ##############################
'This methods call methods written in googlemap_template.html
Private Sub callMapJavascript(sender As Object, e As EventArgs) Handles Button1.Click
    wbmap.Document.InvokeScript("showJavascriptHelloWorld")
End Sub

Private Sub callMapJavascriptWithArguments(sender As Object, e As EventArgs) Handles Button2.Click
    wbmap.Document.InvokeScript("focusMarkerFromIdx", New String() {2})
End Sub
'##########################################################################################

'################################ METHODS CALLED FROM JAVASCRIPT ##########################
'This methods are called by the javascript defined in googlemap_template.html when some events are
triggered
Public Sub getMarkerDataFromJavascript(title As String, idx As String)
    MsgBox("Title: " & title & " idx: " & idx)
End Sub

Public Sub showVbHelloWorld()
    MsgBox("Hello world in WF from HTML")
End Sub
End Class
```

IMPORTANT : don't forget to add these lines before your class Form1 definition:

```
<PermissionSet(SecurityAction.Demand, Name:="FullTrust")>
<System.Runtime.InteropServices.ComVisible(True)>
```

What they do is to tell the .NET Framework that we want fulltrust and make the class visible to COM so Form1 is visible to JavaScript.

Also don't forget this in your Form1 load function:

```
Me.wbmap.ObjectForScripting = Me
```

It exposes your Form1 class to the JavaScript on the googlemap_template.hmtl page.

Now you can execute and it should be working

################################## HOW IT WORKS##################################

Basically, what our GoogleMapHelper class does is to read our googlemap_template.html, make a temporal copy, replace the code related to the markers ([[MARKER_DATA]]) and execute the page in the web browser control of our form. This html loops through all the markers and assigns a 'click' listener to each one. This click function is obviously fully customizable. In the example it opens an infowindow if the marker has a title, centers the map in

www.dbooks.org

such marker and calls two external functions that are defined in our Form1 class.

On the other hand, we can define other javascript functions (with or without arguments) in this html to be called from our Windows Form (by using wbmap.Document.InvokeScript).

Chapter 40: GDI+

Section 40.1: Draw Shapes

To start drawing a shape you need to define a pen object The `Pen` accepts two parameters:

1. Pen Color or Brush
2. Pen Width

The Pen Object is used to create an **outline** of the object you want to draw

After Defining the Pen you can set specific Pen Properties

```
   Dim pens As New Pen(Color.Purple)
   pens.DashStyle = DashStyle.Dash 'pen will draw with a dashed line
   pens.EndCap = LineCap.ArrowAnchor 'the line will end in an arrow
   pens.StartCap = LineCap.Round 'The line draw will start rounded
   '*Notice* - the Start and End Caps will not show if you draw a closed shape
```

Then use the graphics object you created to draw the shape

```
  Private Sub GraphicForm_Paint(sender As Object, e As PaintEventArgs) Handles MyBase.Paint
    Dim pen As New Pen(Color.Blue, 15) 'Use a blue pen with a width of 15
    Dim point1 As New Point(5, 15) 'starting point of the line
    Dim point2 As New Point(30, 100) 'ending point of the line
    e.Graphics.DrawLine(pen, point1, point2)

    e.Graphics.DrawRectangle(pen, 60, 90, 200, 300) 'draw an outline of the rectangle
```

By default, the pen's width is equal to 1

```
    Dim pen2 as New Pen(Color.Orange) 'Use an orange pen with width of 1
    Dim origRect As New Rectangle(90, 30, 50, 60) 'Define bounds of arc
    e.Graphics.DrawArc(pen2, origRect, 20, 180) 'Draw arc in the rectangle bounds

End Sub
```

Section 40.2: Fill Shapes

Graphics.FillShapes draws a shape and fills it in with the color given. Fill Shapes can use

1. `Brush` Tool - to fill shape with a solid color

   ```
   Dim rect As New Rectangle(50, 50, 50, 50)
   e.Graphics.FillRectangle(Brushes.Green, rect) 'draws a rectangle that is filled with green

   e.Graphics.FillPie(Brushes.Silver, rect, 0, 180) 'draws a half circle that is filled with
   silver
   ```

2. `HatchBrush` Tool - to fill shape with a pattern

```
Dim hBrush As New HatchBrush(HatchStyle.ZigZag, Color.SkyBlue, Color.Gray)
'creates a HatchBrush Tool with a background color of blue, foreground color of gray,
'and will fill with a zigzag pattern
Dim rectan As New Rectangle(100, 100, 100, 100)
```


www.dbooks.org

```
e.Graphics.FillRectangle(hBrush, rectan)
```

3. `LinearGradientBrush` - to fill shape with a gradient

```
Dim lBrush As New LinearGradientBrush(point1, point2, Color.MediumVioletRed, Color.PaleGreen)
 Dim rect As New Rectangle(50, 50, 200, 200)
 e.Graphics.FillRectangle(lBrush, rect)
```

4. `TextureBrush` - to fill shape with a picture

You can choose a picture from resources, an already defined Bitmap, or from a file name

```
 Dim textBrush As New TextureBrush(New Bitmap("C:\ColorPic.jpg"))
  Dim rect As New Rectangle(400, 400, 100, 100)
  e.Graphics.FillPie(textBrush, rect, 0, 360)
```

Both the `Hatch Brush Tool` and `LinearGradientBrush` import the following statement : **Imports System.Drawing.Drawing2D**

Section 40.3: Text

To draw text onto the form use the `DrawString` Method

When you draw a string you can use any of the 4 brushes listed above

```
Dim lBrush As New LinearGradientBrush(point1, point2, Color.MediumVioletRed, Color.PaleGreen)
e.Graphics.DrawString("HELLO", New Font("Impact", 60, FontStyle.Bold), lBrush, New Point(40, 400))
'this will draw the word "Hello" at the given point, with a linearGradient Brush
```

Since you can't define the width or height of the text use `Measure Text` to check text size

```
Dim lBrush As New LinearGradientBrush(point1, point2, Color.MediumVioletRed, Color.PaleGreen)
Dim TextSize =  e.Graphics.MeasureString("HELLO", New Font("Impact", 60, FontStyle.Bold), lBrush)
'Use the TextSize to determine where to place the string, or if the font needs to be smaller
```

> Ex: You need to draw the word "Test" on top of the form. The form's width is 120. Use this loop to decrease the font size till it will fit into the forms width

```
Dim FontSize as Integer = 80
Dim TextSize = e.graphics.measeString("Test", New Font("Impact",FontSize, FontStyle.Bold), new
Brush(colors.Blue, 10)
Do while TextSize.Width >120
FontSize = FontSize -1
TextSize = e.graphics.measeString("Test", New Font("Impact",FontSize, FontStyle.Bold), new
Brush(colors.Blue, 10)
Loop
```

Section 40.4: Create Graphic Object

There are three ways to create a graphics object

1. From the **Paint Event**

Every time the control is redrawn (resized, refreshed...) this event is called, use this way if you want the control to

consistently draw on the control

```
'this will work on any object's paint event, not just the form
Private Sub Form1_Paint(sender as Object, e as PaintEventArgs) Handles Me.Paint
  Dim gra as Graphics
  gra = e.Graphics
End Sub
```

2. **Create Graphic**

This is most often used when you want to create a one time graphic on the control, or you don't want the control to repaint itself

```
Dim btn as New Button
Dim g As Graphics = btn.CreateGraphics
```

3. From an **Existing Graphic**

Use this method when you want to draw and change an existing graphic

```
'The existing image can be from a filename, stream or Drawing.Graphic
Dim image = New Bitmap("C:\TempBit.bmp")
Dim gr As Graphics = Graphics.FromImage(image)
```

Chapter 41: WinForms SpellCheckBox

Example on how to add a spell check box to a WindowsForms application. This example DOES NOT require Word to be installed nor does it use Word in any way.

It uses WPF Interop using the ElementHost control to create a WPF UserControl from a WPF TextBox. WPF TextBox has a built in function for spell check. We are going to leverage this built in function rather than relying on an external program.

Section 41.1: ElementHost WPF TextBox

This example is was modeled after an example that I found on the internet. I can't find the link or I would give the author credit. I took the sample that I found and modified it to work for my application.

1. Add the following references:

> System.Xaml, PresentationCore, PresentationFramework, WindowsBase, and WindowsFormsIntegration

2. Create a new Class and past this code

```
Imports System
Imports System.ComponentModel
Imports System.ComponentModel.Design.Serialization
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Forms.Integration
Imports System.Windows.Forms.Design

<Designer(GetType(ControlDesigner))> _
Class SpellCheckBox
Inherits ElementHost

Private box As TextBox

Public Sub New()
    box = New TextBox()
    MyBase.Child = box
    AddHandler box.TextChanged, AddressOf box_TextChanged
    box.SpellCheck.IsEnabled = True
    box.VerticalScrollBarVisibility = ScrollBarVisibility.Auto
    Me.Size = New System.Drawing.Size(100, 20)
End Sub

Private Sub box_TextChanged(ByVal sender As Object, ByVal e As EventArgs)
    OnTextChanged(EventArgs.Empty)
End Sub

<DefaultValue("")> _
Public Overrides Property Text() As String
    Get
        Return box.Text
    End Get
    Set(ByVal value As String)
        box.Text = value
    End Set
End Property
```

```
<DefaultValue(True)> _
Public Property MultiLine() As Boolean
    Get
        Return box.AcceptsReturn
    End Get
    Set(ByVal value As Boolean)
        box.AcceptsReturn = value
    End Set
End Property

<DefaultValue(True)> _
Public Property WordWrap() As Boolean
    Get
        Return box.TextWrapping <> TextWrapping.Wrap
    End Get
    Set(ByVal value As Boolean)
        If value Then
            box.TextWrapping = TextWrapping.Wrap
        Else
            box.TextWrapping = TextWrapping.NoWrap
        End If
    End Set
End Property

<DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)> _
Public Shadows Property Child() As System.Windows.UIElement
    Get
        Return MyBase.Child
    End Get
    Set(ByVal value As System.Windows.UIElement)
        '' Do nothing to solve a problem with the serializer !!
    End Set
End Property

End Class
```

3. Rebuild the solution.
4. Add a new form.
5. Search the toolbox for your Class name. This example is "SpellCheck". It should be listed under 'YourSoulutionName' Components.
6. Drag the new control to your form
7. Set any of the mapped properties in the forms load event

```
Private Sub form1_Load(sender As Object, e As EventArgs) Handles Me.Load
    spellcheckbox.WordWrap = True
    spellcheckbox.MultiLin = True
    'Add any other property modifiers here...
End Sub
```

7. The last thing that you need to do is to change the DPI Awareness of your application. This is because you are using WinForms application. By default all WinForms applications are DPI UNAWARE. Once you execute a control that has an element host (WPF Interop), the application will now become DPI AWARE. This may or may not mess with your UI Elements. The solution to this is to FORCE the application to become DPI UNAWARE. There are 2 ways to do this. The first is through the manifest file and the second is to hard code it in to your program. If you are using OneClick to deploy your application, you must hard code it, not use the manifest file or errors will be inevitable.

Both of the following examples can be found at the following: [WinForms Scaling at Large DPI Settings - Is It Even Possible?] Thanks to Telerik.com for the great explanation on DPI.

> Hard coded DPI Aware code example. This MUST be executed before the first form is initialized. I always place this in the ApplicationEvents.vb file. You can get to this file by right clicking on your project name in the solution explorer and choosing "Open". Then choose the application tab on the left and then click on "View Application Events" on the lower right next to the splash screen drop down.

```
Namespace My

    ' The following events are available for MyApplication:
    '
    ' Startup: Raised when the application starts, before the startup form is created.
    ' Shutdown: Raised after all application forms are closed.  This event is not raised if the
application terminates abnormally.
    ' UnhandledException: Raised if the application encounters an unhandled exception.
    ' StartupNextInstance: Raised when launching a single-instance application and the application is
already active.
    ' NetworkAvailabilityChanged: Raised when the network connection is connected or disconnected.
    Partial Friend Class MyApplication

    Private Enum PROCESS_DPI_AWARENESS
        Process_DPI_Unaware = 0
        Process_System_DPI_Aware = 1
        Process_Per_Monitor_DPI_Aware = 2
    End Enum

    Private Declare Function SetProcessDpiAwareness Lib "shcore.dll" (ByVal Value As
PROCESS_DPI_AWARENESS) As Long

    Private Sub SetDPI()
        'Results from SetProcessDPIAwareness
        'Const S_OK = &H0&
        'Const E_INVALIDARG = &H80070057
        'Const E_ACCESSDENIED = &H80070005

        Dim lngResult As Long

        lngResult = SetProcessDpiAwareness(PROCESS_DPI_AWARENESS.Process_DPI_Unaware)

    End Sub

    Private Sub MyApplication_Startup(sender As Object, e As ApplicationServices.StartupEventArgs)
Handles Me.Startup
        SetDPI()
    End Sub

End Namespace
```

> Manifest Example

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" xmlns:asmv3="urn:schemas-
microsoft-com:asm.v3" >
   <asmv3:application>
        <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
             <dpiAware>true</dpiAware>
```

```
        </asmv3:windowsSettings>
    </asmv3:application>
</assembly>
```

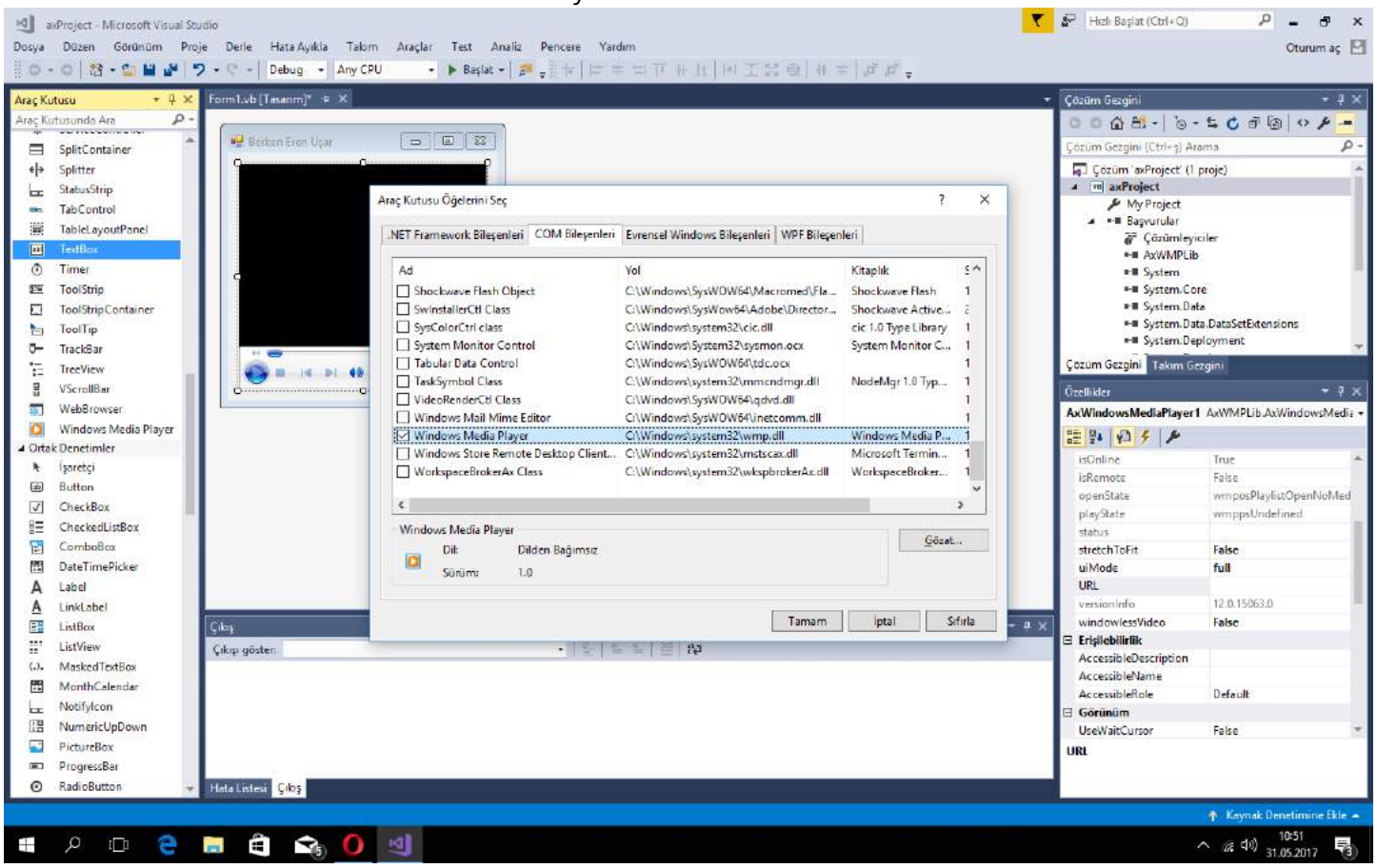
Chapter 42: Using axWindowsMediaPlayer in VB.Net

axWindowsMediaPlayer is the control for the playing multimedia files like videos and music.

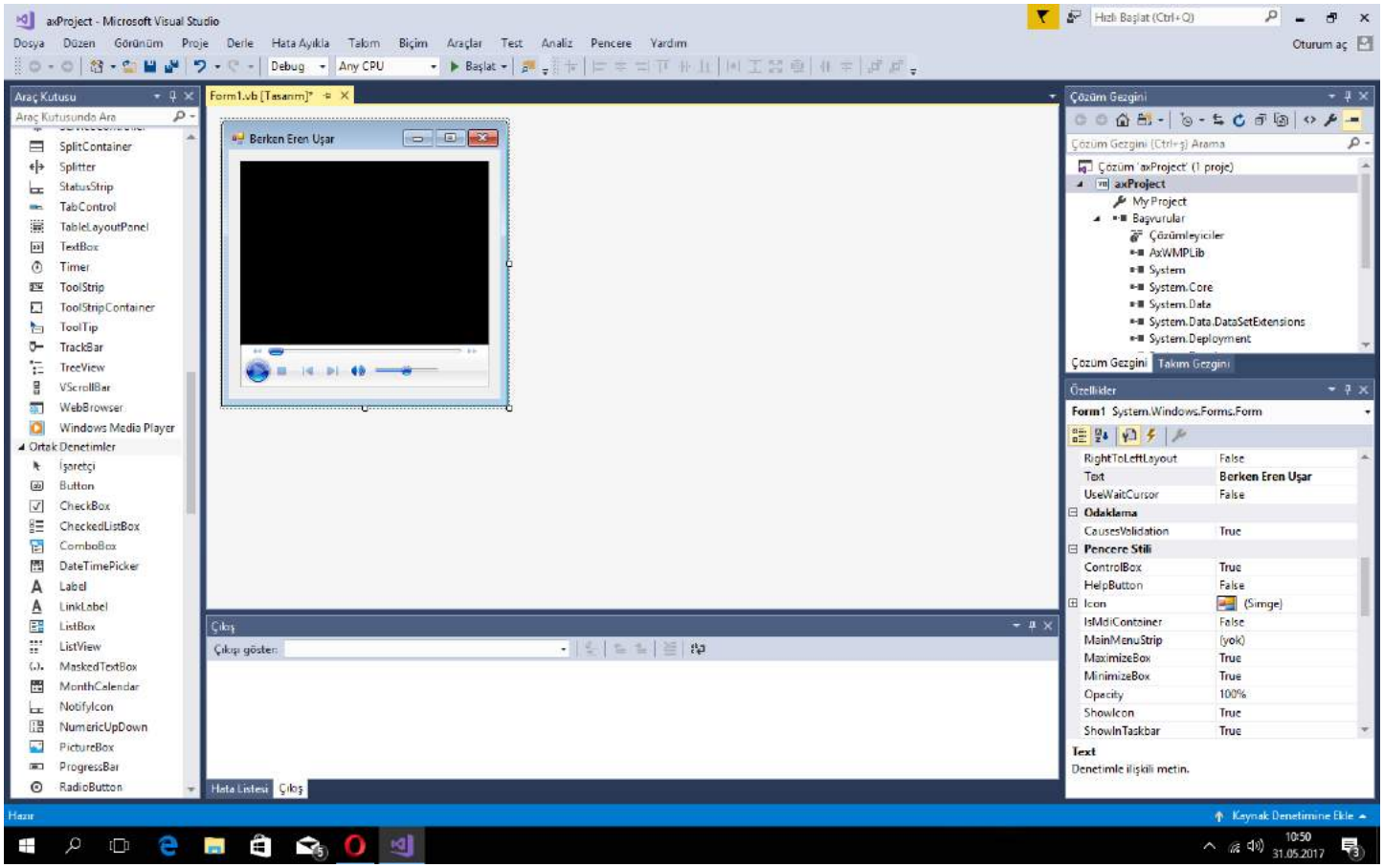
Section 42.1: Adding the axWindowsMediaPlayer

- Right-click on the Toolbox, then click "Choose Items".
- Select the COM Components tab, and then check Windows Media Player.
- axWindowsMediaPlayer will be added to Toolbox.

Select this checkbox to use axWindowsMediaPlayer



Then you can use axWindowsMediaPlayer :)



Section 42.2: Play a Multimedia File

```
AxWindowsMediaPlayer1.URL = "C:\My Files\Movies\Avatar.mp4"
AxWindowsMediaPlayer1.Ctlcontrols.play()
```

This code will play Avatar in the axWindowsMediaPlayer.

Chapter 43: WPF XAML Data Binding

This example shows how to create a ViewModel and a View within the MVVM pattern and WPF, and how to bind the two together, so that each is updated whenever the other is changed.

Section 43.1: Binding a String in the ViewModel to a TextBox in the View

SampleViewModel.vb

```
'Import classes related to WPF for simplicity
Imports System.Collections.ObjectModel
Imports System.ComponentModel

Public Class SampleViewModel
    Inherits DependencyObject
    'A class acting as a ViewModel must inherit from DependencyObject

    'A simple string property
    Public Property SampleString as String
        Get
            Return CType(GetValue(SampleStringProperty), String)
        End Get

        Set(ByVal value as String)
            SetValue(SampleStringProperty, value)
        End Set
    End Property

    'The DependencyProperty that makes databinding actually work
    'for the string above
    Public Shared ReadOnly SampleStringProperty As DependencyProperty = _
                           DependencyProperty.Register("SampleString", _
                           GetType(String), GetType(SampleViewModel), _
                           New PropertyMetadata(Nothing))

End Class
```

A DependencyProperty can be easily added by using the `wpfdp` code snippet (type `wpfdp`, then press the `TAB` key twice), however, the code snippet is not type safe, and will not compile under `Option Strict On`.

SampleWindow.xaml

```
<Window x:Class="SampleWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:des="http://schemas.microsoft.com/expression/blend/2008"
       DataContext="{Binding}"
       Loaded="Window_Loaded">
    <Grid>
        <TextBox>
            <TextBox.Text>
                <Binding Path="SampleString" />
            </TextBox.Text>
        </TextBox>
    </Grid>
</Window>
```

SampleWindow.xaml.vb

```
Class SampleWindow

    Private WithEvents myViewModel As New SampleViewModel()

    Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
        Me.DataContext = myViewModel
    End Sub
End Class
```

Note that this is a very rudimentary way to implement MVVM and databinding. A more robust practice would be to use a platform like Unity to "inject" the ViewModel into the View.

Chapter 44: Reading compressed textfile on-the-fly

Section 44.1: Reading .gz textfile line after line

This class open a .gz file (usual format of compressed log files) and will return a line at each call of `.NextLine()`

There is no memory usage for temporary decompression, very useful for large file.

```
Imports System.IO

Class logread_gz

  Private ptr As FileStream
  Private UnGZPtr As Compression.GZipStream
  Private line_ptr As StreamReader
  Private spath As String

  Sub New(full_filename As String)
    spath = full_filename
  End Sub

  Sub Open()
     Me.ptr = File.OpenRead(spath)
     Me.UnGZPtr = New Compression.GZipStream(ptr, Compression.CompressionMode.Decompress)
     Me.line_ptr = New StreamReader(UnGZPtr)
  End Sub()

  Function NextLine() As String
    'will return Nothing if EOF
    Return Me.line_ptr.ReadLine()
  End Function

  Sub Close()
    Me.line_ptr.Close()
    Me.line_ptr.Dispose()
    Me.UnGZPtr.Close()
    Me.UnGZPtr.Dispose()
    Me.ptr.Close()
    Me.ptr.Dispose()
  End Sub

End Class
```

Note : there is no failsafe, for readbility purpose.

Chapter 45: Threading

Section 45.1: Performing thread-safe calls using Control.Invoke()

Using the `Control.Invoke()` method you may move the execution of a method or function from a background thread to the thread that the control was created on, which is usually the UI (User Interface) thread. By doing so your code will be queued to run on the control's thread instead, which removes the possibility of concurrency.

The `Control.InvokeRequired` property should also be checked in order to determine whether you need to invoke, or if the code is already running on the same thread as the control.

The `Invoke()` method takes a delegate as its first parameter. A delegate holds the reference, parameter list and return type to another method.

In Visual Basic 2010 (10.0) or higher, *lambda expressions* can be used to create a delegate method on the fly:

```
If LogTextBox.InvokeRequired = True Then
    LogTextBox.Invoke(Sub() LogTextBox.AppendText("Check passed"))
Else
    LogTextBox.AppendText("Check passed")
End If
```

Whereas in Visual Basic 2008 (9.0) or lower, you have to declare the delegate on your own:

```
Delegate Sub AddLogText(ByVal Text As String)

If LogTextBox.InvokeRequired = True Then
    LogTextBox.Invoke(New AddLogText(AddressOf UpdateLog), "Check passed")
Else
    UpdateLog("Check passed")
End If

Sub UpdateLog(ByVal Text As String)
    LogTextBox.AppendText(Text)
End Sub
```

Section 45.2: Performing thread-safe calls using Async/Await

If we try to change an object on the UI thread from a different thread we will get a cross-thread operation exception:

```
Private Sub Button_Click(sender As Object, e As EventArgs) Handles MyButton.Click
    ' Cross thread-operation exception as the assignment is executed on a different thread
    ' from the UI one:
    Task.Run(Sub() MyButton.Text = Thread.CurrentThread.ManagedThreadId)
End Sub
```

Before **VB 14.0** and **.NET 4.5** the solution was invoking the assignment on and object living on the UI thread:

```
Private Sub Button_Click(sender As Object, e As EventArgs) Handles MyButton.Click
    ' This will run the conde on the UI thread:
    MyButton.Invoke(Sub() MyButton.Text = Thread.CurrentThread.ManagedThreadId)
End Sub
```

With **VB 14.0**, we can run a `Task` on a different thread and then have the context restored once the execution is complete and then perform the assignment with Async/Await:

```
Private Async Sub Button_Click(sender As Object, e As EventArgs) Handles MyButton.Click
    ' This will run the code on a different thread then the context is restored
    ' so the assignment happens on the UI thread:
    MyButton.Text = Await Task.Run(Function() Thread.CurrentThread.ManagedThreadId)
End Sub
```

Chapter 46: Multithreading

Section 46.1: Multithreading using Thread Class

This example uses the `Thread` Class, but multithreaded applications can also be made using `BackgroundWorker`. The `AddNumber`, `SubstractNumber`, and `DivideNumber` functions will be executed by separate threads:

Edit: Now the UI thread waits for the child threads to finish and shows the result.

```
Module Module1
    'Declare the Thread and assign a sub to that
    Dim AddThread As New Threading.Thread(AddressOf AddNumber)
    Dim SubstractThread As New Threading.Thread(AddressOf SubstractNumber)
    Dim DivideThread As New Threading.Thread(AddressOf DivideNumber)

    'Declare the variable for holding the result
    Dim addResult As Integer
    Dim SubStractResult As Integer
    Dim DivisionResult As Double

    Dim bFinishAddition As Boolean = False
    Dim bFinishSubstration As Boolean = False
    Dim bFinishDivision As Boolean = False

    Dim bShownAdditionResult As Boolean = False
    Dim bShownDivisionResult As Boolean = False
    Dim bShownSubstractionResult As Boolean = False

    Sub Main()

        'Now start the trheads
        AddThread.Start()
        SubstractThread.Start()
        DivideThread.Start()

        'Wait and display the results in console
        Console.WriteLine("Waiting for threads to finish...")
        Console.WriteLine("")

        While bFinishAddition = False Or bFinishDivision = False Or bFinishSubstration = False
            Threading.Thread.Sleep(50)      'UI thread is sleeping
            If bFinishAddition And Not bShownAdditionResult Then
                Console.WriteLine("Addition Result : " & addResult)
                bShownAdditionResult = True
            End If

            If bFinishSubstration And Not bShownSubstractionResult Then
                Console.WriteLine("Substraction Result : " & SubStractResult)
                bShownSubstractionResult = True
            End If

            If bFinishDivision  And Not bShownDivisionResult Then
                Console.WriteLine("Division Result : " & DivisionResult)
                bShownDivisionResult = True
            End If

        End While

        Console.WriteLine("")
        Console.WriteLine("Finished all threads.")
```


www.dbooks.org

```
        Console.ReadKey()
    End Sub


    Private Sub AddNumber()
        Dim n1 As Integer = 22
        Dim n2 As Integer = 11

        For i As Integer = 0 To 100
            addResult = addResult + (n1 + n2)
            Threading.Thread.Sleep(50)      'sleeping Add thread
        Next
        bFinishAddition = True
    End Sub

    Private Sub SubstractNumber()
        Dim n1 As Integer = 22
        Dim n2 As Integer = 11

        For i As Integer = 0 To 80
            SubStractResult = SubStractResult - (n1 - n2)
            Threading.Thread.Sleep(50)
        Next
        bFinishSubstration = True
    End Sub

    Private Sub DivideNumber()
        Dim n1 As Integer = 22
        Dim n2 As Integer = 11
        For i As Integer = 0 To 60
            DivisionResult = DivisionResult + (n1 / n2)
            Threading.Thread.Sleep(50)
        Next
        bFinishDivision = True
    End Sub

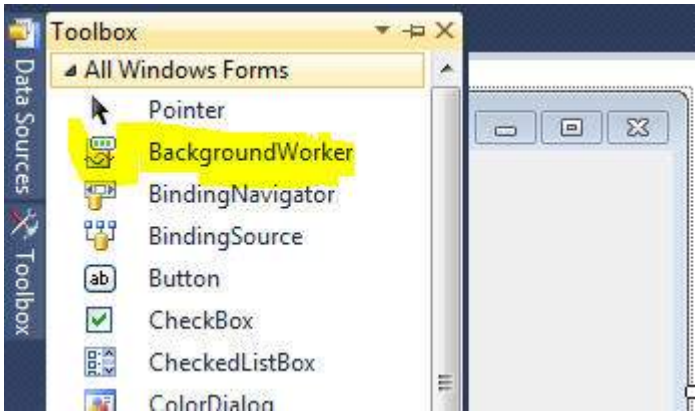
End Module
```

Chapter 47: BackgroundWorker

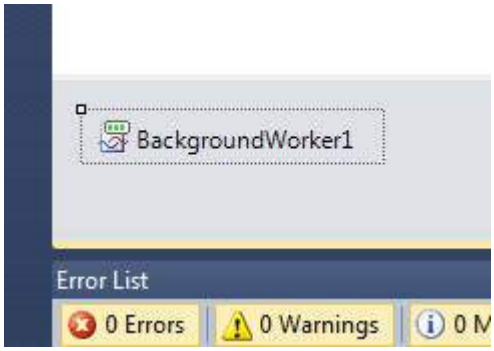
Section 47.1: Using BackgroundWorker

Executing a task with the background worker.

Double Click on the `BackgroundWorker` control from the Toolbox



This is how the BackgroundWorker appears after adding it.



Double click on the added control to get the `BackgroundWorker1_DoWork` event and add the code to be executed when the BackgroundWorker is called. Something like this:

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, ByVal e As
System.ComponentModel.DoWorkEventArgs) Handles BackgroundWorker1.DoWork

    'Do the time consuming background task here

End Sub
```

Calling the BackgroundWorker to perform the task can be done at any event like `Button_Click`, `Textbox_TextChanged`, etc. as follows:

```
BackgroundWorker1.RunWorkerAsync()
```

Modify the `RunWorkerCompleted` event to capture the task finished event of the BackgroundWorker as follows:

```
Private Sub BackgroundWorker1_RunWorkerCompleted(ByVal sender As Object, ByVal e As
System.ComponentModel.RunWorkerCompletedEventArgs) Handles BackgroundWorker1.RunWorkerCompleted
    MsgBox("Done")
End Sub
```

This will display a message box saying `Done` when the worker finishes the task assigned to it.

www.dbooks.org

Section 47.2: Accessing GUI components in BackgroundWorker

You cannot access any GUI components from the BackgroudWorker. For example if you try to do something like this

```
Private Sub BackgroundWorker1_DoWork(sender As Object, e As DoWorkEventArgs)
    TextBox1.Text = "Done"
End Sub
```

you will receive a runtime error saying that "Cross-thread operation not valid: Control 'TextBox1' accessed from a thread other than the thread it was created on."

This is because the BackgroundWorker runs your code on another thread in parallel with the main thread, and the GUI components are not thread-safe. You have to set your code to be run on the main thread using the `Invoke` method, giving it a delegate:

```
Private Sub BackgroundWorker1_DoWork(sender As Object, e As DoWorkEventArgs)
    Me.Invoke(New MethodInvoker(Sub() Me.TextBox1.Text = "Done"))
End Sub
```

Or you can use the ReportProgress method of the BackgroundWorker:

```
Private Sub BackgroundWorker1_DoWork(sender As Object, e As DoWorkEventArgs)
    Me.BackgroundWorker1.ReportProgress(0, "Done")
End Sub

Private Sub BackgroundWorker1_ProgressChanged(sender As Object, e As ProgressChangedEventArgs)
    Me.TextBox1.Text = DirectCast(e.UserState, String)
End Sub
```

Chapter 48: Using BackgroundWorker

Section 48.1: Basic implementation of Background worker class

You need to import System.ComponentModel for using background worker

```
Imports System.ComponentModel
```

Then Declare a private variable

```
Private bgWorker As New BackgroundWorker
```

You need to create two methods for background worker's DoWork and RunWorkerCompleted events and assign them.

```
Private Sub MyWorker_DoWork(ByVal sender As System.Object, ByVal e As
System.ComponentModel.DoWorkEventArgs)
  'Add your codes here for the worker to execute

End Sub
```

The below sub will be executed when the worker finishes the job

```
Private Sub MyWorker_RunWorkerCompleted(ByVal sender As Object, ByVal e As
System.ComponentModel.RunWorkerCompletedEventArgs)
    'Add your codes for the worker to execute after finishing the work.

End Sub
```

Then within your code add the below lines to start the background worker

```
    bgWorker = New BackgroundWorker
    AddHandler bgWorker.DoWork, AddressOf MyWorker_DoWork
    AddHandler bgWorker.RunWorkerCompleted, AddressOf MyWorker_RunWorkerCompleted
    bgWorker.RunWorkerAsync()
```

When you call RunWorkerAsync() function, MyWorker_DoWork will be executed.

www.dbooks.org

Chapter 49: Task-based asynchronous pattern

Section 49.1: Basic usage of Async/Await

You can start some slow process in parallel and then collect the results when they are done:

```
Public Sub Main()
    Dim results = Task.WhenAll(SlowCalculation, AnotherSlowCalculation).Result

    For Each result In results
        Console.WriteLine(result)
    Next
End Sub

Async Function SlowCalculation() As Task(Of Integer)
     Await Task.Delay(2000)

     Return 40
End Function

Async Function AnotherSlowCalculation() As Task(Of Integer)
    Await Task.Delay(2000)

    Return 60
End Function
```

After two seconds both the results will be available.

Section 49.2: Using TAP with LINQ

You can create an `IEnumerable` of `Task` by passing `AddressOf AsyncMethod` to the **LINQ** `SELECT` method and then start and wait all the results with `Task.WhenAll`

If your method has parameters matching the previous **LINQ** chain call, they will be automatically mapped.

```
Public Sub Main()
    Dim tasks = Enumerable.Range(0, 100).Select(AddressOf TurnSlowlyIntegerIntoString)

    Dim resultingStrings = Task.WhenAll(tasks).Result

    For Each value In resultingStrings
        Console.WriteLine(value)
    Next
End Sub

Async Function TurnSlowlyIntegerIntoString(input As Integer) As Task(Of String)
    Await Task.Delay(2000)

    Return input.ToString()
End Function
```

To map different arguments you can replace `AddressOf Method` with a lambda:

```
Function(linqData As Integer) MyNonMatchingMethod(linqData, "Other parameter")
```

Chapter 50: Debugging your application

Whenever you have a problem in your code, it is always a good idea to know what is going on inside. The class System.Diagnostics.Debug in .Net Framework will help you a lot in this task.

The first advantage of the Debug class is that it produces code only if you build your application in Debug mode. When you build your application in Release mode, no code will be generated from the Debug calls.

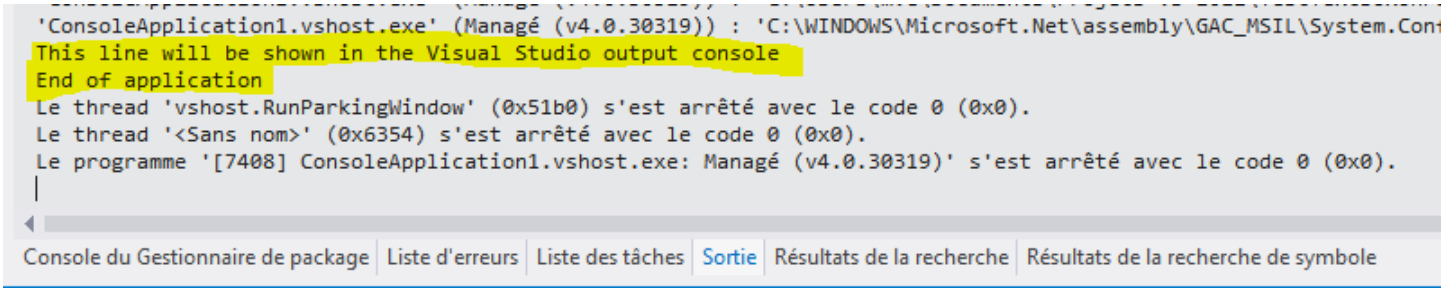
Section 50.1: Debug in the console

```
Module Module1
    Sub Main()
        Debug.WriteLine("This line will be shown in the Visual Studio output console")

        Console.WriteLine("Press a key to exit")
        Console.ReadKey()

        Debug.WriteLine("End of application")
    End Sub
End Module
```

will produce:



Section 50.2: Indenting your debug output

```
Module Module1

    Sub Main()
        Debug.WriteLine("Starting aplication")

        Debug.Indent()
        LoopAndDoStuff(5)
        Debug.Unindent()

        Console.WriteLine("Press a key to exit")
        Console.ReadKey()

        Debug.WriteLine("End of application")
    End Sub

    Sub LoopAndDoStuff(Iterations As Integer)
        Dim x As Integer = 0
        Debug.WriteLine("Starting loop")
        Debug.Indent()
        For i As Integer = 0 To Iterations - 1
            Debug.Write("Iteration " & (i + 1).ToString() & " of " & Iterations.ToString() & ":
Value of X: ")
            x += (x + 1)
```


www.dbooks.org

```
            Debug.WriteLine(x.ToString())
        Next
        Debug.Unindent()
        Debug.WriteLine("Loop is over")
    End Sub
End Module
```

will produce:

```
'ConsoleApplication1.vshost.exe' (Managé (v4.0.30319)) : 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.C
Starting aplication
    Starting loop
        Iteration 1 of 5: Value of X: 1
        Iteration 2 of 5: Value of X: 3
        Iteration 3 of 5: Value of X: 7
        Iteration 4 of 5: Value of X: 15
        Iteration 5 of 5: Value of X: 31
    Loop is over
End of application
Le thread 'vshost.RunParkingWindow' (0x2764) s'est arrêté avec le code 0 (0x0).
Le thread '<Sans nom>' (0xe74) s'est arrêté avec le code 0 (0x0).
Le programme '[8316] ConsoleApplication1.vshost.exe: Managé (v4.0.30319)' s'est arrêté avec le code 0 (0x0).
```

onsole du Gestionnaire de package | Liste d'erreurs | Liste des tâches | Sortie | Résultats de la recherche | Résultats de la recherche de symbole

Section 50.3: Debug in a text file

At the beginning of your application, your must add a TextWriterTraceListener to the Listeners list of the Debug class.

```
Module Module1

    Sub Main()
        Debug.Listeners.Add(New TextWriterTraceListener("Debug of " & DateTime.Now.ToString() &
".txt"))

        Debug.WriteLine("Starting aplication")

        Console.WriteLine("Press a key to exit")
        Console.ReadKey()

        Debug.WriteLine("End of application")
    End Sub
End Module
```

All the Debug code produced will be outputed in the Visual Studio console AND in the text file you chose.

If the file is always the same:

```
Debug.Listeners.Add(New TextWriterTraceListener("Debug.txt"))
```

The output will be appended to the file every time AND a new file starting with a GUID then your filename will be generated.

Chapter 51: Unit Testing in VB.NET

Section 51.1: Unit Testing for Tax Calculation

This example is divided into two pillars

- **SalaryCalculation Class** : Calculating the net salary after tax deduction
- **SalaryCalculationTests Class** : For testing the method that calculates the net salary

Step 1: Create Class Library, name it **WagesLibrary** or any appropriate name. Then rename the class to **SalaryCalculation**

''' ''' Class for Salary Calculations ''' Public Class SalaryCalculation

```
    ''' <summary>
    ''' Employee Salary
    ''' </summary>
    Public Shared Salary As Double

    ''' <summary>
    ''' Tax fraction (0-1)
    ''' </summary>
    Public Shared Tax As Double

    ''' <summary>
    ''' Function to calculate Net Salary
    ''' </summary>
    ''' <returns></returns>
    Public Shared Function CalculateNetSalary()
        Return Salary - Salary * Tax
    End Function
End Class
```

Step 2 : Create Unit Test Project. Add reference to the created class library and paste the below code

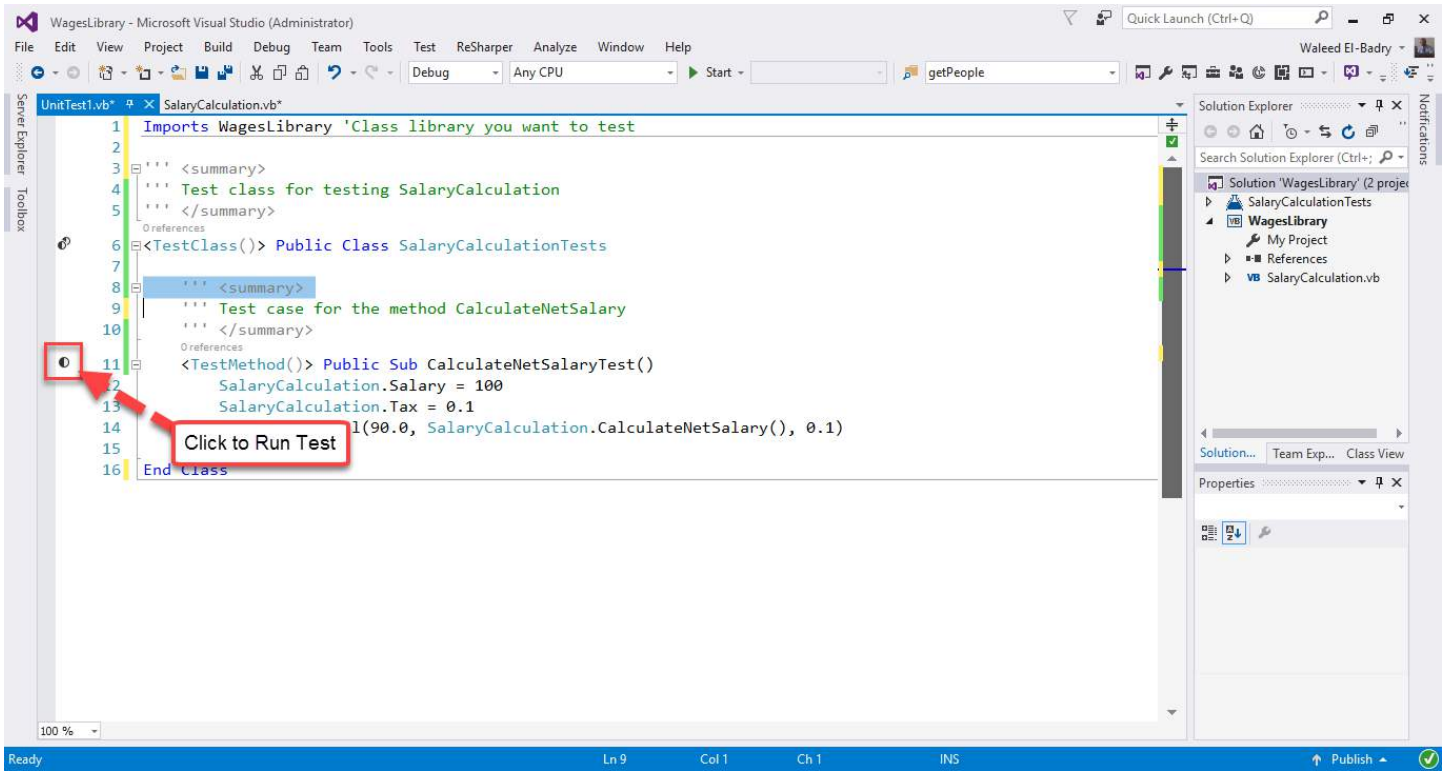
```
Imports WagesLibrary 'Class library you want to test

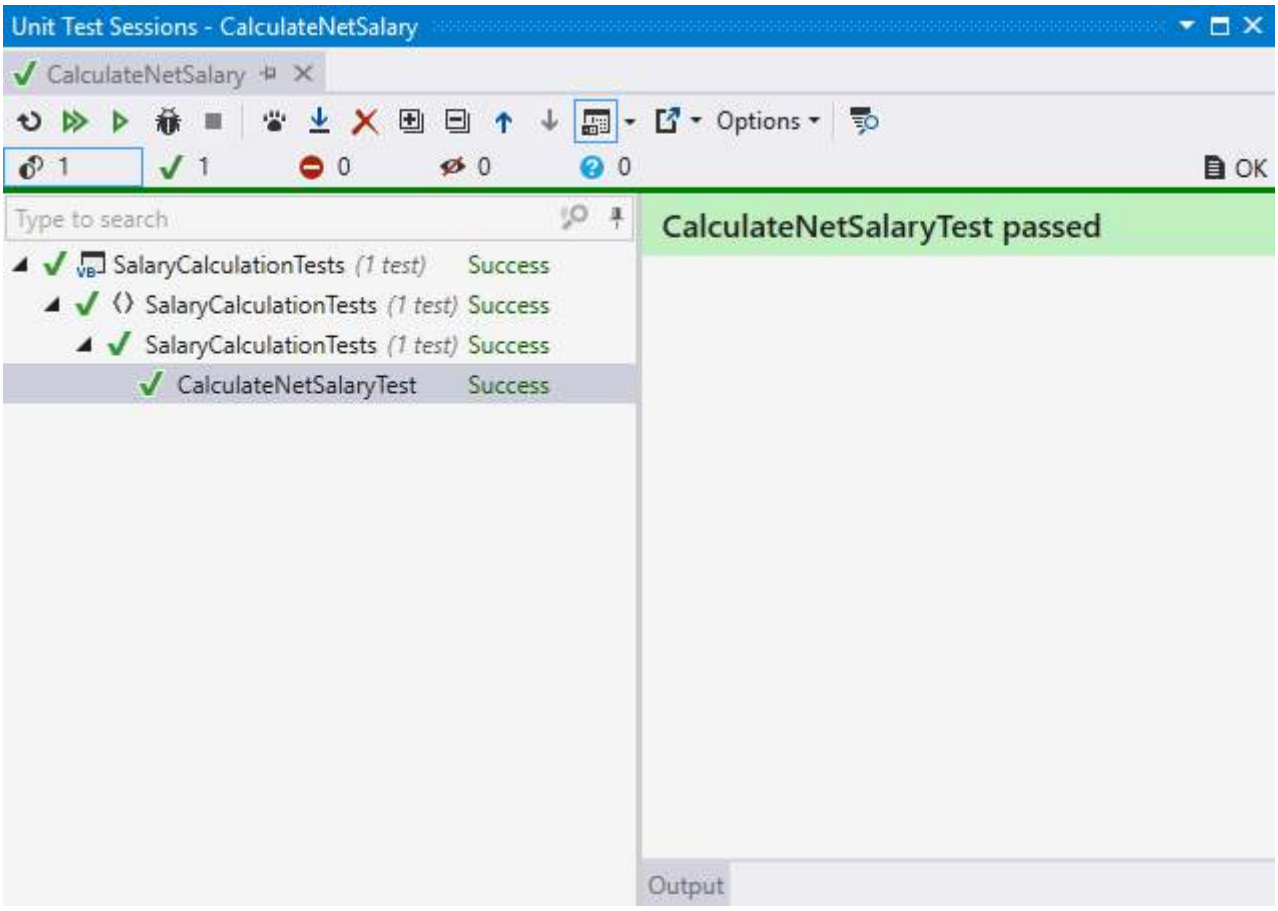
''' <summary>
''' Test class for testing SalaryCalculation
''' </summary>
<TestClass()> Public Class SalaryCalculationTests

    ''' <summary>
    ''' Test case for the method CalculateNetSalary
    ''' </summary>
    <TestMethod()> Public Sub CalculateNetSalaryTest()
        SalaryCalculation.Salary = 100
        SalaryCalculation.Tax = 0.1
        Assert.AreEqual(90.0, SalaryCalculation.CalculateNetSalary(), 0.1)
    End Sub
End Class
```

`Assert.Equal` checks the expected value against the actual calculated value. the value `0.1` is used to allow tolerance or variation between *expected* and *actual* result.

Step 3 : Run the test of the method to see result





Test result

Section 51.2: Testing Employee Class assigned and derived Properties

This example has more tests available in unit testing.

Employee.vb (Class Library)

```
''' <summary>
```

```vb
''' Employee Class
''' </summary>
Public Class Employee

    ''' <summary>
    ''' First name of employee
    ''' </summary>
    Public Property FirstName As String = ""

    ''' <summary>
    ''' Last name of employee
    ''' </summary>
    Public Property LastName As String = ""

    ''' <summary>
    ''' Full name of employee
    ''' </summary>
    Public ReadOnly Property FullName As String = ""

    ''' <summary>
    ''' Employee's age
    ''' </summary>
    Public Property Age As Byte

    ''' <summary>
    ''' Instantiate new instance of employee
    ''' </summary>
    ''' <param name="firstName">Employee first name</param>
    ''' <param name="lastName">Employee last name</param>
    Public Sub New(firstName As String, lastName As String, dateofbirth As Date)
        Me.FirstName = firstName
        Me.LastName = lastName
        FullName = Me.FirstName + " " + Me.LastName
        Age = Convert.ToByte(Date.Now.Year - dateofbirth.Year)
    End Sub
End Class
```

EmployeeTest.vb (Test Project)

```vb
Imports HumanResources

<TestClass()>
Public Class EmployeeTests
    ReadOnly _person1 As New Employee("Waleed", "El-Badry", New DateTime(1980, 8, 22))
    ReadOnly _person2 As New Employee("Waleed", "El-Badry", New DateTime(1980, 8, 22))

    <TestMethod>
    Public Sub TestFirstName()
        Assert.AreEqual("Waleed", _person1.FirstName, "First Name Mismatch")
    End Sub

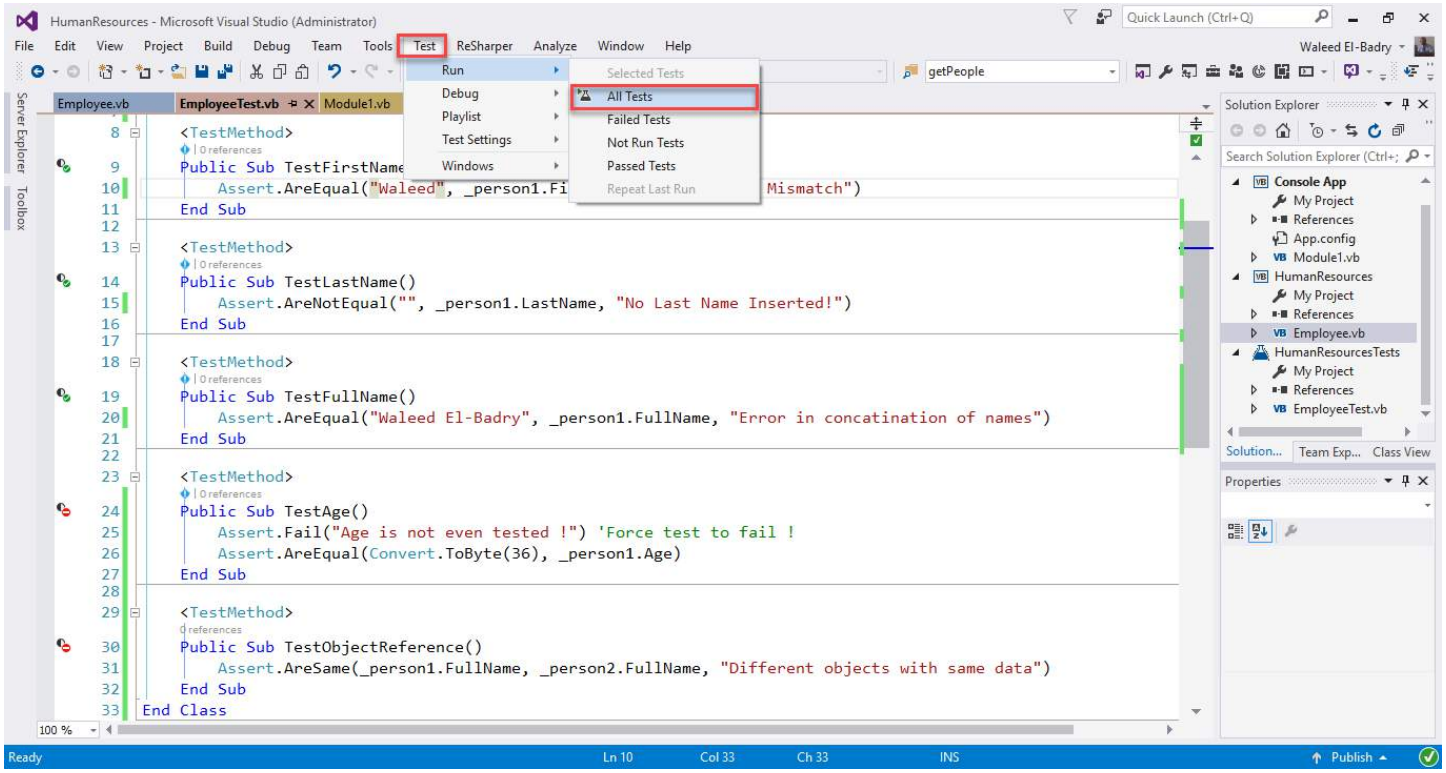
    <TestMethod>
    Public Sub TestLastName()
        Assert.AreNotEqual("", _person1.LastName, "No Last Name Inserted!")
    End Sub

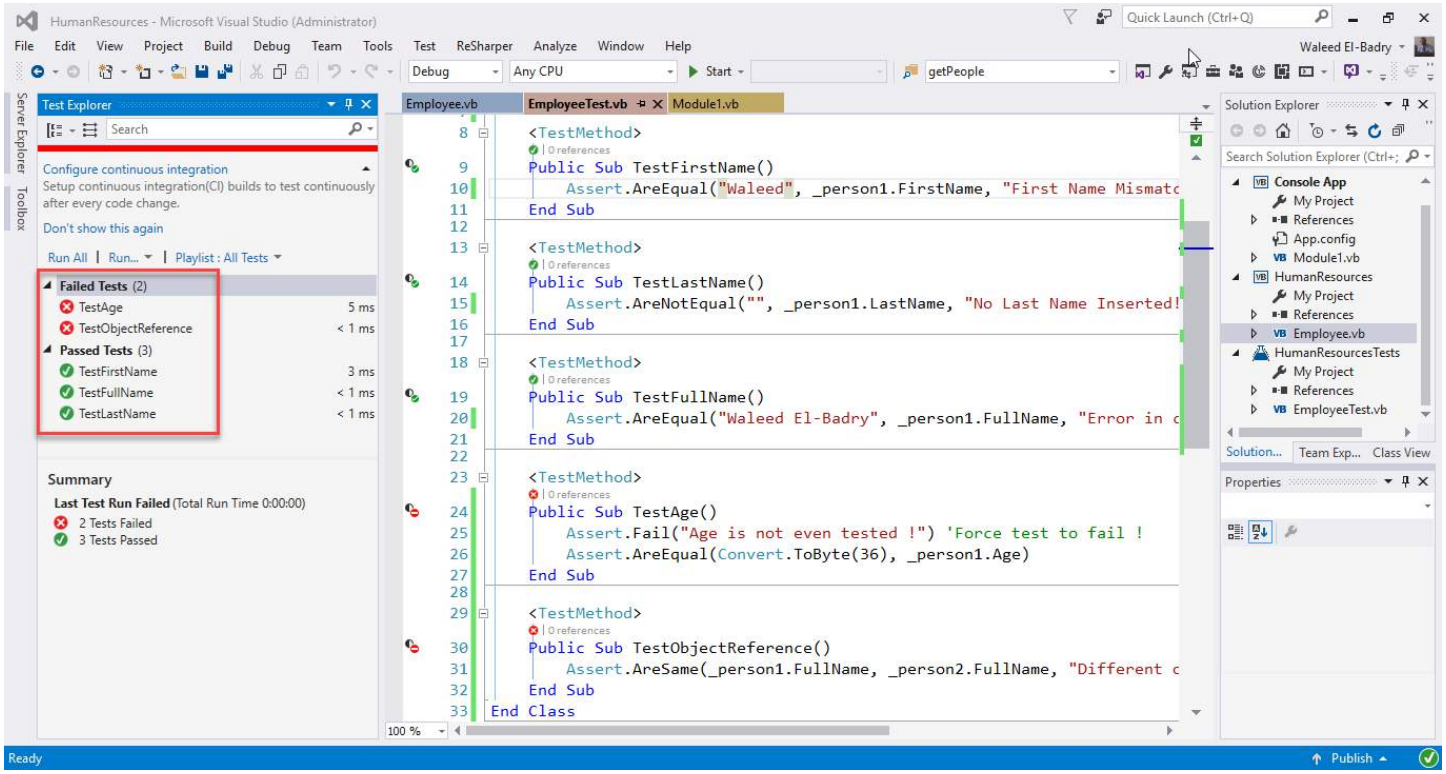
    <TestMethod>
    Public Sub TestFullName()
        Assert.AreEqual("Waleed El-Badry", _person1.FullName, "Error in concatination of names")
    End Sub
```

```
    <TestMethod>
    Public Sub TestAge()
        Assert.Fail("Age is not even tested !") 'Force test to fail !
        Assert.AreEqual(Convert.ToByte(36), _person1.Age)
    End Sub

    <TestMethod>
    Public Sub TestObjectReference()
        Assert.AreSame(_person1.FullName, _person2.FullName, "Different objects with same data")
    End Sub
End Class
```

Result after running tests





Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adam Zuckerman	Chapters 18 and 31
Alessandro Mascolo	Chapter 26
Alex B.	Chapter 30
Allen Binuya	Chapter 5
Andrew Morton	Chapter 28
Axarydax	Chapter 34
Babbillumpa	Chapters 6, 10 and 12
Bart Jolling	Chapter 6
Berken Usar	Chapters 20 and 42
Bjørn	Chapters 4 and 35
Blackwood	Chapter 35
BunkerMentality	Chapter 8
Carlos Borau	Chapter 39
Cary Bondoc	Chapters 1, 4 and 17
Chetan Sanghani	Chapter 5
Cody Gray	Chapters 2 and 14
Dan Drews	Chapter 36
Darren Davies	Chapters 2 and 23
David	Chapter 32
David Wilson	Chapter 22
debater	Chapter 12
djv	Chapters 10 and 38
Dman	Chapters 9 and 40
Drarig29	Chapter 8
DrDonut	Chapters 9 and 11
ElektroStudios	Chapter 10
Fütemire	Chapters 2, 9, 33 and 35
glaubergft	Chapter 2
Happypig375	Chapters 29 and 31
Harjot	Chapter 1
Imran Ali Khan	Chapter 12
InteXX	Chapter 33
JDC	Chapters 24 and 32
Jonas_Hess	Chapter 15
Jones Joseph	Chapter 47
Kendra	Chapter 6
keronconk	Chapter 2
kodkod	Chapter 10
LogicalFlaps	Chapters 2 and 30
lucamauri	Chapter 19
Luke Sheppard	Chapters 9, 13 and 31
Mark	Chapter 12
Mark Hurd	Chapters 3 and 8
Martin Soles	Chapter 19
Martin Verjans	Chapters 1, 3 and 50
Matt	Chapter 34
Matt Wilko	Chapters 7, 13, 25, 28, 33 and 35

www.dbooks.org

MatVAD	Chapters 2, 4, 8, 12, 14, 16, 31, 46 and 48
Mike Robertson	Chapter 16
Milliron X	Chapter 43
Misaz	Chapters 13, 14, 18, 25, 36 and 37
Nadeem_MK	Chapter 35
Nathan	Chapter 41
Nathan Tuggy	Chapter 5
Nico Agusta	Chapters 1, 3, 16 and 31
Nitram	Chapter 35
Proger_Cbsk	Chapters 11 and 44
Robert Columbia	Chapters 4, 5, 8 and 21
RoyalPotato	Chapter 4
Ryan Thomas	Chapter 8
Sam Axe	Chapters 4, 8, 19, 25, 28 and 33
sansknwoledge	Chapter 29
Scott Mitchell	Chapter 12
Seandk	Chapter 9
Sehnsucht	Chapters 2, 3, 8, 11, 18, 26 and 36
Shayan Toqraee	Chapter 47
Shog9	Chapter 10
SilverShotBee	Chapters 12 and 38
StardustGogeta	Chapters 1 and 19
Stefano d'Antonio	Chapters 33, 35, 45 and 49
Steven Doggart	Chapters 8 and 10
TuxCopter	Chapter 8
TyCobb	Chapter 12
varocarbas	Chapter 4
vbnet3d	Chapters 8, 12, 13, 14, 19, 28 and 38
Vishal	Chapter 31
Visual Vincent	Chapter 45
void	Chapters 4, 10, 11, 12 and 34
VortixDev	Chapters 8 and 19
VV5198722	Chapter 27
wbadry	Chapter 51
Zev Spitz	Chapter 36
zyabin101	Chapter 8