Chapter 3: Introduction to Syntax

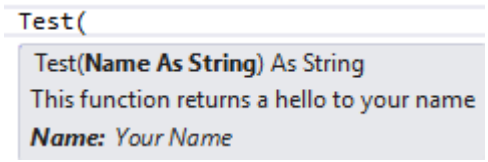
Section 3.1: Intellisense Helper

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
```