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
```

```
        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
```