# Visual Basic .NET

Victor Herrero Cazurro

# Table of Contents

# 1. Introducción

En Visual Basic se programa guiado por los eventos (event-driven programming), esto es, que el flujo del programa, no viene definido por el programador, como ocurre en la programación secuencial, sino por las acciones que en cada momento realiza, sobre la aplicación el usuario de la misma.

El programador de la aplicación, define los eventos que manejará la aplicación y las acciones que se realizarán al producirse dichos eventos, siendo las acciones que realice el usuario, las que dispararán los eventos.

En una aplicación guiada por eventos, la aplicación inicializa los manejadores al arrancarse, y se queda esperando, a la escucha de que se produzcan eventos, cuando estos se producen, se ejecuta el codigo de los manejadores.

# 2. Visual Studio

Herramienta de Microsoft, que permite el desarrollo de aplicaciones tanto de escritorio como Web, con distintos lenguajes.

Actualmente se proporciona una versión Comunity gratuita, que se puede descargar desde **aqui**.

En Visual Studio, la creación de proyectos se divide por lenguaje de .NET Framework (Visual C#, Visual Basic, Visual F#, Visual C++, Pyhon, ...) y por cada lenguaje, por las distintas tecnologias (Web, Windows, WCF, ...).

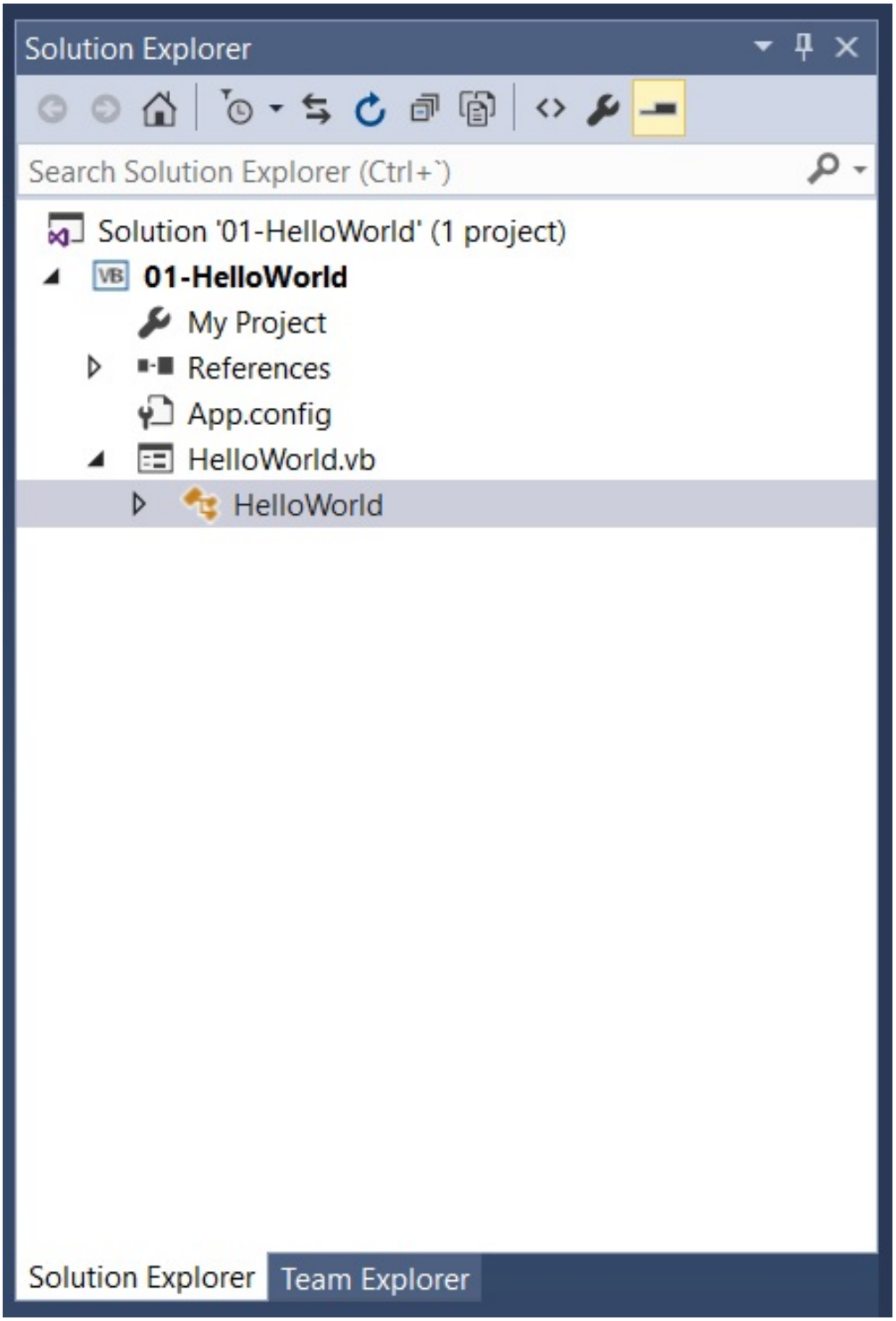
## 2.1. Vistas

Las principales vistas que ofrece Visual Studio son

- Solution Explorer
- Properties
- Toolbox
- Espacio de Edición
- Team Explorer
- Data Sources
- Output

## 2.2. Solution Explorer

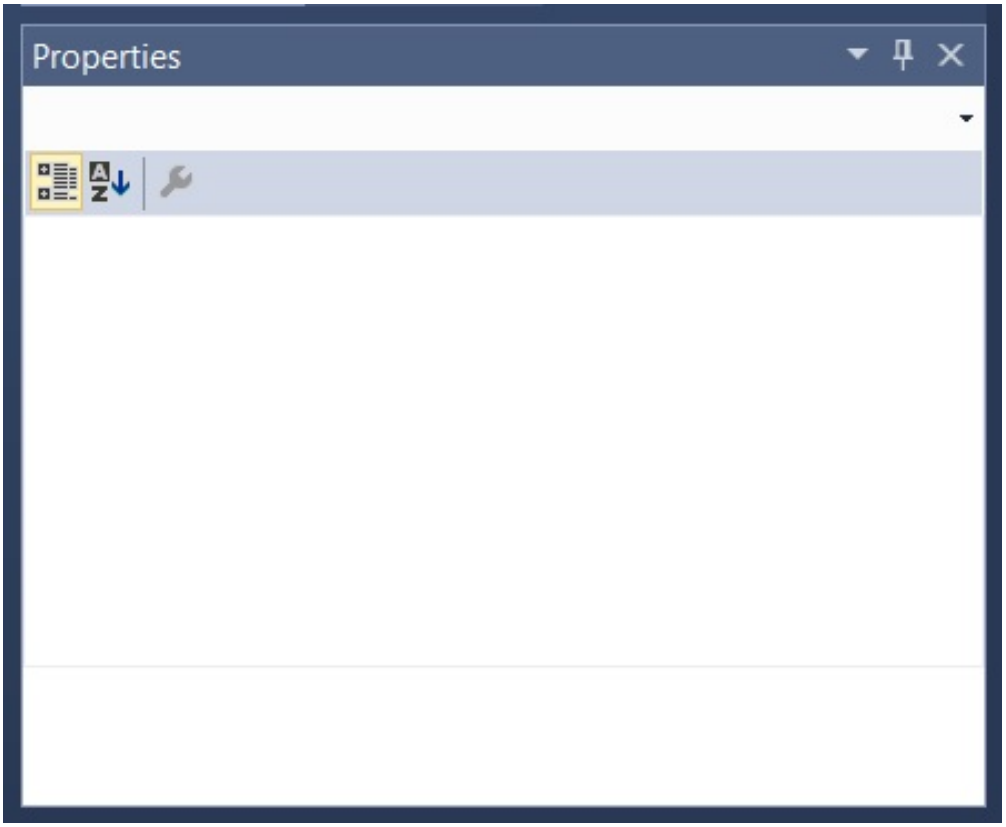
Una de las ventanas principales de VS, es Solution Explorer, que permite navegar por los ficheros que componen la Solución. Se ubica en la parte derecha de la pantalla.




Una Solución será el espacio de trabajo, que puede estar compuestas por distintos proyectos (compilados).

## 2.3. Properties

Otra de las ventanas principales será la de Properties, que permite ver/editar los metadatos de los distintos artefactos que componen la solución, como el nombre de los ficheros. Tambien se ubica en la parte derecha de la pantalla.

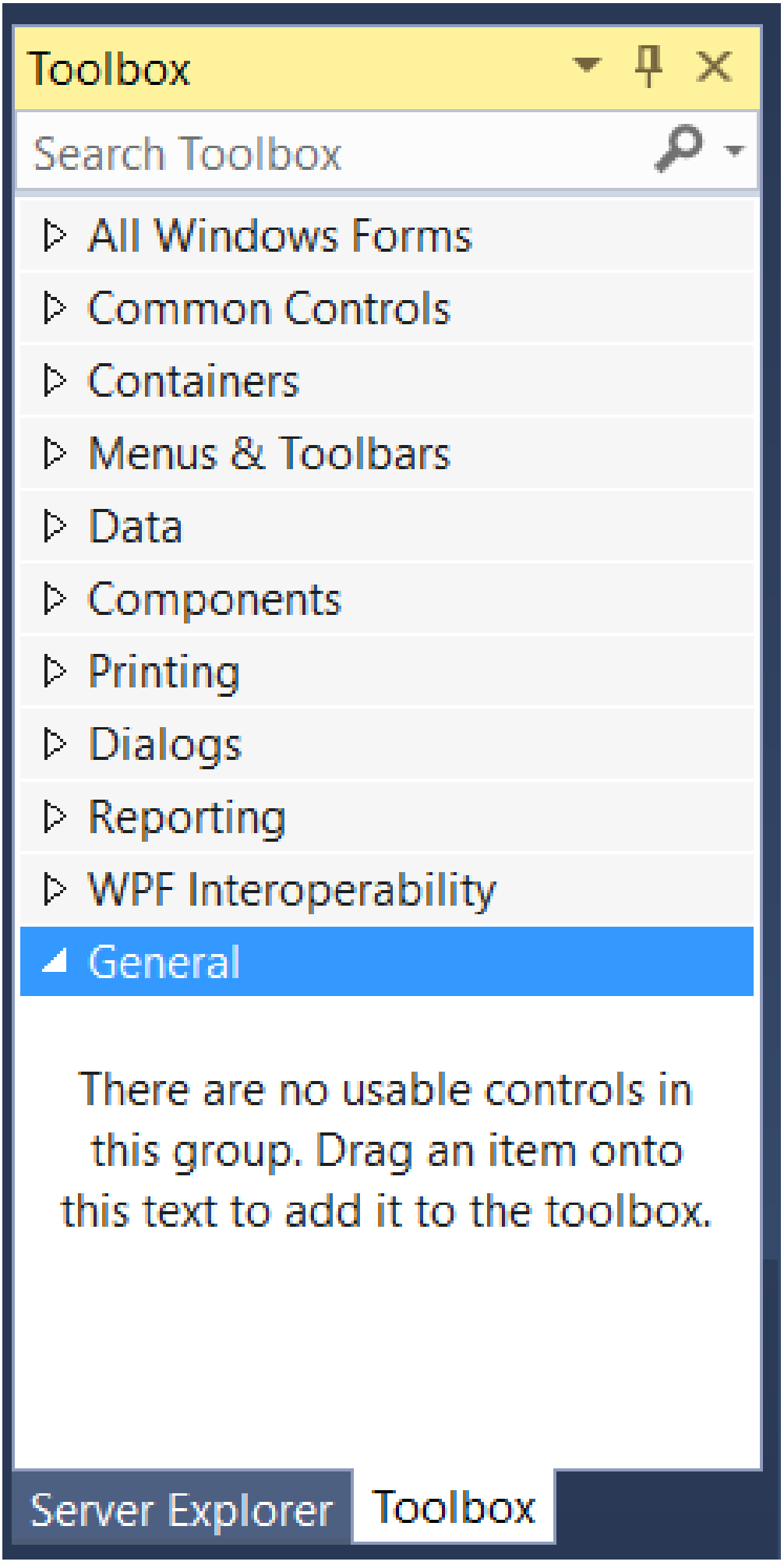



Las Propiedades de los formularios se pueden agrupar en categorias:

- Accesibilidad (Accessibility),
- Apariencia (Appearance),
- Comportamiento (Behavior),
- Datos (Data),
- Diseño (Design),
- Foco (Focus),
- Layout,
- Varios (Misc)
- Estilos Windows (Window Style).

## 2.4. Toolbox

Otra de las ventanas principales, será Toolbox, donde apareceran los componentes que se pueden arrastrar sobre el formulario. Se ubica en la parte izquierda de la pantalla.

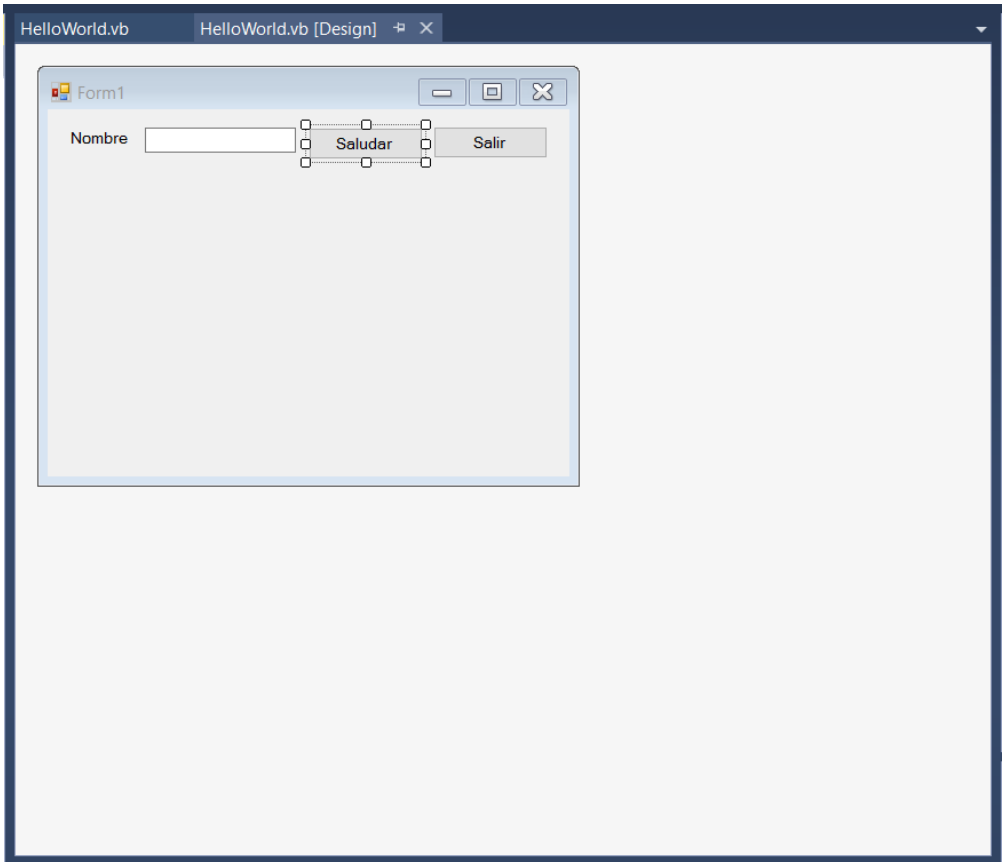

Toolbox
Search Toolbox
All Windows Forms
Common Controls
Containers
Menus & Toolbars
Data
Components
Printing
Dialogs
Reporting
WPF Interoperability
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
Server Explorer
Toolbox

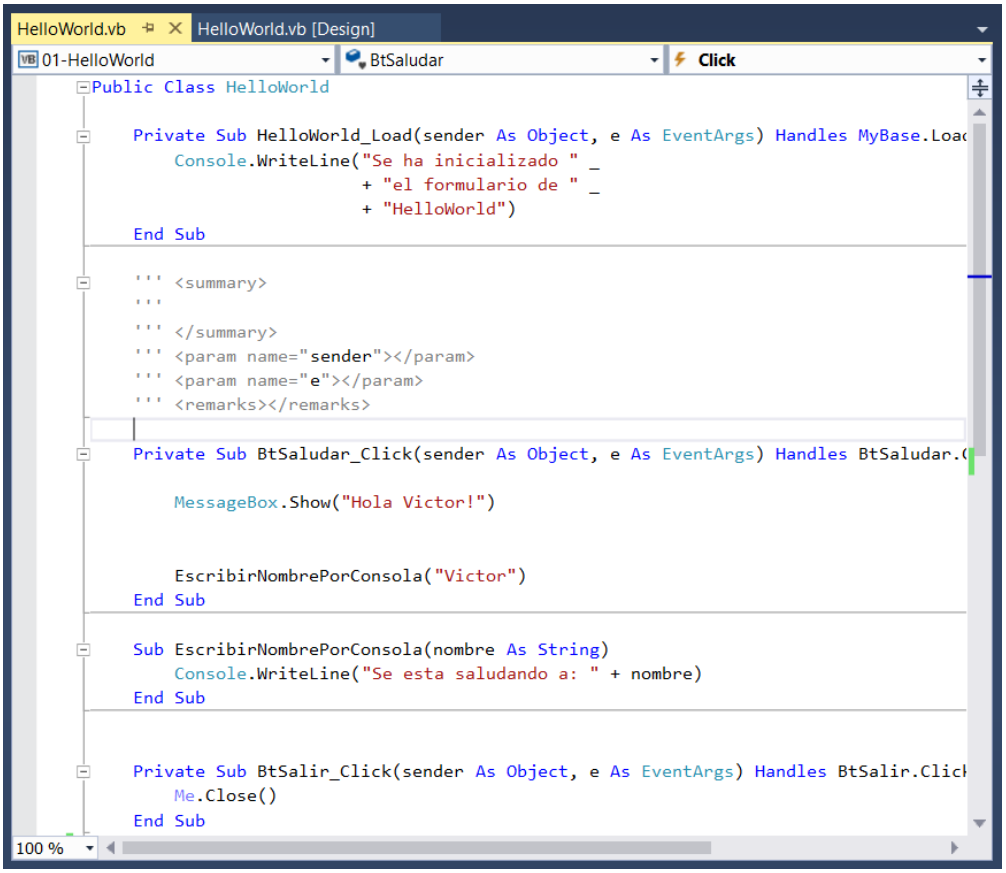
Dado que se pueden añadir varios controles a los formularios que hagan referencia al mismo campo (label, textfield), para los nombres de los controles se suelen emplear prefijos, se suelen emplear los que aparecen en la imagen.

| CONTROL | PREFIX |
|---|---|
| Button | `btn` |
| ComboBox | `cbo` |
| CheckBox | `chk` |
| Label | `lbl` |
| ListBox | `lst` |
| MainMenu | `mnu` |
| RadioButton | `rdb` |
| PictureBox | `pic` |
| TextBox | `txt` |

## 2.5. Espacio de Edición

Otra de las ventanas principales, será el par que forman el editor gráfico de formularios y el editor de código. Ocupan la zona central de la pantalla.




Cuando se edita un fichero de formulario, por defecto se accede al editor gráfico, para acceder al editor de código, se hará doble-click sobre el editor gráfico. Opcionalmente se puede expandir, en la ventana de Solution Explorer, el fichero que representa el formulario y editar el nodo que de este fichero cuelga, que será el fichero de código.

El fichero de código, será una clase de VB, son la siguiente estructura

```
Public Class HelloWorld
.
.
.
End Class
```

# 3. Sintaxis

Veamos algunas caracteristicas especificas de VB.NET en cuanto a la sintaxis.

Por ejemplo, las sentencias no llevan ; al final.

## 3.1. Convenciones de Nomenclatura

Los nombres empezaran por un caracter alfanumerico o _

En palabras compuestas, la primera letra de cada palabra, en mayusculas (nomenclatura camel)

Los nombres de las interfaces, empiezan con **I**

Los nombres de los métodos, empezarán con la primera letra en mayusculas.

## 3.2. Comentarios

Los comentarios se insertan con el caracter '

```
'Esto es un comentario sobre el método
Private Sub BtSaludar_Click(sender As Object, e As EventArgs) Handles BtSaludar.Click

End Sub
```

Se puede incluir un comentario que permite incluir la documentación dentro de la propia clase, con el triple apostrofe (''), al introducirlo Visual Studio incluirá lo siguiente

```
''' <summary>
'''
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>
Private Sub BtSaludar_Click(sender As Object, e As EventArgs) Handles BtSaludar.Click

End Sub
```

## 3.3. Subrayado

El caracter de subrayado (_), permite partir una sentencia en varias lineas.

```
Private Sub BtSaludar_Click(sender As Object, e As EventArgs) Handles BtSaludar.Click

    Console.writeLine("Esta es la primera linea, " _
            + "esta es la segunda, " _
            + "y por ultimo la tercera")

End Sub
```

## 3.4. Separador de sentencias

Para poder escribir varias instrucciones en una misma linea, se emplea el separador **:**

```
a = 3 : b = 4 : c = 5
```

# 4. Tipos de datos

| Tipo | Memoria | Valores | Observaciones |
|---|---|---|---|
| Byte | 1 byte | 0 a 255 | Entero sin signo |
| SByte | 1 byte | -128 y 127 | Entero con signo |
| Short | 2 bytes | -32768 y 32767 | Entero con signo |
| UShort | 2 bytes | 0 a 65535 | Entero sin signo |
| Integer | 4 bytes | -2.147.483.648 a 2.147.483.647 | Entero con signo |
| UInteger | 4 bytes | 0 y 4.294.967.295 | Entero sin signo |
| Long | 8 bytes | - 2147483648 a 2147483647 ó -9,2E+18 a 9,2E+18 | Entero con signo |
| ULong | 8 bytes | 0 a 18.446.744.073.709.551.615 | Entero sin signo |
| Decimal | 16 bytes | +/- 79,228,162,514,264,337,593,543,950,335 y el valor más pequeño distinto de cero es +/- 0.000000000000000000000000001 | Enteros con signo de 12 bytes ajustados por una potencia variable de 10. El número de dígitos que hay a la derecha del separador decimal; va de 0 a 28. |

| Tipo | Memoria | Valores | Observaciones |
|---|---|---|---|
| Single | 4 bytes | - 3,4·1038 a 3,4·1038 | Real |
| Double | 8 bytes | - 1,79·10308 a 1,79·10308 | Real |
| Boolean | 2 bytes | True o False | Booleano |
| Char | 2 bytes | 0 a 65535 | Caracter unicode |
| String | 10 bytes + 1 byte por cada carácter | | Alfanumérico |
| Object | 4 bytes | | Tipo de dato referencia, contiene una direccion en memoria de un objeto |
| Date | Variable | | Fechas, se asigna como cadena con el formato "dd-MM-yy", se puede separar con - o con /. Ejemplo "01-02-16". Oto formato valido seria "dd-MMM-yyyy". Ejemplo "01-JAN-2016" |

# 5. Definición de Variables

Para la declaración de la variable y su tipo, se emplea las palabras reservadas **Dim** y **As**

```
Dim variable As Double
```

Para la asginación de un valor a la variable, se emplea el operador =

```
variable = 3.2;
```

Se pueden realizar la declaración y la asignación en una misma sentencia

```
' Si es un tipo basico
Dim real As Double = 3.2

' Si es un objeto
Dim cadena As New String("Esto es el string")
```

# 6. Operadores

## 6.1. Operadores aritméticos

Permiten realizar calculos con valores numéricos.

| Operador | Simbolo | Descripción |
|---|---|---|
| asignación | = | Permite asignar a una variable el resultado de una sentencia |
| suma | + | Permite realizar una suma con dos valores numéricos |
| resta | - | Permite realizar una resta con dos valores numéricos |
| negativo | - | Permite indicar que un número es negativo |
| multiplicación | * | Permite multplicar dos valores numéricos |
| división | / | Permite dividir dos valores numéricos obteniendo un número real |
| exponenciación | ^ | Permite elevar un numeró a otro número |
| división entera | \ | Permite obtener el cociente de la división, prescindiendo del resto |

### 6.1.1. División por cero

La división por cero produce resultados diferentes dependiendo de los tipos de datos que se utilicen.

- En divisiones de enteros (SByte, Byte, Short, UShort, Integer, UInteger, Long, ULong), se lanza una excepción DivideByZeroException.
- En operaciones de división del tipo de datos Decimal o Single, se lanza una excepción DivideByZeroException.
- En las divisiones con el tipo Double, no se produce ninguna excepción, el resultado es

| Tipo de datos del dividendo | Tipo de datos del divisor | Valor del dividendo | Resultado |
|---|---|---|---|
| Double | Double | 0 | NaN (no es un número definido matemáticamente) |

| Tipo de datos del dividendo | Tipo de datos del divisor | Valor del dividendo | Resultado |
|---|---|---|---|
| Double | Double | > 0 | PositiveInfinity |
| Double | Double | < 0 | NegativeInfinity |

# 6.2. Operadores lógicos

Permiten compara dos expresiones, retornando un valor Booleano.

| Operador | Simbolo | Descripción |
|---|---|---|
| igualdad | = | Compara si dos expresiones son iguales |
| desigualdad | < > | Compara si dos expresiones son distintas |
| menor que | < | Compara si la primera expresion es menor que la segunda |
| mayor que | > | Compara si la primera expresion es mayor que la segunda |
| menor o igual que | < = | Compara si la primera expresion es menor o igual que la segunda |
| mayor o igual que | > = | Compara si la primera expresion es mayor o igual que la segunda |

## 6.2.1. Comparación de cadenas de caracteres

Se pueden compara cadenas de caracteres (String), teniendo el siguiente comportamiento

- Se compara caracter a caracter
- En caso de igualdad, se recurre al siguiente caracter
- En el caso de que una sea subcadena de la otra, la mas larga es la mayor

## 6.2.2. Operador Like

Permite comparar un cadena con un patrón.

Para definir el patrón se pueden emplear los siguientes caracteres especiales

| Caracteres de pattern | Coincidencias en string |
|---|---|
| ? | Cualquier carácter individual |

| Caracteres de pattern | Coincidencias en string |
|---|---|
| * | Cero o más caracteres |
| # | Cualquier dígito individual (0-9) |
| [ charlist ] | Cualquier carácter individual de charlist |
| [! charlist ] | Cualquier carácter individual que no esté incluido en charlist |

```
Dim resultado As Boolean
' La siguiente sentencia retorna True
resultado = "F" Like "F"

' La siguiente sentencia retorna False, ya que una efe mayuscula (F),
' no es igual a una efe minuscula (f)
resultado = "F" Like "f"

' La siguientes sentencia retorna False, ya que una efe (F),
' no es igual a 3 efes (FFF)
resultado = "F" Like "FFF"

' La siguiente sentencia retorna True, ya que la expresión 'a*a' indica que
' el String deberá empezar y terminar por 'a' y entre medias puede haber un número
' indeterminado de caracteres
resultado = "aBBBa" Like "a*a"

' La siguiente sentencia retorna True, ya que F esta en e intervalo que va desde
' A hasta Z
resultado = "F" Like "[A-Z]"
' La siguiente sentencia retorna False, ya que R no esta en e intervalo que va desde
' A hasta M
resultado = "R" Like "[A-M]"
' La siguiente sentencia retorna False, ya que F esta en e intervalo que va desde
' A hasta Z, y se está negando el intervalo
resultado = "F" Like "[!A-Z]"
' La siguiente sentencia retorna True, ya que a no esta en el intervalo que va desde
' A hasta Z, y se está negando el intervalo
resultado = "a" Like "[!A-Z]"
' La siguiente sentencia retorna True, ya que empieza y termina con a, y en el medio
' tiene un unico digito numérico
resultado = "a2a" Like "a#a"
' La siguiente sentencia retorna True, ya que "aM5b" empieza por a seguido de un
' caracter entre L y P, seguido de un digito numerico y acaba con un caracter
' fuera del rango entre c y e
resultado = "aM5b" Like "a[L-P]#[!c-e]"
' La siguiente sentencia retorna True, ya que "BAT123khg" empieza por "B", seguido
' seguido por un caracter unico, en este caso A, despues T y finalmente un numero
' indeterminado de caracteres
resultado = "BAT123khg" Like "B?T*"
```

## 6.3. Operadores con objetos

Exiten en VisualBasic.Net dos operadores que permiten comparar si dos variables hacen referencia exactamente a la misma instancia en memoria de un objeto

- **Is**
- **IsNot**

```
' Operador Is

Dim x As Persona
    Dim y As New Persona()
  x = y
  If x Is y Then
    Console.WriteLine("Las variables hacen referencia a la misma instancia")
End If

' Operador IsNot

x = New Persona()
If x IsNot y Then
    Console.WriteLine("Ahora las variables no hacen referencia a la misma instancia")
End If
```

Tambien es posible, conocer el tipo del objeto al que apunta una variable, con el operador **TypeOf**

```
' Operador TypeOf

Dim y As New Persona()
If TypeOf y Is Persona Then
        Console.WriteLine("La variable x hace referencia a un objeto de tipo Persona")
End If
```

## 6.4. Operadores de concatenación

En VB se pueden emplear los operadores **&** y + para concatenar cadenas de caracteres, siendo su comportamiento el siguiente

**&** concatena

+ concatena cuando se trabaja con afanumericos y suma cuando son numericos, ojo que si el numerico se representa como una cadena, tambien suma.

```
var1 = "10.01"
var2 = 11
resultado1 = var1 & var2 //10.0111
resultado2 = var1 + var2 //21.01
```

## 6.5. Operadores de acceso

Se pueden emplear **.** y **!**

**.** permite acceder a miembros (campos, propiedades, eventos o metodos) de Clases, Estructuras, Interfaces o Enumeraciones.

**!** permite acceder al diccionario de Clases o Interfaces. Se conoce como diccionario a una Propiedad predeterminada que acepta un solo argumento String

```
Public Class MiClase

    Default Public ReadOnly Property index(ByVal s As String) As String

        Get
            return s
        End Get

    End Property

End Class

Public Class TestMiClase

    Public Sub AccesoDiccionario()

        Dim variable As MiClase = New MiClase()

        variable!X //Retornará el String "X"

    End Sub

End Class
```

# 7. Arrays

Un array es un conjunto ordenado de objetos del mismo tipo.

Es necesario definir el tamaño del **Array** al construir el objeto, ya sea de forma explicita, indicando el numero de elementos del **Array** o de forma implicita, indicando los elementos del **Array**.

A la hora de declarar una variable, se ha de afectar o bien al nombre de la varaible o bien al tipo de

los elementos del **Array**, con **()**

```
' Declaracion implicita
Dim numbers = New Integer() {1, 2, 4, 8}
Dim doubles() As Double = {1.5, 2, 9.9, 18}
Dim singles As Single() = {1.5, 2, 9.9, 18}

'Declaracion explicita
Dim numbers = New Integer(4) {}
Dim doubles() As Double = New Double(4) {}
Dim singles() As Single = New Single(4) {}
```

De indicar los elementos, no se indica el tamaño, es el tamaño - 1, es decir el índice del último elemento, ya que el primer indice es 0.

El tamaño es fijo, pero se puede redimensionar con la palabra reservada **ReDim**, si se emplea adicionalmente **Preserve**, se preservaran los elementos que estuvieran definidos. En realidad un **Array** no se redimensiona, sino que se crea un objeto nuevo de tipo Array y se pueden o no copiar los datos de uno a otro.

```
ReDim Preserve numbers(20)
```

Para añadir elementos al **Array**, se emplea la variable de tipo Array, indicando entre parentesis la posición donde se quiere añadir el elemento y a esa expresion se le asigna el valor, para leerlos de forma analoga.

```
numbers(1) = 0
```

Se pueden definir arrays multidimensionales

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

Dim matrix()() As Integer = {{1, 2}, {3, 4}}

Dim matrix = New Integer(3, 1) {{1, 2}, {3, 4}, {5, 6}, {7, 8}}

Dim matrix()() As Integer = New Integer(11)() {}

Dim matrix As Integer(,) = {{1, 2}, {3, 4}, {5, 6}}
```

Un **Array** es iterable, dado que permite conocer el indice mayor de cada dimension del **Array**, con el método **GetUpperBound**

```
' Se itera desde 0 hasta el indice mayor de la dimension primera del array numbers
For index = 0 To numbers.GetUpperBound(0)
    Debug.WriteLine(numbers(index))
Next
```

De forma analoga se pueden iterar **Arrays** multidimensionales

```
For index0 = 0 To matrix.GetUpperBound(0)
    For index1 = 0 To matrix.GetUpperBound(1)
        Debug.Write(matrix(index0, index1).ToString & " ")
    Next
    Debug.WriteLine("")
Next
```

Tambien se puede iterar con la sentencia For...Each..Next, aunque esta da menos control sobre los elementos y su posición

```
For Each number In numbers
    Debug.WriteLine(number)
Next

' Iteracion sobre array multidimensional, no hay control sobre si se recorre por filas
o columnas, es siempre por filas,
' y tampoco se controla cuando acaba una fila
For Each number In matrix
    Debug.WriteLine(number)
Next
```

# 8. Sentencias de control de flujo

VB.NET se ejecuta de forma secuencial, sentencia a sentencia, cuando hay una invocación de un método, se ejecuta secuencialmente el contenido del método.

Existen una serie de sentencias que permiten modificar este comportamiento.

## 8.1. If...Then...Else

Estructura de control, que permite tomar una decisión de ejecutar un conjunto de sentencias, si se cumple una condición.

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

If count = 0 Then
    message = "There are no items."
ElseIf count = 1 Then
    message = "There is 1 item."
Else
    message = "There are " & count & " items."
End If
```

## 8.2. Select Case...Case

Estructura de control, que permite evaluar una expresión, y dependiendo de su valor, ejecutar un conjunto de sentencias u otro.

El valor de la expresión, debera ser de tipo básico (Boolean, Byte, Char, Date, Double, Decimal, Integer, Long, Object, SByte, Short, Single, String, UInteger, ULong y UShort)

Las sentencias de Case son excluyentes, si se cumple una, no se sigue comprobando el resto.

Se pueden emplear las palabras reservadas **Is** y **To** en los Case, para establecer

**Is** → Expresiones booleanas ciertas

**To** → Rangos de valores

===

```
Dim number As Integer = 8
Select Case number
    Case 1 To 5
        Debug.WriteLine("Entre 1 y 5, incluidos")
    ' La siguiente clausula es la unica que se cumple.
    Case 6, 7, 8
        Debug.WriteLine("Entre 6 y 8, incluidos")
    Case 9 To 10
        Debug.WriteLine("9 o 10")
        Case Is > 10
                Debug.WriteLine("mayor que 10")
    Case Else
        Debug.WriteLine("Ninguna de las anteriores")
End Select
```

## 8.3. While...End While

Estructura que permite ejecutar una serie de sentencias de forma repetida, siempre que se cumpla

una expresión Booleana

```
Dim index As Integer = 0
While index <= 10
    Debug.Write(index.ToString & " ")
    index += 1
End While

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Se puede finalizar la iteración en cualquier momento ejecutando **Exit While**

O tambien se puede finalizar una iteración, dejando de ejecutar el resto de sentencias dentro del bloque **While**, pasando inmediatamente a la siguiente ieración, si la hubiera, con **Continue While**

```
Dim index As Integer = 0
While index < 100000
    index += 1

    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 And index <= 8 Then
        Continue While
    End If

    ' Display the index.
    Debug.Write(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit While
    End If
End While

Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

# 8.4. Do While…Loop

Repite un bloque de instrucciones mientras una condición Boolean sea True, si se emplea **While** o hasta que la condición se convierta en True si se emplea **Until**.

La condición se comprueba al principio, asociado a **Do**, con lo que puede ser que el bloque no se ejecute nunca

```
Dim index As Integer = 0
Do While index <= 10
    Debug.Write(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

De forma analoga a como sucede con la sentencia **While**, se puede finalizar las iteraciones con **Exit Do**

```
Dim index As Integer = 0
Do While index <= 100
    If index > 10 Then
        Exit Do
    End If

    Debug.Write(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

## 8.5. Do…Loop Until

La condición se compruebe al final del bloque, asociado a **Loop**, con lo que el bloque al menos se ejecuta una vez.

```
Dim index As Integer = 0
Do
    Debug.Write(index.ToString & " ")
    index += 1
Loop Until index > 10

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

## 8.6. For…To…Step…Next

Sentencia que permite repetir un grupo de sentencias un número de veces especificado.

Se establece una variable de control **Counter** que variará en un rango definido por **To** con una variación por iteración de 1 por defecto, puediendose definir con **Step**

Los posibles tipos de **Counter** son un Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single, o Double.

```
For number As Double = 2 To 0 Step -0.25
    Debug.Write(number.ToString & " ")
Next
Debug.WriteLine("")
' Output: 2 1.75 1.5 1.25 1 0.75 0.5 0.25 0
```

O una enumeración.

```
Public Enum Mammals
    Buffalo
    Gazelle
    Mongoose
    Rhinoceros
    Whale
End Enum

Public Sub ListSomeMammals()
    For mammal As Mammals = Mammals.Gazelle To Mammals.Rhinoceros
        Debug.Write(mammal.ToString & " ")
    Next
    Debug.WriteLine("")
    ' Output: Gazelle Mongoose Rhinoceros
End Sub
```

Se pueden anidar sentencias **For…Next**, teniendo en cuenta que variable contador ha de controlar la repetición de cada bucle.

```
For indexA = 1 To 3
    ' Create a new StringBuilder, which is used
    ' to efficiently build strings.
    Dim sb As New System.Text.StringBuilder()

    ' Append to the StringBuilder every third number
    ' from 20 to 1 descending.
    For indexB = 20 To 1 Step -3
        sb.Append(indexB.ToString)
        sb.Append(" ")
    Next indexB

    ' Display the line.
    Debug.WriteLine(sb.ToString)
Next indexA
' Output:
'  20 17 14 11 8 5 2
'  20 17 14 11 8 5 2
'  20 17 14 11 8 5 2
```

De forma analoga a como sucede con la sentencia **While**, se puede finalizar la iteración en cualquier momento ejecutando **Exit For**

O tambien se puede finalizar una iteración, dejando de ejecutar el resto de sentencias dentro del bloque **For**, pasando inmediatamente a la siguiente iteración, si la hubiera, con **Continue For**

```
For index As Integer = 1 To 100000
    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 And index <= 8 Then
        Continue For
    End If

    ' Display the index.
    Debug.Write(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit For
    End If
Next
Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

## 8.7. For Each...In...Next

Repite un grupo de instrucciones para cada elemento de una colección.

```
' Create a list of strings by using a
' collection initializer.
Dim lst As New List(Of String) _
    From {"abc", "def", "ghi"}

' Iterate through the list.
For Each item As String In lst
    Debug.Write(item & " ")
Next
Debug.WriteLine("")
'Output: abc def ghi
```

Se pueden anidar.

```
' Create lists of numbers and letters
' by using array initializers.
Dim numbers() As Integer = {1, 4, 7}
Dim letters() As String = {"a", "b", "c"}

' Iterate through the list by using nested loops.
For Each number As Integer In numbers
    For Each letter As String In letters
        Debug.Write(number.ToString & letter & " ")
    Next
Next
Debug.WriteLine("")
'Output: 1a 1b 1c 4a 4b 4c 7a 7b 7c
```

De forma analoga a como sucede con la sentencia **For**, se puede finalizar la iteración en cualquier momento ejecutando **Exit For**

O tambien se puede finalizar una iteración, dejando de ejecutar el resto de sentencias dentro del bloque **For**, pasando inmediatamente a la siguiente iteración, si la hubiera, con **Continue For**

## 8.8. Using...End Using

Sentencia que permite declarar uno o varios recursos que serán gestionados por el entorno, por lo que se cerrarán de forma automatica.

Los recursos gestionados deben implementar la interface **IDisposable**.

```
Private Sub WriteFile()
    Using writer As System.IO.TextWriter = System.IO.File.CreateText("log.txt")
        writer.WriteLine("This is line one.")
        writer.WriteLine("This is line two.")
    End Using
End Sub

Private Sub ReadFile()
    Using reader As System.IO.TextReader = System.IO.File.OpenText("log.txt")
        Dim line As String

        line = reader.ReadLine()
        Do Until line Is Nothing
            Console.WriteLine(line)
            line = reader.ReadLine()
        Loop
    End Using
End Sub
```

Esta sentenia permite ahorrar la escritura de una buena parte de codigo dado que los siguientes bloques son equivalentes

```
' Bloque que permite el uso del recurso resource y su cerrado automatico
Using resource As New resourceType
    ' Insert code to work with resource.
End Using

' Bloque que permite el uso del recurso resource, pero el cerrado (Dispose()), se hace
' manualmente
Dim resource As New resourceType
Try
    ' Insert code to work with resource.
Finally
    If resource IsNot Nothing Then
        resource.Dispose()
    End If
End Try
```

# 8.9. With...End With

Sentencia que permite referenciar  a un objeto y dentro del bloque acceder de forma sucesiva a las caracteristicas de dicho objeto, sin necesidad de volver a establecer la referencia.

```
Private Sub AddCustomer()
    Dim theCustomer As New Customer

    With theCustomer
        .Name = "Coho Vineyard"
        .URL = "http://www.cohovineyard.com/"
        .City = "Redmond"
    End With

    With theCustomer.Comments
        .Add("First comment.")
        .Add("Second comment.")
    End With
End Sub

Public Class Customer
    Public Property Name As String
    Public Property City As String
    Public Property URL As String

    Public Property Comments As New List(Of String)
End Class
```

Se pueden anidar estos bloques

```
Dim theWindow As New EntryWindow

With theWindow
    With .InfoLabel
        .Content = "This is a message."
        .Foreground = Brushes.DarkSeaGreen
        .Background = Brushes.LightYellow
    End With

    .Title = "The Form Title"
    .Show()
End With
```

# 9. Modificadores de Acceso

Los modificadores e acceso permiten defini que codigo tiene acceso a un determinado elemento.

## 9.1. Public

Especifica que se puede acceder desde cualquier contexto al elemento declarado.

Se puede emplear sobre: Interface, Module, Class, Structure, Dim, Sub, Function, Property, Const,

Enum, Event y Delegate

## 9.2. Protected

Especifica que se puede acceder al elemento declarado desde el contexto de declaración o desde una clase derivada del contexto de declaración.

Se puede emplear sobre: Interface, Class, Structure, Dim, Sub, Function, Property, Const, Enum, Event y Delegate.

## 9.3. Friend

Especifica que se puede acceder al elemento solo desde el mismo ensamblado donde esta declarado el elemento, y no desde otros ensamblados.

Se puede emplear sobre: Interface, Module, Class, Structure, Dim, Sub, Function, Property, Const, Enum, Event y Delegate.

## 9.4. Protected Friend

Especifica que se puede acceder al elemento declarado, desde el mismo ensamblado, desde clases derivadas del contexto donde se declara o ambas.

Se puede emplear sobre: Interface, Class, Structure, Dim, Sub, Function, Property, Const, Enum, Event y Delegate.

## 9.5. Private

Especifica que sólo se puede tener acceso al elemento declarado desde el contexto de la declaración.

Se puede emplear sobre: Interface, Class, Structure, Dim, Sub, Function, Property, Const, Enum, Event y Delegate.

# 10. Orientacion a Objetos

Paradigma de programación que emplea **Objetos** para la creación de aplicaciones.

Se persigue que los **Objetos** representen elementos concretos, que realicen tareas especificas, consiguiendo que cada elemento aporte al conjunto algo único y útil para la funcionalidad de la aplicación.

Uno de los objetivos de la Orientación a Objetos es la de no repetir código, lo cual permite identificar facilmente problemas en los algoritmos, lo cual repercute en la buena mantenibilidad.

Se basa principalmente en tres tecnicas

- Encapsulación

- Herencia
  - Abstracción
  - Interfaces
- Polimorfismo
  - Sobrecarga
  - Sobrescritura

# 10.1. Encapsulación

Busca que los objetos sean responsables de sus caracteristicas, de tal forma que otros objetos que interaccionen con el, no puedan acceder o modificar sus caracteristicas, para ello se precisan modificadores de acceso para las caracteristicas.

Tipicamente, se definen métodos que permiten interaccionar con dichas caracteristicas, estos métodos por convenio se llaman **Get** para leer el valor de la caracteristica y **Set** para establecer el valor de la caracteristica.

En VB.NET la encapsulación se consigue con las **propiedades**.

```
Public Class employee
    ' Declaración de propiedad sobre un campo privado con get y set
    Public Property salary() As Double
        Get
            Return salaryValue
        End Get
        Set(ByVal value As Double)
            salaryValue = value
        End Set
    End Property
End Class
```

# 10.2. Herencia

Permite la reutilización de código, para aquellos elementos que compartan parte de su definición.

La idea de la herencia es poder agrupar en una clase padre todas las caracteristicas comunes de varios tipos de objetos, clases hijas, de tal forma que se escriban una unica vez, aunque esten presentes en todas esas tipologias hijas.

La herencia en VB.NEt es simple, no se permite heredar de varias clases a la vez.

El modificador de acceso de una clase hija debe ser igual o más restrictivo que el de su clase padre. Por ejemplo, una clase Public no puede heredar una clase Friend o Private, y una clase Friend no puede heredar una clase Private.

En VB.NET la herencia se establece con la palabra reservada **Inherits**

```
Class Persona

End Class

Class Trabajador
    Inherits Persona

End Class
```

Se puede heredar de todas las clases, excepto que hayan sido marcadas con la palabra reservada **NotInheritable**, que indicará que no puede ser heredada.

```
NotInheritable Class Trabajador
    Inherits Persona

End Class
```

Se pueden mantener referencias a objetos mas concretos, con variables mas genericas, esto es

```
' Declaración de una variable de tipo Persona, que es clase padre de Trabajador
Dim p As Persona

' Asignación a la variable de tipo Persona de un objeto de tipo Trabajador, que será
mas concreto.
p = New Trabajador()
```

### 10.2.1. Abstracción

Permite declarar un tipo parcialmente, esto es, sin declarar totalmente todos sus miembros.

En VB:NET se emplea la palabra reserada **MustInherit**.

Las clases **Abstractas** por no estar completamente definidas no son instanciables, su suo se limia unicamente a ser clases Padre.

```
MustInherit Class Persona

End Class
```

Las Clases Abstracatas habitualemente tiene métodos abstractos, esto se consigue con la palabra **MustOverride**, cuando un método es marcado como **MustOverride** no llevará cuerpo.

```
MustInherit Class Persona

    MustOverride Sub metodo()

End Class
```

### 10.2.2. Interfaces

Permiten establecer un contrato que ha de cumplir una clase, en forma de las acciones que como minimo va a tener que proporcionar la clase que cumpla el contrato.

Son utiles para establecer arquitecturas, dado que permiten definir relaciones entre clases que no tienen porque existir todavia.

En VB.NET, una interface se declara con la palabra reservada **Interface**

Por convenio los nombres de las interfaces deberan empezar por **I**.

Todos los métodos declarados en una **Interface**, son **Public**

```
Public Interface IAccount
   Sub PostInterest()
End Interface
```

Para indicar que una clase se compromete en cumplir una interface, se emplea la palabra reservada **Implements**, tanto a nivel de la clase, como del método implementado

```
Public Class BusinessAccount
                Implements IAccount

    Sub PostInterest() Implements IAccount.PostInterest

    End Sub

End Class
```

Se pueden definir variables de un tipo **Interface**, aunque no se pueden instanciar Interfaces, a este respecto funcionan como clases abstractas.

## 10.3. Polimorfismo

Existen dos tipos de polimorfismo

- Estático → Se resuelve en tiempo de compilación. Es el mecanismo de la sobrecarga de métodos.
- Dinámico → Se resuelve el tiempo de ejecución. Es el mecanismo de la sobrescritura de métodos.

### 10.3.1. Sobrecarga

Permite definir métodos que se llamen de la misma forma y dependiendo de los tipos de los parametros, se ejecuta un método u otro.

Cuando se produzca la sobrecarga, se deberán marcar los métodos con **Overloads** o con **Overrides**, si es que además de sobrecarga, se está produciendo sobrescritura.

```
Class Trabajador

    Overloads Sub metodo()
        Console.WriteLine("En Trabajador")
    End Sub

    Overloads Sub metodo(parametro As String)

    End Sub

End Class
```

### 10.3.2. Sobrescritura y Sombreado

Permite reemplazar métodos de la clase padre.

Para ello VB.NET propociona diferentes palabras reservadas

- Overridable → Marca un método de una clase padre, como invalidable.
- Overrrides → Reemplaza un método marcado como **Overridable** en la clase base
- NotOverridable → Marca un método como no invalidable, es obligatorio que dicho método sobrescriba otro
- MustOverride → Indica que el método es abstracto, no tiene cuerpo y por tanto debe ser implementado por la clase hija, cualquier clase con métodos **MustOverride**, deberá ser marcada como **MustInherit**
- Shadows → Marca un método de una clase Hija, para que sombree el método con la misma firma de la clase Padre.

El sombreado, es el comportamiento por defecto en VB.NET con respecto a la sobrescritura.

Para poder hacer sobrescritura, el método del padre ha de ser marcado como **Overridable**, aunque esto no garantiza que se produzca la sobrescritura, para asegurarse la sobrescritura y no el sombreado, hay que marcar el método de la clase hija con **Overrides**, de no hacerlo, se presupone que está marcado con Shadows.

La diferencia entre sobrescritura y sombreado, viene determinada por el codigo que se ejecuta cuando se tiene una variable del tipo Padre que hace referencia a un objeto del tipo Hijo y se invoca un método que está sobrescrito o sombreado.

Veamos un ejemplo, en el que se produce Sombreado

```
Module Module1

    Sub Main()

        Dim p As Persona = New Trabajador()

        Dim t As Trabajador = p

        Console.WriteLine("Sombreado")
        'Se invoca el método desde una variable de tipo Trabajador, por lo que se
accede a lo definido en Trabajador
        t.metodoSombreado()
        'Se invoca el método desde una variable de tipo Persona, por lo que se accede
a lo definido en Persona, aunque el
        'objeto sea de tipo Trabajador
        p.metodoSombreado()

        Console.ReadLine()

    End Sub

End Module

Class Persona

    ' La marca Overridable en este caso es opcional
    Overridable Sub metodoSombreado()
        Console.WriteLine("En persona")
    End Sub

End Class

Class Trabajador

    Shadows Sub metodoSombreado()
        Console.WriteLine("En Trabajador")
    End Sub

End Class
```

Veamos un ejemplo, en el que se produce Sobrescritura

```
Module Module1

    Sub Main()

        Dim p As Persona = New Trabajador()

        Dim t As Trabajador = p

        Console.WriteLine("Sobrescritura")
        'Se invoca el método desde una variable de tipo Trabajador, por lo que se
accede a lo definido en Trabajador
        t.metodoSobrescrito()
        'Se invoca el método desde una variable de tipo Persona, pero se accede a lo
definido en Trabajador, ya que el
        'objeto es de tipo Trabajador
        p.metodoSobrescrito()

        Console.ReadLine()

    End Sub

End Module

Class Persona

    ' La marca Overridable en este caso es obligatoria
    Overridable Sub metodoSobrescrito()
        Console.WriteLine("En persona")
    End Sub

End Class

Class Trabajador

    Overrides Sub metodoSobrescrito()
        Console.WriteLine("En Trabajador")
    End Sub

End Class
```

# 11. Clases

Una clase describe las variables, propiedades, procedimientos y eventos de un tipo de objetos.

Los objetos son instancias de clases, pudiendo crearse tantos objetos como sean necesarios de una determinada clase.

Las clases son tipos referencia.

Los miembros de clases de tipo variables y constantes son por defecto **Private**, el resto son **Public**.

Una variable de clase puede hacer referencia a varias instancias de clase en distintos momentos.

Si se asigna Nothing a una variable de clase, no se tiene acceso a los métodos ya que no hay instancia asociada.

Dos variables de clase se pueden comparar mediante el método Equals. Equals indica si las dos variables apuntan a la misma instancia.

```
Public Class employee

End Class
```

## 11.1. Miembros

Los miembros de una **Clase** pueden ser

- Campos
- Propiedades
- Metodos
- Eventos

## 11.2. Campos y Propiedades

Representan información almacenada en un objeto.

Un **Campo**, representa información asociada al objeto que se puede establecer directamente, se declaran como una variable a nivel de **Clase**, tambien se las llama "Variables miembro".

Normalmente los **Campos** serán **Private** o **Protected**, dado que siguiendo el paradigma de orientación a objetos no es interesante permitir el acceso directo a la información del objeto.

```
Public Class employee
    ' Decalracion de Campo
    Private salaryValue As Double
End Class
```

Una **Propiedad**, representa la forma de encapsular el acceso a información asociada al objeto, que para ser leida o establecida, se empleará una propiedad cuando se precise de algun tipo de procesamiento sobre el campo previo a la lectura o establecimiento, ya sea, por ejemplo, una validación para su establecimiento o un calculo para su lectura.

```
Public Class employee
    ' Declaración de propiedad sobre un campo privado con get y set
    Public Property salary() As Double
        Get
            Return salaryValue
        End Get
        Set(ByVal value As Double)
            salaryValue = value
        End Set
    End Property
End Class
```

Si se define una **Propiedad** como **ReadOnly**, no será obligatorio definir el **Set** y si se define como **WriteOnly**, no será necesario definir el **Get**, de no emplearse ninguno de estos modificadores, será obligatorio declarar tanto **Get** como **Set**

```
ReadOnly Property quoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

# 11.3. Métodos

Un método es una acción que puede realizar un objeto.

Los Métodos pueden ser de varios tipos

- Subprocedimientos (Sub)
- Funciones (Function)
- Procedimientos de control de eventos
- Get y Set asociados a una propiedad

Se puede aplicar **sobrecarga** a los métodos.

## 11.3.1. Subprocedimientos

Procedimientos que no retornan valor.

La forma de declararlos será

```
<Modificador de visibilidad> Sub <Nombre del procedimiento>(<variable> As <Tipologia
de la variable>, ...)

End Sub
```

Se puede terminar una subprocedimiento antes de terminar todas las sentencias con las sentencias de **Exit Sub** o **Return**.

Se puede definir un subprocedimiento en **Módulos**, **Clases** y **Estructuras**.

Son **Public** de forma predeterminada.

La invocación se hará a traves de la instancia de un objeto, on el operador punto (**.**).

### 11.3.2. Control de Eventos

Si el procedimiento representa el manejador de un evento de un componente, se puede añadir **Handles**

```
<Modificador de visibilidad> Sub <Nombre del procedimiento>(<variable> As <Tipologia
de la variable>, ...) Handles <El evento de que componente procesa>

End Sub
```

### 11.3.3. Funciones

Procedimientos que retornan valor.

La forma de declararlos será

```
<Modificador de visibilidad> Function <Nombre del procedimiento>(<variable> As
<Tipologia de la variable>, ...) As <Tipo retornado>
    Return <Objeto retornado>
End Function
```

Se puede terminar una subprocedimiento antes de terminar todas las sentencias con las sentencias de **Exit Function** o **Return**.

Son **Public** de forma predeterminada.

La invocación se hará a traves de la instancia de un objeto, on el operador punto (**.**).

## 11.4. Paso de parametros a métodos

### 11.4.1. Por referencia

Se emplea la palabra reservada **ByRef**, su uso indica que un parametro se pasa por referencia, es decir, se pasa la direccion de memoria donde esta el objeto, lo que implica que si la función que recibe el parametro lo cambia, cambia alguna caracteristica del objeto al que apunta el parametro, este objeto cambia de cara al exterior de la función.

### 11.4.2. Por valor

Cuando se utiliza este mecanismo para pasar argumentos, Visual Basic copia el valor del objeto pasado por parametro en una variable local de la función, por lo que la fución, no tiene acceso al objeto original, solo a la copia local, por lo tanto cualquier modificación se hace sobre la copia y no afecta al objeto original.

```
' Esta funcion recibe dos parametros, uno por referencia debt, que al recibirse
representa el valor original y al acabar la función representa el valor original mas
el tanto por ciento aplicado
' En cambio rate, se pasa por valor, ya que no se quiere que la función pueda
modificar el tanto por ciento aplicado
Sub Calculate(ByVal rate As Double, ByRef debt As Double)
        debt = debt + (debt * rate / 100)
End Sub
```

## 11.5. Constructores

Son los que permiten crear instancias (objetos) de las **Clases** definidas.

Los constructores en VB.NET son procediminetos **Sub** llamados **New**.

La primera linea del constructor debera ser siempre la invocación al constructor del padre, esto se consigue con la palabra reservada **MyBase**. De tener la clase padre un constructor sin parametros, esta linea es opcional, dado que el compilador la añade automaticamente.

```
'Constructor sin parametros
Public Sub New()
    ' MyBase.New() siempre debe estar en el constructor
    MyBase.New()
End Sub
```

Al igual que con los métodos, se puede aplicar **sobrecarga** a los constructores, pudiendo existir tantos constructores como queramos, siempre que varien en orden o número los parametros que reciben.

```
'Constructor con parametros
Public Sub New(propiedad As String)
    ' MyBase.New() siempre debe estar en el constructor
    MyBase.New()

    Me.propiedad = propiedad

End Sub
```

Para la construcción de un objeto, se emplea la palabra reservada **New**

```
Dim variable As New String("texto de la variable")
```

## 11.6. Casting

Dado que se pueden tener referencias a objetos con variables de distinto tipo, en ocasiones es necesario realizar conversiones de tipo, para ello VB.NEt proporciona el operador **CType**

```
Dim entero As Integer = 1

Dim cadena As String = CType(i, String)
```

No todos los tipos son convertibles en todos los demas tipos, dado que no tiene porque estar definido el operador **CType** que relaciones el tipo origen y el destino.

Los tipos basicos se pueden convertir entre si y cuando hay herencia tambien.

# 12. Estructuras

Las estructuras son tipo valor, una variable de un tipo de estructura contiene los datos de la estructura, en lugar de contener una referencia a los datos, como sucede con los tipos de clase.

Además de campos, las estructuras pueden exponer propiedades, métodos y eventos.

```
Private Structure employee
    Public givenName As String
    Public familyName As String
    Public phoneExtension As Long
    Private salary As Decimal
    Public Sub giveRaise(raise As Double)
        salary *= raise
    End Sub
    Public Event salaryReviewTime()
End Structure
```

No se puede heredar de una estructura, aunque una estructura si puede implmentar una **Interface**.

Las estructuras heredan de forma implicita de System.ValueType.

Se recomienda emplear estructuras cuando el tipo de dato definido sea pequeño.

Por defecto todos los miembros de una estructura son **Public**.

Una estructura debe tener al menos una variable no compartida.

Los elementos de estructura no se pueden declarar como Protected.

Las estructuras no son recolectadas por el GC.

Las estructuras no requieren de un constructor, ya que podeen de forma implicita el constructor por defecto, que no puede ser redefinido.

Cada variable de estructura está enlazada de forma permanente a una instancia de estructura individual.

Si se asigna Nothing a una variable de estructura, no se elimina el enlace con la instancia, aunque se inicialicen los miembros, por lo que se tiene acceso a los métodos.

La comprobación de igualdad de dos instancias de estructura debe realizarse mediante una prueba elemento a elemento.

# 13. Modulos

# 14. Namespace

Permiten organizar los elementos definidos en un ensamblado, es una forma de agrupar **Clases**, **Interfaces** y demas elementos definibles en un ensamblado.

Se emplea la palabra reservada **Namespace**, creando un bloque de código que contendra los elementos ordenados dentro de ese **Namespace**.

```
Namespace Entidades

    Public Class Persona

    End Class

End Namespace
```

Su principal misión, es que no haya conflictos con los nombres, dado que el nombre de los elementos, sera el de su Namespace mas el del propio elemento.

```
' Para declarar una variable del tipo anteriormente descrito, es preciso hacer
referencia al nombre completo
Dim p As Entidades.Persona
```

Si se hace referencia a un tipo dentro del mismo **Namespace** donde es declarado, no es necesario indicarlo.

Los **Namespace** se pueden anidar, creando estructuras mas comlejas de organizacion

```
Namespace Aplicacion

    Namespace Entidades

        Public Class Persona

        End Class

    End Namespace

End Namespace
```

Este anidamiento puede provocar que los nombres completos sean muy largos, y por tanto dificiles de escribir, para evitar tener que escribir continuamente los nombres se pueden crear alias de esto con la palabra reservada **Imports**

```
Imports Aplicacion.Entidades

Sub Main()

    Dim p As Persona

End Sub
```

En Visual Studio, en las propiedades del proyecto, se define un **Namespace Raiz**, que es el que tendran de forma implicita todos los elementos definidos dentro del proyecto (ensamblado)

# 15. Palabras reservadas

## 15.1. Me

Palabra reservada que permite referenciar a la instancia actual de la clase donde se este ejecutando.

## 15.2. MyBase

Hace referencia a la parte de la instancia que corresponde a la clase padre, se emplea en los constructores y en los métodos sobreescritos.

## 15.3. New

Permite instanciar objetos de clases. Se pueden crear objetos en el momento de la declaración de una variable, o en cualquier otro momento.

```
' Declaración de variable, creación de objeto y asignación del objeto a la variable
Dim variable As New String("texto de la variable")

' Declaración de variable
Dim otra As String

' Creación de objeto y asignación a la variable
otra = New String("valor de otra variable")
```

## 15.4. Partial

Permite dividir la definición de un tipo en varias declaraciones.

Los tipos que se pueden dividir son: **Class**, **Structure**, **Interface** y **Module**.

```
Partial Public Class sampleClass
    Public Sub sub1()
    End Sub
End Class
Partial Public Class sampleClass
    Public Sub sub2()
    End Sub
End Class
```

Por ejemplo, el Diseñador de Windows Forms define clases parciales para controles como Form.

## 15.5. ReadOnly

Especifica que una variable o una propiedad se puede leer, pero no se puede modificar.

No se puede especificar **ReadOnly** junto con **Static** en la misma declaración.

Sólo puede asignar un valor a una variable **ReadOnly** en su declaración o en el constructor de una clase o estructura en la que está definida.

## 15.6. WriteOnly

Especifica que se puede escribir pero no leer una propiedad.

Se puede declarar una propiedad como WriteOnly, pero no una variable.

## 15.7. Shared

Indica que un atributo de clase es compartido por todas las instancias de esa clase.

## 15.8. Static

Especifica si una variable local a un método debe seguir existiendo y conservar su último valor tras la finalización del método en el que se han declarado.

Normalmente, una variable local de un método deja de existir en cuanto finaliza el método, en cambio una variable estática sigue existiendo y conserva su valor más reciente, por lo que la próxima vez que se invoque el método, la variable no se reinicializa, conservando el último valor asignado.

```
' totalSales solo se inicializa a 0 una v ez, la primera vez que se invoca el método,
el resto de ocasiones en las que se invoca el método, el valor de totalSales, será el
último que tuviese.
Function updateSales(ByVal thisSale As Decimal) As Decimal
    Static totalSales As Decimal = 0
    totalSales += thisSale
    Return totalSales
End Function
```

## 15.9. Nothing

Palabra reservada que representa NULL.

Los tipos referencia aceptan el valor Nothing.

Para que un tipo valor acepte Nothing, en la declaración, hay que indicarlo con el caracter **?**

## 15.10. Optional

Permite indicar si un parametro es obligatorio o es opcional.

De definirse el parametro como opcion, será obligatorio indicar un valor por defecto para dicho parametro.

```
Sub notify(Optional ByVal office As String = "QJZ")

End Sub
```

# 16. Enumeraciones y constantes

Permiten definir grupos de constantes.

```
Public Enum Mammals
    Buffalo
    Gazelle
    Mongoose
    Rhinoceros
    Whale
End Enum
```

Son de tipo Integer por defecto, aunque pueden ser de cualquiera e los tipos enteros Byte, Short, Long o Integer.

```
Public Enum Mammals As Integer
    Buffalo
    Gazelle
    Mongoose
    Rhinoceros
    Whale
End Enum
```

A cada uno de los elementos de la enumeración se le puede asignar un valor entero.

```
Public Enum Mammals As Integer
    Buffalo = 1
    Gazelle
    Mongoose = -3
    Rhinoceros
    Whale
End Enum
```

Todos los enum heredan de System.Enum, que proporciona una serie de métodos como **GetNames** que permite obtener la representacion en **String** del valor enumerado.

Para iterar sobre un enumerado, se emplea la clase System.Enum

```
Dim items As Array
items = System.Enum.GetValues(GetType(Mammals))
Dim item As String
For Each item In items
    Console.WriteLine(item)
Next
```

Se pueden definir constantes individuales con la palabra reservada **Const**, tanto a nivel de campo de clase, como de variable.

```
Public Const DaysInYear = 365
```

# 17. Excepciones

Las excepciones son eventos que se producen en el código, que indican que ha sucedido algo, que no entra dentro de la ejecución **normal** del codigo.

El API de VB.NET, lanza excepciones en alguna de sus fundionalidades, por lo que habrá que tenerlo en cuenta.

El propio programador, puede establecer cuando quiere que se genere una excepcion, para ello ha de crear un objeto de tipo Exception y lanzarlo

```
Throw New NullReferenceException("Something happened.")
```

Se pueden hacer dos cosas con las excepiones

- Propagarlas
- Capturarlas

La propagación en VB.NET es automatica, no hay que indicar nada.

Para capturarlas, se emplea la sentencia de control **Try...Catch...Finally**, donde se prueba a ejecutar, el contenido de **Try**, si durante la ejecución se produce una excepción, se ejecutará el bloque de **Catch** y sino no, y pase lo que pase, se ejecutará siempre **Finally** al finalizar de ejectar el bloque.

Cuando se produce una excepcion en una sentencia el resto de sentencias del bloque, no se ejecuta.

```
Public Sub TryExample()
    ' variables.
    Dim x As Integer = 5
    Dim y As Integer = 0

    ' Estructura de captura de excepciones.
    Try
        ' Esta sentencia genera una excepcion de division por 0.
        x = x \ y

        ' Estas sentencia no se ejecutan porque ocurre una excepcion.
        MessageBox.Show("Fin del bloque Try")
    Catch ex As Exception
        ' Se muestra el mensaje asociado a la excepcion.
        MessageBox.Show(ex.Message)

        ' Se muesra la traza de la excepción, que puede dar muestras de donde
               ' viene dicha excepcion.
        MessageBox.Show("Stack Trace: " & vbCrLf & ex.StackTrace)
    Finally
        ' Esta sentencia se ejecuta hay o no excepcion.
        MessageBox.Show("Dentro del bloque Finally")
    End Try
End Sub
```

Pueden existir multiples bloques de **Catch**

```
Public Sub RunSample()
    Try
        CreateException()
    Catch ex As System.IO.IOException
        ' Este bloque se ejecuta si se produce una IOException.
    Catch ex As NullReferenceException
        ' Este bloque se ejecuta si se produce una NullReferenceException.
    Catch ex As Exception
        ' Este bloque se ejecuta si se produce cualquier otra Excepcion, ya que
               ' todas heredan de Exception.
    End Try
End Sub
```

Se pueden establecer condiciones para ejecutar el bloque **Catch**

```
Private Sub WhenExample()
    Dim i As Integer = 5

    Try
        Throw New ArgumentException()
    Catch e As OverflowException When i = 5
        Console.WriteLine("First handler")
    Catch e As ArgumentException When i = 4
        Console.WriteLine("Second handler")
    Catch When i = 5
        Console.WriteLine("Third handler")
    End Try
End Sub
```

# 18. Colecciones

Son una forma que proporciona el API de VB.NET de tener agrupados varios objetos de una misma tipologia.

Son muy parecidas a los Array, salvo por que si bien los **Array** tienen un tamaño fijo durante toda la vida del objeto, las **Colecciones** pueden cambiar de forma dinamica su tamaño.

Se basan en las clases dentro del **Namespace** llamado **System.Collections**

El API se basa en las siguientes **Interfaces**

- System.Collections.ICollection → Interface general para las colecciones.
- System.Collections.IList → Interface para modelar colecciones de elementos ordenados por indice
- System.Collections.IDictionary → Interface para modelar colecciones de pares, clave y valor
- System.Collections.Generic.ICollection
- System.Collections.Generic.IList
- System.Collections.Generic.IDictionary
- System.Collections.Generic.ISet → Interface par modelar colecciones de elementos no repetibles

Algunas de las clases que nos podemos encontrar en ese paquete son

- System.Collections.ArrayList → Implmentación de **IList** y de **ICollection**, basado en un **Array** interno que se redimensiona segun las necesidades
- System.Collections.Hashtable → Implementación de **IDictionary** y **ICollection**, que emplea el Hash de los objetos para organizarlos dentro de la coleccion y que sea mas eficiente la búsqueda.
- System.Collections.Queue → Implementacon de **Collection** que modela una **Cola**
- System.Collections.Stack → Implementacion de **ICollection** que modela una **Pila**

- System.Collections.SortedList → Implementacion de **IDictionary** y **ICollection** ordenados por clave.
- System.Collections.Generic.Dictionary
- System.Collections.Generic.HashSet
- System.Collections.Generic.LinkedList
- System.Collections.Generic.List
- System.Collections.Generic.Queue
- System.Collections.Generic.SortedList
- System.Collections.Generic.SortedDictionary
- System.Collections.Generic.SortedSet

Las colecciones **Generic**, siguen la misma pauta que las normales, pero limitan el tipo de datos que contienen en el momento de la declaración dle objeto.

Para definir las collecciones

```
'Collection
Dim coleccion As Collection = New Collection

'Normales
Dim arrayList As ArrayList

Dim hashTable As Hashtable

Dim queue As Queue

Dim stack As Stack

Dim sortedList As SortedList

'Generic
Dim dictionaryGeneric As Dictionary(Of String, String)

Dim hashSetGeneric As HashSet(Of String)

Dim linkedListGeneric As LinkedList(Of String)

Dim listGeneric As List(Of String) = New List(Of String)

Dim queueGeneric As Queue(Of String)

Dim sortedListGeneric As SortedList(Of String, String)

Dim sortedDictionary As SortedDictionary(Of String, String)

Dim sortedSet As SortedSet(Of String)
```

Para añadir elementos a un collecion

```
arrayList.Add("dato")

hashTable.Add("key", "value")

queue.Enqueue("dato")

stack.Push("dato")
```

Para eliminar elementos

```
arrayList.Remove("dato")
' Borra el eleemnto por su posición
arrayList.RemoveAt(0)

hashTable.Remove("key")
'Borra y retorna el objeto al principio de la cola
queue.Dequeue()
'Retorna el objeto al principio de la cola sin borrarlo
queue.Peek()
'Borra y retorna el objeto en la tapa de la pila
stack.Pop()
'Retorna el objeto en la tapa de la pila sin borrarlo
stack.Peek()
```

Para recorrer las colecciones, se pueden emplear estructuras de For Each

```

```

Especial mención a la clase **Microsoft.VisualBasic.Collection**, que es una clase que permite definir varios tipos de colecciones.

# 19. ADO.NET

Existen varias posibilidades de base de datos

- LocalDB, es representado por un fichero local con extension mdf
- SQL Server
- ODBC
- Oracle

## 19.1. Componentes ADO.NET

- Proveedores de datos

- Connection → Proporciona conectividad a un origen de datos.
- Command → Representa sentencias de SQL sobre la BD.
- DataReader → Proporciona un flujo de datos de entrada de alto rendimiento desde el origen de datos (BD) a la aplicación.
- DataAdapter → Proporciona un puente entre el origen de datos (BD) y el **DataSet**, a través de **Command**.
- DataSet → Conjunto de objetos **DataTable**, representa una copia en memoria de los datos de la base de datos.
    - DataTable → Representa una Tabla de una base de datos en memoria.

Para poder emplear el API para el caso de SQLServer, se han de añadir los siguientes **Imports**

```
Imports System.Data.SqlClient
Imports System.Data
```

## 19.2. Connection

Representan la conexion a la base de datos, dado que la base de datos no admite infinitas conexiones, se ha de cuidar no dejarlas abiertas y cerrarlas cuando no se necesiten, para ello y aprovechando que implementan **IDisposable**, se puede emplear la estructura **Using...End Using**, que garantiza la invocación del método **close** de **IDiposable**.

No se puede reabrir un objeto **Connection** una vez cerrado, se ha de crear un nuevo objeto.

Una parte importante de la conexión, es la cadena de conexión, que dependerá del origen de datos al que se realice la conexión.

La cadena de conexión es la representación en String de la conexion sobre un origen de datos.

Un ejemplo de conexión sobre una BD SQLServer contenida en un fichero mdf, sería

```
Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename="D:\Dropbox\Contenido\dotNET\Visual
Basic\BaseDeDatos.mdf";Integrated Security=True;Connect Timeout=30
```

Visual Studio, permite facilmente obtenerlas y almacenarlas en el fichero de configración **App.config**.

```
<configuration>
    <connectionStrings>
        <add name="PersonasConnectionString"
            connectionString="Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=&quot;D:\BaseDeDatos.mdf&quot;;Integrat
ed Security=True;Connect Timeout=30"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

Para cargar una connectionString del fichero de configuracion, hay que añadir la referencia al proyecto de **System.Configuracion** y añadir el Imports a la clase correspondiente

```
Imports System.Configuracion
```

Para recuperarla, se emplea la clase **ConfigurationManager**

```
Dim stCon As String =
ConfigurationManager.ConnectionStrings("PersonasConnectionString").ConnectionString
```

### 19.2.1. Conexiones sobre Excel

Se pueden establecer conexiones sobre un Excel con ADO.NET, tomando este como la base de datos, con las restricciones que Excel tiene.

Para versiones de Excel entre la 8.0 y la 10.0, se empleará la cadena de conexión

```
Dim cadconex As String = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & ruta &
";Extended Properties=""Excel 8.0;HDR=Yes;IMEX=1"""
```

Para versiones de Excel superior a la 10.0, se empleará la cadena de conexión

```
Dim cadconex As String = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" & ruta &
";Extended Properties=""Excel 12.0 Xml;HDR=YES;IMEX=0"""
```

Para este último caso, existe un problema en los equipos de x64, ya que no hay disponible Drivers, y habrá que instalar los de x86, que se pueden descargar **aqui**, donde habrá que seleccionar la versión de descarga de x86.

Las consulas a crear tienen una serie de caracteristicas por no ser Excel una base de datos relacional, ni seguir el lenguaje SQL.

Así pues, se considerara una tabla, o bien una hoja entera, o un subconjunto de ella, indicandolo en las consultas con la siguiente sintaxis

- Una tabla es una hoja

```
select * from [Clientes$]
```

Donde podemos **Clientes** es el nombre de la hoja Excel.

- Una tabla es un subcojunto de celdas de una hoja

```
select * from [Clientes$A1:C4]
```

Donde se indica que el conjunto de celdas comprendidas entre las columnas A y C y las filas 1 y 4, forman la tabla.

En todo caso, la primera fila, se tomará como los nombres de los campos. Estos nombre de los campos, no debeán tener espacios en blanco.

De no querer que la primera fila sea la cabacera, se ha de indicar con HDR=NO en las propiedades extendidas de la cadena de conexión. En caso de no tener cabecera, el proveedor OLE DB denomina automáticamente a los campos en su lugar (donde F1 representaría el primer campo, F2 representaría el segundo campo, etcétera).

Los tipos de datos para cada campo, son adivinados por ADO.NET, inspeccionando los valores de 8 campos, se puede modificar el numero de campos inspeccionado para obtener el tipo, con MAXSCANROWS=<numero> en las propiedades extendidas de la cadena de conexión.

### 19.2.2. Conexiones sobre DB2

**Aquí** se pueden encontrar los enlaces de descarga para los Driver de DB2, para las distitnas tecnologias, incluida .NET.

**Aquí** se pueden acceder a ejemplos que proporciona IBM, para el acceso a datos en DB2, emplean ADO.NET.

## 19.3. Command

Representan una consulta sobre la base de datos.

Pueden ser de tres tipos

- CommandType.Text → Consulta SQL
- CommandType.StoredProcedure → Nombre de un procedimiento almacenado
- CommandType.TableDirect → Nombre de una tabla

Se pueden ejecutar de tres formas

- ExecuteNonQuery → Se emplea para ejecutar sentencias de tipo INSERT, UPDATE y DELETE, que cambian el estado de la BD.

- ExecuteReader → Retorna un objeto **Reader**, con los registros retornados por la consulta, se emplea para sentencias de tipo SELECT.
- ExecuteScalar → Retorna el primer valor del conjunto de registros.

Los **Command** son parametrizables, se pueden definir en el string que representa la consulta parametros indicados con el prefijo @, y posteriormente ser sustituidos.

Ejemplo de Command que realiza una consulta

```
Dim con = New SqlConnection(stCon)

Dim queryVictor As String = "Select * from personas where nombre = @nombre"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryVictor, con)

    Dim param As SqlParameter = New SqlParameter("@nombre", "Victor")

    command.Parameters.Add(param)

    Dim reader = command.ExecuteReader()

    Dim tablaPersonasConNombreVictor = New DataTable("PersonasConNombreVictor")

    tablaPersonasConNombreVictor.Load(reader)

    DataGridView3.DataSource = tablaPersonasConNombreVictor

End Using
```

Ejemplo de Command que realiza un borrado

```
Dim con As SqlConnection = New SqlConnection(stCon)

Dim queryDeletePersona As String = "Delete from personas where nombre = @nombre"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryDeletePersona, con)

    Dim paramNombre As SqlParameter = New SqlParameter("@nombre", "Jacinto")

    command.Parameters.Add(paramNombre)

    command.ExecuteNonQuery()

End Using
```

Ejemplo de Command que realiza un alta

```
Dim con = New SqlConnection(stCon)

Dim queryInsertPersona As String = "Insert into personas (id, nombre)  values (@id,
@nombre)"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryInsertPersona, con)

    Dim paramNombre As SqlParameter = New SqlParameter("@nombre", "Jacinto")
    Dim paramId As SqlParameter = New SqlParameter("@id", "9")

    command.Parameters.Add(paramNombre)
    command.Parameters.Add(paramId)

    command.ExecuteNonQuery()

End Using
```

## 19.4. DataReader

Representa un conjunto de registros leidos de la base de datos, es un objeto que solo permite leer de la base de datos.

Se puede rellenar un **DataTable** directamente con los registros retornados.

```
Dim con = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryPersonas, con)

    Dim reader = command.ExecuteReader()

    Dim tablePersonas = New DataTable("MisPersonas")

    tablePersonas.Load(reader)

    DataGridView1.DataSource = tablePersonas

End Using
```

Se puede recorrer, creando objetos por cada registro y volcandolos sobre una colleción.

```
Dim con As SqlConnection = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryPersonas, con)

    Dim reader = command.ExecuteReader()

    Dim listaPersonas As List(Of Persona) = New List(Of Persona)

    If reader.HasRows Then
        Do While reader.Read()
            listaPersonas.Add(New Persona(reader.GetInt32(0), reader.GetString(1)))
        Loop
    End If

    DataGridView4.DataSource = listaPersonas

End Using
```

## 19.5. DataAdapter

Es el puente entre el **DataSet** y el **DataSource**.

Permite sincronizar en las dos direcciones

- Fill → Rellena el **DataSet** con los datos del **DataSource**
- Update → Actualiza el **DataSource** para que los datos de la BD coincidan con los del **DataSet**.

## 19.6. DataSet

Contiene n objetos DataTable, que estan formados por filas **DataRow** y columnas de datos **DataColumn**.

Los DataSet permiten

- Almacenar datos en la memoria caché de la aplicación para poder manipularlos. Si solamente se necesita leer los resultados de una consulta, DataReader es mejor elección.
- Utilizar datos de forma remota entre un nivel y otro o desde un servicio Web XML.
- Interactuar con datos dinámicamente, por ejemplo para enlazar con un control de Windows Forms o para combinar y relacionar datos procedentes de varios orígenes.
- Realizar procesamientos exhaustivos de datos sin necesidad de tener una conexión abierta con el origen de datos, lo que libera la conexión para que la utilicen otros clientes.

```
Dim con As SqlConnection = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Dim adapterPersonas As New SqlDataAdapter(queryPersonas, con)

Using con

    con.Open()

    adapterPersonas.Fill(ds, "MisPersonas")

    DataGridView1.DataSource = ds.Tables("MisPersonas")

End Using
```

# 20. Windows Forms

API de .NET que permite definir aplicaciones de escritorio.

## 20.1. DataGridView

Componente visual, que permite representar tablas de datos.

Se ha de asociar la propiedad **DataSource** con un **DataTable**, este puede ser obtenido con

- DataReader

```
Dim con = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryPersonas, con)

    Dim reader = command.ExecuteReader()

    Dim tablePersonas = New DataTable("MisPersonas")

    tablePersonas.Load(reader)

    DataGridView1.DataSource = tablePersonas

End Using
```

- DataSet

```
Dim con As SqlConnection = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Dim adapterPersonas As New SqlDataAdapter(queryPersonas, con)

Using con

    con.Open()

    adapterPersonas.Fill(ds, "MisPersonas")

    DataGridView1.DataSource = ds.Tables("MisPersonas")

End Using
```

- Coleccion

```
Dim con As SqlConnection = New SqlConnection(stCon)

Dim queryPersonas As String = "Select * from personas"

Using con

    con.Open()

    Dim command As SqlCommand = New SqlCommand(queryPersonas, con)

    Dim reader = command.ExecuteReader()

    Dim listaPersonas As List(Of Persona) = New List(Of Persona)

    If reader.HasRows Then
        Do While reader.Read()
            listaPersonas.Add(New Persona(reader.GetInt32(0), reader.GetString(1)))
        Loop
    End If

    DataGridView4.DataSource = listaPersonas

End Using
```

De producirse cambios en el origen de datos y querer que el DataGrid se entere, se ha de refrescar

```
DataGridView1.Refresh()
```

Se puede desactivar la fila para añadir elementos, seteando la propiedad **AllowUserToAddRows** del **DataGridView**, a False.

Tambien se puede indicar que aunque este presente dicha fila, no sean sus campos editables, con la propiedad **ReadOnly** a True.

Incluso se podria indicar que solo unos campos son editables, accediendo a las **DataColumn** del **DataGrid** y seteando la propiedad **ReadOnly** a True.