Anexo IV

Visual Basic.Net

Guion-resumen

1. **Introducción a VB.Net**
2. **Comentarios**
3. **Variables**
4. **Operadores**
5. **Sentencias o expresiones**
6. **Arrays**
7. **Entrada / Salida**
8. **Clases en VB.Net**
9. **Assembly (ensamblado)**
10. **Espacios de nombres (Namespace)**
11. **Excepciones**

1. Introducción a VB.Net

Desde que Microsoft liberó Visual Basic 1.0 en 1991 han tenido lugar muchos cambios. Visual Basic 1.0 revolucionó la forma de desarrollar software para Windows, desmitificó el proceso de desarrollo de aplicaciones con interfaz gráfica del usuario y abrió este tipo de programación a las masas. En sus posteriores versiones, Visual Basic ha continuado proporcionando nuevas y nuevas características que facilitaron la creación de aplicaciones para Windows cada vez más potentes; por ejemplo la versión 3.0 introdujo el control de datos para facilitar el acceso a Bases de datos y la versión 4.0 mejoró y potenció este acceso con los objetos ADO. Con la aparición de Windows 95, Microsoft liberó Visual Basic 4.0 que abrió la puerta al desarrollo de aplicaciones de 32 bits y a la creación de DLL. La Versión 5.0 mejoró la productividad con la incorporación de la ayuda inteligente y a introducción de los controles ActiveX. Finalmente la versión 6.0 nos introdujo en la programación de Internet con las aplicaciones DHTML y el objeto Web-Class. Y ahora disponemos de la versión 7.0, o simplemente Visual Basic.NET que viene a revolucionar el mundo de las comunicaciones permitiendo escribir aplicaciones escalables para Internet. Y por supuesto, totalmente orientado a objetos, como su homónimo JAVA.

La palabra «Visual» hace referencia, desde el lado del diseño, al método que le utiliza para crear la interfaz gráfica de usuario si se dispone de la herramienta adecuada (con Microsoft Visual Studio .NET se utiliza el ratón para arrastrar y colocar los objetos prefabricados en el lugar deseado dentro de un formulario). Y desde el lado de la ejecución, al aspecto gráfico que toman los objetos cuando se ejecuta el código que los crea, objetos que formarán la interfaz gráfica que el usuario de la aplicación utiliza para acceder a los servicios que esta ofrece. La palabra «Basic» hace referencia al lenguaje BASIC (Beginners All-Purpose Symbolic Instruction Code), un lenguaje utilizado por más programadores que ningún otro lenguaje en la historia de la informática. Visual Basic ha evolucionado a partir del lenguaje BASIC original y ahora contiene centenares de instrucciones, funciones y palabras clave, muchas de las cuales están directamente relacionadas con la interfaz gráfica de Windows.

La palabra «NET» hace referencia al ámbito donde operarán nuestras aplicaciones (Network). Visual Basic.NET proporciona la tecnología necesaria para saltar desde el desarrollo de aplicaciones cliente-servidor tradicionales a la siguiente generación de aplicaciones escalables para la Web, introduciendo algunos conceptos nuevos, como ensamblados, formularios Web, servicios Web, ADO.NET y .NET Framework. Es importante saber también que la inversión realizada en el aprendizaje de Visual Basic ayudará a abarcar otras áreas, porque este lenguaje de programación es utilizado también por Microsoft Excel, Microsoft Access y muchas otras aplicaciones Windows.

1.1. Extensión de los ficheros de código

En Visual Basic .NET, a diferencia de lo que ocurría en las versiones anteriores de Visual Basic, solo existe un tipo de fichero de código, el cual tiene la **extensión .vb.** En este tipo de fichero pueden coexistir distintos tipos de elementos (por ejemplo: un módulo de clase, un formulario, un módulo de

código, un control, etc.), mientras que en las versiones anteriores de Visual Basic, cada uno de estos elementos tenían su propio tipo de fichero con su respectiva extensión.

1.2. Tipos de ejecutables

Como sucede con JAVA, en Visual Basic .NET puede crear básicamente estos dos tipos de ejecutables:

- Consola.
- Gráficos, con una interfaz gráfica (GUI).

Existen otros tipos de aplicaciones que se pueden crear con Visual Basic .NET: aplicaciones ASP.NET, servicios Web, servicios Windows, controles, componentes, DLL's, etc.

2. Comentarios

Los comentarios empiezan por una comilla simple (apóstrofe). En los comentarios podemos poner lo que queramos, con la seguridad de que no será tenido en cuenta por el Visual Basic. Los comentarios solo pueden ocupar una línea, salvo que dicha línea al final tenga el signo _ (guión bajo), lo cual indica al Entorno de Desarrollo (IDE) que se quiere continuar en la siguiente línea. Ese símbolo se puede llamar «continuador de línea» y lo podemos usar siempre que queramos, no solo para los comentarios. Los comentarios también se pueden hacer con la palabra reservada Rem, aunque se encuentra en desuso, viene de versiones anteriores.

3. Variables

Existen distintos tipos de datos que VB.NET maneja. No es obligatorio declarar las variables según el tipo de datos que va a almacenar. Para declarar una variable se usa la sintaxis:

Especificador_acceso Nombre_variable Tipo

Ejemplos:

Dim i As Integer	*Private x As Double*	*Public cad As String*

Los especificadores de acceso los veremos más adelante; aquí veremos los diferentes tipos que puede tomar una variable. Recuerde que son los tipos básicos, ya que en la programación orientada a objetos, nosotros nos podemos crear nuestros propios tipos compuestos (clases).

TIPO DE VISUAL BASIC	ESPACIO DE MEMORIA QUE OCUPA
Boolean	2 bytes
Byte	1 byte
Char	2 bytes
Date	8 bytes
Decimal	16 bytes
Double	8 bytes
Integer	4 bytes
Long (entero largo)	8 bytes
Object	4 bytes
Short (entero corto)	2 bytes
Single	4 bytes

3.1. Options

Se pueden declarar variables sin tipo específico: **Dim x**, que en realidad es como si se hubiese declarado del tipo Object (As Object), por tanto aceptará cualquier tipo de datos.

Los options permiten decirle al compilador una serie de parámetros sobre restricciones. Deben ir en el fichero de código al principio del mismo. Visual Basic no obliga a que se declaren todas las variables que vayamos a usar. La instrucción **Option Explicit** obliga a que declaremos las variables.

Con **Option Strict On**, obligará a que los tipos de datos que uses sean del tipo adecuado. Por ejemplo, con el Option Strict On no podemos hacer esto:

Dim c As Char = «TAI», ya que «TAI» es del tipo String.

La instrucción **Option Compare** dependerá de si se quiere que las cadenas se comparen diferenciando las mayúsculas de las minúsculas o no. Con el valor Binary se diferencian las mayúsculas de las minúsculas y con el otro valor, Text, no se hace ningún tipo de distinción.

Las variables se pueden declarar de dos formas:

- Declarando la variable y dejando que VB asigne el valor por defecto.
- Declarando la variable y asignándole el valor inicial que queramos que tenga.

Por defecto, cuando no se asigna un valor a una variable, estas contendrán los siguientes valores, dependiendo del tipo de datos que sea:

- Las variables numéricas tendrán un valor 0.
- Las cadenas de caracteres una cadena vacía: «».
- Las variables Boolean un valor False.
- Las variables de tipo Objeto tendrán un valor Nothing.

Por ejemplo: Dim i As Integer Tendrá un valor inicial de 0.

Pero si queremos que inicialmente valga 27, podemos hacerlo de cualquiera de estas dos formas:

Dim i As Integer i = 27	*Dim i As Integer = 27*

Las constantes se declaran: Const x As Integer = 7.

Para declarar una constante de tipo String, lo haremos de esta forma: Const cad As String = «TAI».

De igual manera, para declarar una variable de tipo String y que contenga un valor, lo haremos de esta forma: *Dim Nombre As String = «Holita».*

Podemos usar cualquier constante o variable en las expresiones, e incluso, podemos usar el resultado de esa expresión para asignar un valor a una variable. Por ejemplo:

Dim x As Integer = 27
Dim i As Integer
*i = x * 2*

3.2. Funciones de conversión

Existen unas funciones de conversión, que sirven para pasar datos de un tipo a otro.

Por tanto, esto: *i = Val(«10 * 25»)* es lo mismo que esto otro: i = Val(«10»)

En este caso, usamos la función Val para convertir una cadena en un número, pero ese número es del tipo Double y si tenemos Option Strict On, no nos dejará convertirlo en un Integer.

Para solucionarlo, usaremos la función CType: *i = CType(Val(«10 * 25»), Integer).*

Con esto le estamos diciendo al VB que primero convierta la cadena en un número mediante la función Val (que devuelve un número de tipo Double), después le decimos que ese número Double lo convierta en un valor Integer.

Funciones de conversión de tipos:

NOMBRE DE LA FUNCIÓN	TIPO DE DATOS QUE DEVUELVE
CBool(*expresion*)	Boolean
CByte(*expresion*)	Byte
CChar(*expresion*)	Char
CDate(*expresion*)	Date
CDbl(*expresion*)	Double
CDec(*expresion*)	Decimal
CInt(*expresion*)	Integer
CLng(*expresion*)	Long
CObj(*expresion*)	Object
CShort(*expresion*)	Short
CSng(*expresion*)	Single
CStr(*expresion*)	String
CType(*expresion*, Tipo)	El indicado en el segundo parámetro
Val(*expresion*)	Double
Fix(*expresion*)	Depende del tipo de datos de la expresión
Int(*expresion*)	Depende del tipo de datos de la expresión

Aunque si esos números son negativos, Fix devuelve el siguiente valor igual o mayor que el número indicado, mientras que Int lo hace con el primer número menor o igual.

Por ejemplo*: Fix(-8.4) devuelve -8, mientras que Int(-8.4) devolverá -9*.

En caso de que sean positivos, las dos funciones devuelven el mismo valor:

Int(8.4) devuelve 8, lo mismo que Fix(8.4).

Podemos declarar varios tipos de variables en una misma línea: *Dim i As Integer, s As String.*

En este caso, tenemos dos variables de dos tipos distintos, cada una con su As tipo correspondiente, pero separadas por una coma: *Dim j, k As Integer, cad1, Nombre As String, d1 As Decimal.*

En esta ocasión, las variables j y k son del tipo Integer, las variables cad1 y Nombre del tipo String y, por último, la variable d1 es de tipo Decimal.

No se permite la inicialización explícita con varios declaradores. Ejemplo: D*im i, j As Integer=10* dara error.

Deberíamos hacerlo de esta otra forma: *Dim i As Integer, j As Integer = 1.*

No tienen por qué ser del mismo tipo:

Dim x As Integer = 25, cad1, cad2 As Long, cad3 As String = «TAI», d2, d3 As Decimal

Que sería lo mismo que:

Dim x As Integer = 25
Dim cad1, cad2 As Long
Dim cad3 As String = «TAI»
Dim d2, d3 As Decimal

3.3. Visibilidad de las variables

Las variables declaradas dentro de un procedimiento se dicen que son «locales» a ese procedimiento y, por tanto, solo visibles (o accesibles) dentro del procedimiento en el que se ha declarado.

Al mismo tiempo, una variable local puede ocultar a otra variable. Cuando se declara una variable dentro de un procedimiento (Sub, función o propiedad), esa variable oculta a otras variables que, teniendo el mismo nombre, pudieran existir a nivel de módulo o a un nivel «superior».

ÁMBITO DE LAS VARIABLES LOCALES	VARIABLES QUE OCULTAN A OTRAS VARIABLES
Option Strict On *Module Module1* *' Variable declarada a nivel de módulo* *Dim n As Integer = 15* *Sub Main()* *' Variable declarada a nivel de procedimiento* *Dim i As Long = 10* *' Esto mostrará que n vale 15* *Console.WriteLine(«El valor de n es: {0}», n)* *Console.WriteLine(«El valor de i es: {0}», i)* *Console.ReadLine()* *End Sub* *Sub func()* *' Esto mostrará que n vale 15* *Console.WriteLine(«El valor de n es: {0}», n)* *' Error, ya que la variable i no está declarada* *Console.WriteLine(«El valor de i es: {0}», i)* *Console.ReadLine()* *End Sub* End Module	Option Strict On Module Module1 ' Variable declarada a nivel de módulo Dim n As Integer = 27 Sub Main() 'Mostrará n como 27 Console.WriteLine(«El valor de n Main es: {0}», n) ' Console.ReadLine() End Sub Sub func() Dim n As Long = 7 ' Mostrará n como 7 Console.WriteLine(«El valor de n func es:{0}», n) Console.ReadLine() End Sub End Module

4. Operadores

Los operadores VB.NET se dividen en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos y de desplazamiento y de asignación.

A) Operadores aritméticos

OPERADOR	USO	DESCRIPCIÓN
+	obj1 + op2	Suma obj1 y op2
-	obj1 - op2	Resta op2 de obj1
*	obj1 * op2	Multiplica obj1 y op2
/	obj1 / op2	Divide decimal o flotante
Mod	obj1 Mod op2	Módulo
\	obj1 \ op2	División entera
&	cad1&cad2	Concatenación de cadenas
^	obj1 ^ op2	Exponenciación

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

OPERADOR	USO	DESCRIPCIÓN
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos, ++ que incrementa en uno su operando, y — que decrementa en uno el valor de su operando.

OPERADOR	USO	DESCRIPCIÓN
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
—	op —	Decrementa op en 1; evalúa el valor antes de decrementar
—	— op	Decrementa op en 1; evalúa el valor después de decrementar

B) Operadores relacionales y condicionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, <> devuelve true si los dos operandos son distintos.

OPERADOR	USO	DEVUELVE TRUE SI
>	obj1 > op2	obj1 es mayor que op2
>=	obj1 >= op2	obj1 es mayor o igual que op2
<	obj1 < op2	obj1 es menor que op2
<=	obj1 <= op2	obj1 es menor o igual que op2
==	obj1 == op2	obj1 y op2 son iguales
<>	obj1 <> op2	obj1 y op2 son distintos

Aquí tiene tres operadores condicionales:

OPERADOR	USO	DEVUELVE TRUE SI
And	obj1 And op2	obj1 y op2 son verdaderos
Or	obj1 Or op2	al menos uno de los dos es verdadero
Not	Not op	op es falso

C) Operadores de asignación

Puede utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, VB proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo.

+=	-=	*=	/=	\=

5. Sentencias o expresiones

Las expresiones realizan el trabajo de un programa VB. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa VB. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor. Una expresión es, por tanto, una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

Si deseamos que en una misma línea aparezcan dos o más expresiones, podemos hacerlo separándolas por el carácter ":". Ejemplo: x=7 : y=x+5

5.1. Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa. Existen dos bifurcaciones diferentes: if y select case.

A) If

- **Bifurcación if end if**

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true). Tiene la forma siguiente:

```
if (expresión_booleana) then
  SENTENCIAS
end if
```

- **Bifurcación if else end if**

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el else se ejecutan en el caso de no cumplirse la expresión de comparación (false).

```
if (expresión_booleana) then
SENTENCIAS 1
else
SENTENCIAS 2
end if
```

- **Bifurcación if elseif else**

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al "else".

if (expresión_booleana) then
instrucción_si_true
 	
if (expresión_booleana) then
instrucciones_si_true
else
instrucciones_si_false
end if
	if (expresión_booleana1) then
	SENTENCIAS 1
	elseif (expresión_booleana2) then
	SENTENCIAS 2
	elseif (expresión_booleana3) then
	SENTENCIAS 3
	else
	SENTENCIAS 4
	End if

B) Sentencia Select Case

Select case (expresión) case (valor1) instrucciones_1 case (valor2) instrucciones_2 ..... case (valorN) instrucciones_N case else instrucciones_por_defecto End Select	Select case (expression) case value1 SENTENCIAS case value2, value 22 SENTENCIAS case Is >22 SENTENCIAS case value4 SENTENCIAS case value5 SENTENCIAS case value6 SENTENCIAS [case else: sentencias] End Select

5.2. Bucles

- **Bucle while**

  Permite ejecutar un grupo de instrucciones mientras se cumpla una condición dada:

  while (expresión_booleana)
  instrucciones...
  end while
  Por ejemplo:

Dim i As Integer ' While i < 10 Console.WriteLine(i) i = i + 1 End While	Dim n As Integer = 3 i = 1 While i = 10 * n ' no se repetirá ninguna vez End While

- **Bucle for**

  La forma general del bucle for es la siguiente:

  For variable = valor_inicial To valor_final [Step incremento o decremento]
  ...
  Next

For i = 1 To 10 ... Next	For i = 1 To 100 Step 2 ... Next
For i = 10 To 1 Step -1 ... Next	For i = 100 To 1 Step -10 ... Next

For Each, este bucle repetirá o iterará por cada uno de los elementos contenidos en una colección.

La forma de usarlo es:

For Each variable In colección del tipo de la variable

....

Next

Ejemplos:

```
Dim cad As String

  For Each cad In «Somos los aprobados de TAI 2005, el que nos hayan hecho machacar VB no nos va a apartar de nuestro aprobado»

    Console.WriteLine(cad)

Next

Console.ReadLine()
```

- **Bucle do while y do until**

Es similar al bucle while pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados las sentencias, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle. Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

Do While (expresión_booleana) ... Loop	Do Until (expresión_booleana) ... Loop
Do ... Loop While (expresión_booleana)	Do ... Loop Until (expresión_booleana)

Ejemplos:

i = 0
Do Until i > 9
  Console.WriteLine(i)
  i = i + 1
Loop
	i = 0
	Do While Not (i > 9)
	  Console.WriteLine(i)
	  i = i + 1
	Loop

Este bucle se repetirá para valores de i desde 0 hasta 9 (ambos inclusive).	

Para poder abandonar un bucle, hay que usar la instrucción Exit seguida del tipo de bucle que queremos abandonar:

Exit For *Exit While* *Exit Do*

6. Arrays

Los tipos de datos de las variables usadas como array, pueden ser de cualquier tipo, dependiendo de lo que queramos guardar. Se puede crear un array de un tipo que nosotros hayamos definido o de cualquier clase que exista en el .NET Framework.

6.1. Declarar variables como arrays

Para poder indicarle al VB que nuestra intención es crear un array podemos hacerlo de dos formas distintas, para este ejemplo crearemos un array de tipo Integer:

- La clásica: *Dim x() As Integer*
- La nueva forma introducida en .NET: *Dim x As Integer()*

De cualquiera de estas dos formas estaríamos creando un array de tipo Integer llamada x.

Cuando declaramos una variable de esta forma, solo le estamos indicando al VB que nuestra intención es que la variable x sea un array de tipo Integer, pero ese array no tiene reservado ningún espacio de memoria.

6.2. Reservar memoria para un array

Para poder hacerlo tenemos que usar la instrucción ReDim: *ReDim x(5)*

Al ejecutarse este código, tendremos un array con capacidad para 6 elementos.

En .NET Framework el índice menor de un array siempre es cero y en Visual Basic, el índice superior es el indicado entre paréntesis. Por tanto el array tendrá reservada memoria para 6 valores de tipo Integer, los índices serían desde 0 hasta 5 ambos inclusive.

Además de usar ReDim, que realmente sirve para «*redimensionar*» el contenido de un array, es decir, para volver a dimensionarlo o cambiarlo por un nuevo valor. Si sabemos con antelación el tamaño que contendrá el array, podemos hacerlo de esta forma: *Dim x(5) As Integer*

Con este código estaríamos declarando la variable x como un array de 6 elementos (de 0 a 5) del tipo Integer.

Cuando indicamos la cantidad de elementos que contendrá el array no podemos usar la segunda forma de declaración que te mostré anteriormente: Dim x As Integer(5) ya que esto produciría un error sintáctico.

Cuando tenemos un array declarado y asignado, podemos acceder a los elementos de ese array mediante un índice, nos será de utilidad saber cuantos elementos tiene el array, para ello podemos usar la propiedad Length, la cual

devuelve el número total de elementos, por tanto, esos elementos estarán comprendidos entre 0 y Length - 1.

```
For i = 0 To x.Length - 1
    Console.WriteLine(x(i))
Next
```

6.3. Inicializar un array al declararla

Al igual que las variables normales se pueden declarar y al mismo tiempo asignarle un valor inicial, con los arrays también podemos hacerlo, pero de una forma diferente, ya que no es lo mismo asignar un valor que varios. Aunque hay que tener presente que si inicializamos un array al declararla, no podemos indicar el número de elementos que tendrá, ya que el número de elementos estará supeditado a los valores asignados.

Ejemplo: Dim x() As Integer = {1, 42, 15, 90, 2}

También podemos hacerlo de esta otra forma: Dim x As Integer() = {1, 42, 15, 90, 2}

Usando cualquiera de estas dos formas mostradas, el número de elementos será 5; por tanto los índices irán desde 0 hasta 4.

Otros Ejemplos:

```
Dim cad As String() = {«TAI», «de «, «ADAMS», «los», "mejores"}

Dim i As Integer
For i = 0 To cad.Length - 1
    Console.WriteLine(cad(i))
Next
```

- **Usar un bucle For Each para recorrer los elementos de un array**

El tipo de bucle For Each es muy útil para recorrer los elementos de un array sin indicar el índice.

```
Dim x() As Integer = {1, 2, 3, 4, 5}
'
Console.WriteLine(«Elementos del array x()= {0}», x.Length)
'
Dim i As Integer
For Each i In x
    Console.WriteLine(i)
Next
```

Los arrays son tipos por referencia en lugar de tipos por valor. El contenido de los arrays son tipos por referencia.

6.4. Copiar los elementos de un array en otro array

La única forma de tener copias independientes de dos arrays que contengan los mismos elementos es haciendo una copia de un array a otro. Esto lo podemos hacer mediante el método CopyTo, al cual habrá que indicarle el array de destino y el índice de inicio a partir del cual se hará la copia. Solo aclarar que el destino debe tener espacio suficiente para recibir los elementos indicados por tanto deberá estar inicializado con los índices necesarios.

Ejemplo:

```
Dim x() As Integer = {1, 2, 3, 4, 5}
Dim y(x.Length - 1) As Integer
'
x.CopyTo(y, 0)
'
x(3) = 7
'
Dim i As Integer
For i = 0 To x.Length - 1
   Console.WriteLine(«x(i) = {0}, y(i)= {1}», x(i), y(i))
Next
```

En este ejemplo, inicializamos un array, declaramos otro con el mismo número de elementos, utilizamos el método CopyTo del array con los valores, en el parámetro le decimos qué array será el que recibirá una copia de esos datos y la posición (o índice) a partir de la que se copiarán los datos, (indicando cero se copiarán todos los elementos); después cambiamos el contenido de uno de los elementos del array original y al mostrar el contenido de ambos arrays, comprobamos que cada uno es independiente del otro.

Para declarar un array multidimensional, lo podemos hacer (al igual que con las unidimensionales), de varias formas, dependiendo de que simplemente declaremos el array, que le indiquemos (o reservemos) el número de elementos que tendrá o de que le asignemos los valores al mismo tiempo que la declaramos. Veamos ejemplos:

Dim x() As Integer	Dim z(,,) As Integer	Dim u(1, 6) As Integer	Dim w() As Integer = {1, 2}
Dim y(,) As Integer	Dim t(2) As Integer	Dim v(3, 1, 5, 2) As Integer	Dim r(,) As Integer={{1,2},{4}}

6.5. Cambiar el tamaño de un array

Si deseamos cambiar el tamaño de un array manteniendo los valores que tuviese, debemos usar ReDim seguida de la palabra clave Preserve: *Dim x() As Integer = {1, 2, 3, 4, 5}*

Si queremos que en lugar de 5 elementos (de 1 a 5) tenga, por ejemplo 10 y no perder los otros valores, usaremos la siguiente instrucción: *ReDim Preserve x(10)*

A partir de ese momento, el array tendrá 11 elementos (de 0 a 10), los 5 primeros con los valores que antes tenía y los nuevos elementos tendrán un valor cero, que es el valor por defecto de los valores numéricos.

Si solo usamos ReDim a(10), también tendremos once elementos en el array, pero todos tendrán un valor cero; es decir, si no se usa Preserve, se pierden los valores contenidos en el array.

Observaciones:

- Solo podemos cambiar el número de elementos de la última dimensión del array.
- Podemos usar ReDim para cambiar el número de elementos de cada una de las dimensiones.
- Podemos usar ReDim Preserve para cambiar el número de elementos de la última dimensión sin perder los valores que previamente hubiera.
- No podemos cambiar el número de dimensiones de un array.

6.6. Eliminar un array de la memoria

Si en algún momento del programa queremos eliminar el contenido de un array, por ejemplo para que no siga ocupando memoria, ya que es posible que no siga ocupando memoria, podemos usar Erase seguida del array que queremos «limpiar», por ejemplo: *Erase x.* Esto eliminará el contenido del array x.

Si después de eliminar el contenido de un array queremos volver a usarlo, tendremos que *ReDimensionarlo* con el mismo número de dimensiones que tenía, ya que Erase solo borra el contenido, no la definición del array.

7. Entrada / Salida

Las clases son la base del lenguaje Visual Basic.NET que viene dada por la biblioteca .NET. En ella, hay un espacio de nombres que destaca por las clases de propósito general que incluye: System. El espacio de nombres System contiene clases que se aplican al lenguaje mismo. Por ejemplo, estructuras que encapsulan los tipos primitivos de datos, la clase Console que proporciona los métodos para

manipular la entrada/salida (E/S) estándar, la clase String para manipular cadenas de caracteres, una clase Math que proporciona los métodos correspondientes a las funciones matemáticas de uso más frecuente, etc.

La entrada/salida queda reforzada con la funcionalidad aportada por la clase System.IO que, a su vez, aporta clases para el manejo de ficheros.

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde el teclado, o bien enviar información a la pantalla. La comunicación entre el origen de cierta información y el destino, se realiza mediante un flujo de información (en inglés *stream*). Un flujo es un objeto que hace de intermediario entre el programa y el origen o el destino de la información.

Cuando un programa Visual Basic.NET se ejecuta, se abren automáticamente tres flujos identificados por las propiedades de la clase Console indicadas a continuación:

- Un flujo desde la entrada estándar (el teclado): **In**.
- Un flujo hacia la salida estándar (la pantalla): **Out**.
- Un flujo hacia la salida estándar de error (la pantalla): **Error**.

La propiedad In hace referencia a un objeto de la clase System.IO. TextReader y las propiedades Out y Error hacen referencia a objetos de la clase System.IO.TextWriter.

7.1. Flujos de entrada

Cuando un programa define un flujo de entrada, por ejemplo, el definido por la propiedad In de la clase Console, dicho programa es destino de ese flujo de caracteres, y eso es todo lo que se necesita saber.

Dos métodos que tienen un especial interés porque permiten a un programa leer datos de la entrada estándar son:

— Public Shared Function Read() As Integer.

— Public Shared Function ReadLine() As String.

El método Read simplemente lee caracteres individuales del flujo de entrada estándar; concretamente lee el siguiente carácter disponible. Devuelve un entero (Integer) correspondiente al código del carácter leído, o bien un valor negativo cuando en un intento de leer se alcanza el final del flujo.

7.2. Flujos de salida

La propiedad Out de la clase Console define un flujo de salida. Los dos métodos que permiten a un programa escribir en la salida estándar son:

— *Overloads Public Shared Sub Write(parametros).*

— *Overloads Public Shared Sub WriteLine(parametros).*

Por ejemplo, el siguiente código lee un carácter del origen vinculado con el flujo In (entrada estándar) y lo imprime a través del flujo de salida.

```
Imports System
Module Lee
   Sub Main()
      Dim c As Char
      Console.Write(«Introduzca un caracter: «)
      c=Convert.Tochar(System.Console.Read())
      Console.WriteLine(c)
   End Sub
End Module
```

8. Clases en VB.Net

Todo .NET Framework está basado en clases (u objetos). A diferencia de las versiones anteriores de Visual Basic, la versión .NET de este lenguaje basa su funcionamiento casi exclusivamente en las clases contenidas en .NET Framework. Un programa orientado a objetos se compone solamente de objetos, cada uno de ellos es una entidad que tiene unas propiedades, los atributos, y unas formas de operar con ellas; los métodos. Los atributos definen el estado de cada uno de los objetos de esa clase y los métodos su comportamiento. No obstante, más adelante recalcaremos estos detalles.

Todo lo que tiene el .NET Framework, en realidad son clases. Cuando definimos una clase, realmente estamos definiendo dos cosas diferentes:

- Los datos que dicha clase puede manipular o contener.
- La forma de acceder a esos datos.

Por ejemplo, si tenemos una clase de tipo Empleado, por un lado tendremos los datos de dicho Empleado y por otro la forma de acceder o modificar esos datos. En el primer caso, los datos del Empleado, como por ejemplo el nombre, domicilio etc., estarán representados por una serie de campos o propiedades, mientras que la forma de modificar o acceder a esa información del Empleado se hará por medio de métodos.

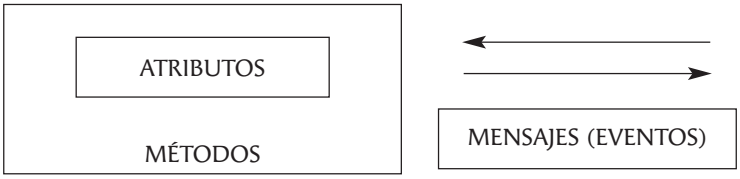
Esas propiedades o características y las acciones a realizar son las que definen a una clase. La clase es la «plantilla» a partir de la cual podemos crear un objeto en la memoria. Podemos tener varias instancias en memoria de una clase. Una instancia es un objeto (los datos) creado a partir de una clase (la plantilla o el código).

8.1. Los miembros de una clase

Las clases contienen datos, esos datos son variables. A esas variables, cuando pertenecen a una clase, se les llama campos o propiedades. Por ejemplo, el nombre de un Empleado sería una propiedad de la clase Empleado. Ese nombre lo almacenaremos en una variable de tipo String; de dicha variable podemos decir que es el «campo» de la clase que representa al nombre del Empleado.

Por otro lado, si queremos mostrar el contenido de los campos que contiene la clase Empleado, usaremos un procedimiento que nos permita mostrarlos; ese procedimiento será un método de la clase Empleado.

Por tanto, los miembros de una clase son las propiedades (los datos) y los métodos las acciones a realizar con esos datos. Sabemos que podemos almacenar esa información (en las propiedades de la clase) y que tenemos formas de acceder a ella (mediante los métodos de dicha clase).



8.2. Crear o definir una clase

Para crear una clase debemos usar la instrucción **Class** seguida del nombre que tendrá dicha clase y debe de ser cerrada usando **End Class**:

```
Class Empleado
    'propiedades de la clase
    'métodos de la clase
End Class
```

8.3. Definir los miembros de una clase

Para definir los miembros de una clase, escribiremos dentro del bloque de definición de la clase, las declaraciones y procedimientos que creamos convenientes. Veamos un ejemplo:

```
Class Empleado
    Public Nombre As String
    Sub Mostrar()
        Console.WriteLine(«El nombre del Empleado: {0}», Nombre)
    End Sub
End Class
```

En este caso, la línea *Public Nombre As String*, estaría definiendo una propiedad o «campo» público de la clase Empleado. Por otro lado, el procedimiento Mostrar sería un método de dicha clase; en esta caso, nos permitiría mostrar la información contenida en la clase Empleado.

8.4. Crear un objeto a partir de una clase

Definimos una variable capaz de contener un objeto del tipo de la clase; esto lo haremos como con cualquier variable: *Dim obj_empl As Empleado.*

Para poder crear un objeto basado en una clase, necesitamos algo más de código que nos permita «crear» ese objeto en la memoria, ya que con el código usado en la línea anterior, simplemente estaríamos declarando una variable que es capaz de contener un objeto de ese tipo, pero aún no existe ningún objeto en la memoria; para ello tendremos que usar el siguiente código: *obj_empl = New Empleado().*

Con esto le estamos diciendo al Visual Basic: crea un nuevo objeto en la memoria del tipo Empleado.

Estos dos pasos los podemos simplificar de la siguiente forma: *Dim obj_empl As New Empleado().*

A partir de este momento existirá en la memoria un objeto del tipo Empleado.

8.5. Acceder a los miembros de una clase

Para acceder a los miembros de una clase (propiedades o métodos) usaremos la variable que apunta al objeto creado a partir de esa clase, seguida de un punto y el miembro al que queremos acceder, por ejemplo, para asignar el nombre al objeto obj_empl, usaremos este código: *obj_empl.Nombre = «Lolo».*

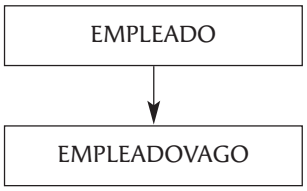
Es decir, de la misma forma que haríamos con cualquier otra variable, pero indicando el objeto al que pertenece dicha variable.

Y para acceder al método Mostrar: *obj_empl.Mostrar().*

8.6. Herencia

La herencia, como ya debíamos de saber, es una de las características más importantes de la POO, ya que permite que una clase herede los atributos y métodos de otra clase (los constructores no se heredan). Esta característica garantiza la reutilización del código.

Para poder usar la herencia en nuestras clases disponemos de la instrucción Inherits, la cual se usa seguida del nombre de la clase de la que queremos heredar. Veamos un ejemplo.



Empezaremos definiendo una clase «base» la cual será la que heredaremos en otra clase.

Ya sabemos cómo definir una clase, aunque para este ejemplo, usaremos la clase Empleado, después crearemos otra, llamada EmpleadoVago la cual heredará todas las características de la clase Empleado además de añadirle una propiedad a esa clase derivada de Empleado.

Veamos el código de estas dos clases.

```
Class Empleado
   Public Nombre As String
   Sub Mostrar()
      Console.WriteLine(«El nombre del Empleado: {0}», Nombre)
   End Sub
End Class
Class EmpleadoVago
   Inherits Empleado
   Public Sueldo As Decimal
End Class
```

Como puede comprobar, para que la clase EmpleadoVago herede la clase Empleado, se ha usado Inherits Empleado; con esta línea le estamos indicando que estamos creando una clase hija.

Haciendo esto, añadiremos a la clase EmpleadoVago la propiedad Nombre y el método Mostrar, aunque también tendremos la nueva propiedad que hemos añadido: Sueldo.

Ahora vamos a ver cómo podemos usar estas clases; para ello vamos a añadir código en el procedimiento Main del módulo del proyecto:

```
Module Module1
  Sub Main()
    Dim obj_empl As New Empleado()
    Dim obj_emplV As New EmpleadoVago()
    '
    obj_empl.Nombre = «Mary»
    obj_emplV.Nombre = «Luz»
    obj_emplV.Sueldo = 2000
    '
Console.WriteLine(«Mostrar clase Empleado»)
    obj_empl.Mostrar()
    '
Console.WriteLine(«Mostrar EmpleadoVago»)
    obj_emplV.Mostrar()
    '
    Console.WriteLine(«El Sueldo del Vago es: {0}», obj_emplV.Sueldo)
    '
    Console.ReadLine()
  End Sub
End Module
```

Lo que hemos hecho es crear un objeto basado en la clase Empleado y otro basado en EmpleadoVago.

Le asignamos el nombre a ambos objetos y a la variable obj_emplV (la del EmpleadoVago) le asignamos un valor a la propiedad Sueldo.

Fíjese que en la clase EmpleadoVago no hemos definido ninguna propiedad llamada Nombre, pero esto es lo que nos permite hacer la herencia: heredar las propiedades y métodos de la clase base. Por tanto podemos usar esa propiedad como si la hubiésemos definido en esa clase. Lo mismo ocurre con los métodos, el método Mostrar no está definido en la clase EmpleadoVago, pero sí que lo está en la clase Empleado y como resulta que EmpleadoVago hereda todos los miembros de la clase Empleado, también hereda ese método.

La salida de este programa sería la siguiente:

Mostrar clase Empleado

El nombre del Empleado: Mary

Mostrar EmpleadoVago

El nombre del Empleado: Luz

El Sueldo del Vago es: 2000

Ahora veamos cómo podríamos hacer uso del polimorfismo en una de las formas que nos permite el .NET Framework.

Teniendo ese mismo código que define las dos clases, podríamos hacer lo siguiente:

```
Sub Main()
  Dim obj_empl As Empleado
  Dim obj_emplV As New EmpleadoVago()
  '
  obj_emplV.Nombre = «Mary»
  obj_emplV.Sueldo = 2000
  obj_empl = obj_emplV
  '
  Console.WriteLine(«Mostrar Empleado»)
  obj_empl.Mostrar()
  '
   Console.WriteLine(«Usando Mostrar de la clase EmpleadoVago»)
  obj_emplV.Mostrar()
  '
   Console.WriteLine(«El Sueldo del Vago es: {0}», obj_emplV.Sueldo)
  '
  Console.ReadLine()
End Sub
```

En este caso, la variable obj_empl simplemente se ha declarado como del tipo Empleado, pero no se ha creado un nuevo objeto, simplemente hemos asignado a esa variable el contenido de la variable obj_emplV.

Con esto lo que hacemos es asignar a esa variable el contenido de la clase Empleado-Vago, pero como comprenderá, la clase Empleado «no entiende» nada de las nuevas propiedades implementadas en la clase derivada; por tanto, solo se podrá acceder a la parte que es común a esas dos clases: la parte heredada de la clase Empleado.

Realmente, las dos variables apuntan a un mismo objeto; por eso, al usar el método Mostrar se muestra lo mismo. Además de que si hacemos cualquier cambio a la propiedad Nombre, al existir solo un objeto en la memoria, ese cambio afectará a ambas variables.

Para comprobarlo, añada este código antes de la línea Console.ReadLine():

```
Console.WriteLine()
obj_empl.Nombre = «Isma»
Console.WriteLine(«Después de asignar un nuevo nombre a obj_empl.Nombre»)
obj_empl.Mostrar()
obj_emplV.Mostrar()
```

La salida de este nuevo código sería la siguiente:

```
Usando Mostrar de la clase Empleado
El nombre del Empleado: Mary
Usando Mostrar de la clase EmpleadoVago
El nombre del Empleado: Mary
La Sueldo del Vago es: 2000

Después de asignar un nuevo nombre a obj_empl.Nombre

El nombre del Empleado: Isma
```

Como puede comprobar, al cambiar en una de las variables el contenido de la propiedad Nombre, ese cambio afecta a las dos variables; solo existe un objeto en la memoria y las dos variables acceden al mismo objeto. A este tipo de variables se las llama **variables por referencia**, ya que hacen referencia o apuntan a un objeto que está en la memoria. A las variables que antes hemos estado viendo se las llama **variables por valor**, ya que cada una de esas variables tiene asociado un valor que es independiente de los demás.

Notas:

- En las clases podemos tener: campos, propiedades, métodos y eventos.
- Los métodos son procedimientos de tipo Sub o Function que realizan una acción.
- Los campos son variables usadas a nivel de la clase, es decir, son variables normales y corrientes, pero que son accesibles desde cualquier parte dentro de la clase e incluso fuera de ella.
- Las propiedades son procedimientos especiales, que al igual que los campos, representan una característica de las clases pero, a diferencia de los campos, nos permiten hacer validaciones o acciones extras que un campo nunca podrá hacer.
- Los eventos son mensajes que utilizará la clase para informar de un hecho que ha ocurrido.

8.7. Los procedimientos: métodos de las clases

Los métodos de una clase pueden ser de dos tipos: Sub o Function. Los procedimientos Sub son como las instrucciones o palabras clave de Visual Basic: realizan una tarea. Los procedimientos Function, además de realizar una tarea, devuelven un valor, el cual suele ser el resultado de la tarea que realizan. Debido a que las funciones devuelven un valor, esos valores se pueden usar para asignarlos a una variable además de poder usarlos en cualquier expresión.

```
Module Module1
  Sub Main()
    MostrarS()
    Dim cad As String = MostrarF()
    Console.WriteLine(cad)
    '
    Console.ReadLine()
  End Sub
  '
  Sub MostrarS()
     Console.WriteLine(«Este es el procedimiento
MostrarS»)
  End Sub
  '
  Function MostrarF() As String
    Return «Esta es la función MostrarF»
  End Function
End Module
```

La salida producida por este código será la siguiente:

Este es el procedimiento MostrarS
Esta es la función MostrarF

En este módulo tenemos tres procedimientos, dos de tipo Sub y uno de tipo Function, el Sub Main es un procedimiento de tipo Sub y como ya hemos comprobado ejecuta el código que esté entre la definición del procedimiento, el cual empieza con la declaración del procedimiento, que siempre se hace de la misma forma, es decir: usando Sub seguido del nombre del procedimiento y termina con End Sub.

Por otro lado, los procedimientos de tipo Function empiezan con la instrucción Function seguido del nombre de la función y el tipo de dato que devolverá la función, ya que, debido a que las funciones siempre devuelven un valor, lo lógico es que podamos indicar el tipo que devolverá. El final de la función viene indicado por End Function.

Pero como se ha comentado, las funciones devuelven un valor, el valor que una función devuelve se indica con la instrucción Return seguido del valor a devolver. En este ejemplo, el valor devuelto por la función MostrarF es el texto que está entrecomillado.

En el procedimiento Main utilizamos el procedimiento Sub usando simplemente el nombre del mismo: MostrarS. Ese procedimiento se usa en una línea independiente; cuando la ejecución del código llegue a esa línea, se procesará el contenido del mismo, el cual simplemente muestra un mensaje en la consola.

Por otro lado, el resultado devuelto por la función MostrarF se asigna a la variable **cad**. Cuando Visual Basic se encuentra con este tipo de asignación, procesa el código de la función y asigna el valor devuelto; por tanto, la variable "s" contendrá la cadena «Esta es la función MostrarF» y tal como podemos comprobar por la salida producida al ejecutar este proyecto, eso será lo que se muestre en la consola.

Cuando los procedimientos de tipo Sub o las funciones (Function) pertenecen a una clase se dicen que son métodos de esa clase. Los métodos siempre ejecutan una acción, y en el caso de las funciones, esa acción suele reportar algún valor, el cual se podrá usar para asignarlo a una variable o para usarlo en una expresión, es decir, el valor devuelto por una función se puede usar en cualquier contexto en el que se podría usar una variable o una constante. Por otro lado, los procedimientos de tipo Sub solo ejecutan la acción y nada más.

Cuando los procedimientos se convierten en métodos (porque están declarados en una clase), estos suelen representar lo que la clase (o módulo o estructura) es capaz de hacer. Es decir, siempre representarán una acción de dicha clase.

8.8. Parámetros

Cuando queramos que un procedimiento realice una tarea, es posible que necesitemos indicarle alguna información adicional. Esa información se

suele indicar mediante parámetros o argumentos. Los argumentos pasados a los procedimientos se indican a continuación del nombre del procedimiento y deben estar incluidos dentro de los paréntesis que siempre hay que usar con los procedimientos.

Por ejemplo, el método WriteLine de la clase Console permite que se indiquen mediante parámetros (o argumentos) los datos a mostrar en la consola.

Para indicar que un procedimiento acepta argumentos, estos se indicarán de la siguiente forma:

Tipo_procedimiento Nombre_procedimiento (parámetros)

Supongamos que tenemos un procedimiento llamado Saludar, al cual hay que pasarle un parámetro de tipo cadena. Dicho procedimiento usará ese parámetro como parte de un mensaje que tendrá que mostrar por la consola. Sería algo como esto:

```
Sub Saludar(ByVal nombre As String)
    Console.WriteLine(«TAI « & nombre)
End Sub
```

En este ejemplo, nombre sería el parámetro o argumento del método Saludar.

Para usar este procedimiento lo podríamos hacer de esta forma: *Saludar(«Lolo»)*.

Si necesitamos que el procedimiento reciba más de un parámetro, se podrán indicar separándolos unos de otros con una coma. Veamos el método anterior en el que se indica, además del nombre, el tipo de saludo a realizar:

```
Sub Saludar(ByVal tipoSaludo As String, ByVal nombre As String)
    Console.WriteLine(tipoSaludo & « « & nombre)
End Sub
```

Este procedimiento con dos parámetros lo usaríamos de la siguiente forma: *Saludar(«Hola», «Lolo»)*.

8.9. Parámetros por valor y parámetros por referencia

Lo que se ha pasado al procedimiento es una copia del contenido de la variable elNombre, con lo cual, cualquier cambio que se realice en la variable nombre solo afectará a la copia, no al original. Porque se ha pasado por valor (ByVal). Pero si queremos que el procedimiento pueda modificar el valor recibido como parámetro, tendremos que indicarle al Visual Basic .NET de que lo pase por referencia; para ello habrá que usar la instrucción ByRef en lugar de ByVal.

La explicación es que al pasar la variable por referencia (ByRef), el VB lo que ha hecho es asignar a la variable nombre del procedimiento la misma dirección de memoria que tiene la variable elNombre, de forma que cualquier cambio realizado en nombre afectará a elNombre.

En Visual Basic .NET, de forma predeterminada, los parámetros serán ByVal (por valor), a diferencia de lo que ocurría con las versiones anteriores de Visual Basic que eran por referencia (ByRef). Es decir, si se declara un parámetro sin indicar si es ByVal o ByRef, el VB.NET lo interpretará como si fuera ByVal.

A modo de resumen, diremos que las variables indicadas con ByVal se pasan por valor, es decir, se hace una copia del contenido de la variable o constante y es esa copia la que se pasa al procedimiento.

Por otro lado, los parámetros indicados con ByRef se pasan por referencia, es decir, se pasa al procedimiento una referencia a la posición de memoria en la que está el contenido de la variable en cuestión; por tanto, cualquier cambio efectuado a la variable dentro del procedimiento afectará a la variable indicada al llamar al procedimiento.

Todo esto es aplicable tanto a los procedimientos de tipo Sub como a los de tipo Function. En el caso de las funciones, el utilizar parámetros ByRef nos permiten devolver más de un valor: el que devuelve la función más los que se puedan devolver en los parámetros declarados con ByRef. En un procedimiento se pueden usar indistintamente parámetros por valor como por referencia, es decir, podemos tener tanto parámetros declarados con ByVal como con ByRef, y, por supuesto, solo los indicados con ByRef podrán cambiar el contenido de las variables indicadas al llamar al procedimiento.

8.10. Parámetros opcionales

Para poder indicarle al Visual Basic .NET que un parámetro es opcional debemos indicarlo usando la instrucción Optional antes de la declaración del parámetro en cuestión. Además tenemos que indicar el valor que tendrá por defecto, es decir, si no se indica ese parámetro, este debe tener un valor predeterminado. Solo podemos especificar parámetros opcionales después de todos los parámetros obligatorios.

Dicho esto, veamos cómo declarar el procedimiento Func para indicar que el segundo parámetro es opcional y que el valor predeterminado (si no se indica) es cinco:

Sub Func(ByVal uno As Integer, Optional ByVal dos As Integer = 5)

Con esta declaración podemos usar este procedimiento de estas dos formas: Func(10, 20)

En este caso se indicará un 10 para el parámetro uno y 20 para el parámetro dos. Func(10)

Si no indicamos el segundo parámetro, el valor que se usará dentro del procedimiento será el valor indicado en la declaración, es decir: 5.

8.11. Sobrecarga de procedimientos

La sobrecarga consiste en crear más de un procedimiento, constructor de instancias o propiedad en una clase con el mismo nombre y distintos tipos de

argumento. La sobrecarga es especialmente útil cuando un modelo de objeto exige el uso de nombres idénticos para procedimientos que operan en diferentes tipos de datos.

Es decir, que si necesitamos un procedimiento que utilice distinto número de parámetros o parámetros de distintos tipos, podemos usar la sobrecarga de procedimientos. Sabiendo esto, podríamos hacer lo mismo que con el procedimiento Func mostrado anteriormente con estas declaraciones:

Func()

Func(ByVal i As Integer)

Func(ByVal i As Integer, ByVal j As Integer)

Cuando se usan procedimientos sobrecargados, es el propio compilador de Visual Basic .NET el que decide cuál es el procedimiento que mejor se adecúa a los parámetros que se han indicado al llamar a ese procedimiento. Otra de las ventajas de la sobrecarga de procedimientos, es que además de poder indicar un número diferente de parámetros, podemos indicar parámetros de distintos tipos.

Esto es útil si queremos tener procedimientos que, por ejemplo, reciban parámetros de tipo Integer o que reciba parámetros de tipo Double.

Incluso podemos hacer que una función devuelva valores de tipos diferentes, aunque en este caso el número o tipo de los parámetros debe ser diferente, ya que *no se pueden sobrecargar procedimientos si solo se diferencian en el tipo de datos devuelto.*

Tampoco se pueden sobrecargar Propiedades con métodos (Sub o Function), es decir, solo podemos sobrecargar propiedades con otras propiedades, o procedimientos (Sub o Function) con otros procedimientos (Sub o Function).

8.12. Sobrecargar el constructor de las clases

Un constructor es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto. El constructor de una clase es un procedimiento de tipo Sub llamado New, dicho procedimiento se ejecuta cada vez que creamos un nuevo objeto basado en una clase. Si al declarar una clase no escribimos el «constructor», será el compilador de Visual Basic .NET el que se encargará de escribir uno genérico.

Esto es útil si queremos que al crear un objeto (o instancia) de una clase podamos hacerlo de varias formas, por ejemplo, sin indicar ningún parámetro o bien indicando algo que nuestra clase necesite a la hora de crear una nueva instancia de dicha clase.

Por ejemplo, si tenemos una clase llamada Empleado, puede sernos útil crear nuevos objetos indicando el nombre del Empleado que contendrá dicha clase. Veámoslo con un ejemplo:

```
Class Empleado
   Public Nombre As String
   Public email As String
   '
   Sub New()
     '
   End Sub

   Sub New(ByVal elNombre As String)
     Nombre = elNombre
   End Sub
End Class
```

Esta clase nos permite crear nuevos objetos del tipo Empleado de dos formas.

Por ejemplo si tenemos una variable llamada obj_empl, declarada de esta forma: *Dim obj_empl As Empleado* podemos crear nuevas instancias sin indicar ningún parámetro: *obj_empl = New Empleado()* o indicando un parámetro, el cual se asignará a la propiedad Nombre de la clase:

```
obj_empl = New Empleado(«Lolo»)
```

Igual que existe un constructor, existe también un destructor denominado Finalize que es invocado automáticamente por el recolector de basura siempre que un objeto es eliminado, cosa que ocurre cuando ese objeto queda sin referencia.

8.13. Los campos y las propiedades

Los campos son variables usadas a nivel de una clase. Los campos representan los datos de la clase. Los campos y propiedades representan los datos manipulados por la clase, mientras que los métodos manipulan (o permiten manipular) esos datos. Podemos declarar cualquier miembro de una clase de dos formas, según el nivel de visibilidad o ámbito que queramos que tenga.

- Si lo declaramos con el modificador de acceso Private, ese miembro solo será accesible desde «dentro» de la clase, es decir, en cualquier sitio de la clase podremos usar ese miembro, pero no será accesible desde «fuera» de la clase, por ejemplo, en una nueva instancia creada.
- Si declaramos un miembro de la clase como Public, ese miembro será accesible tanto desde dentro de la clase como desde fuera de la misma. Un miembro público de una clase siempre será accesible.

Cuando declaramos un campo con el modificador Public, estamos haciendo que ese campo (o variable) sea accesible desde cualquier sitio; por otro lado, si lo declaramos como Private, solo estará accesible en la propia clase.

Ejemplo:

En el siguiente código vamos a declarar una clase que tendrá tres miembros públicos y uno privado.

De estos tres miembros públicos, dos de ellos serán campos y el tercero será un método que nos permitirá mostrar por la consola el contenido de esos campos. El campo privado simplemente lo usaremos dentro del método, en otro ejemplo le daremos una utilidad más práctica, ya que en este ejemplo no sería necesario el uso de ese campo privado, pero al menos nos servirá para saber que «realmente» es privado y no accesible desde fuera de la clase.

```
Public Class Popo
  ' campo privado
  Private cad As String
  ' campos públicos
  Public Nombre As String
  Public Apel As String
  '
  ' método público
  Public Sub Mostrar()
    cad = Nombre & « « & Apel
    Console.WriteLine(cad)
  End Sub
End Class

Module Module1
  Sub Main()
    ' creamos una nueva instancia de la clase
    Dim p As New Popo()
    '
    ' asigmos los valores a los campos públicos
    p.Nombre = «Lolo»
    p.Apel = «Lolito»
    ' usamos el método para mostrar la información en la consola
    p.Mostrar()
    '
    ' esto dará error
    'Console.WriteLine(p.cad)
    '
    Console.WriteLine(«Pulsa Intro»)
    Console.ReadLine()
  End Sub
End Module
```

En la clase Popo (que está declarada como Public) tenemos declarado Nombre y Apel con el modificador Public; por tanto podemos acceder a estos dos campos desde una nueva instancia de la clase, así como desde cualquier sitio de la clase. Lo mismo es aplicable al método Mostrar, ya que al ser público se puede usar desde la variable declarada en el procedimiento Main.

Por otro lado, el campo cad está declarado como Private; por tanto, solo será accesible desde la propia clase y no desde fuera de ella. Es decir, no podremos usar ese campo desde la instancia creada en Main por la sencilla razón de que es «privada» y, por tanto, no visible ni accesible desde fuera de la propia clase.

No obstante, las propiedades (Property) son otra cosa diferente; al menos así deberíamos planteárnoslo y, tanto en Visual Basic .NET como en las versiones anteriores, además de declarar una propiedad usando la declaración de un campo público, también podemos usar la instrucción Property.

8.14. Property

La forma de usar Property es muy parecido a como se declara una función, pero con un tratamiento especial, ya que dentro de esa declaración hay que especificar por un lado lo que se debe hacer cuando se quiera recuperar el valor de la propiedad, y por otro lo que hay que hacer cuando se quiere asignar un nuevo valor.

Cuando queremos recuperar el valor de una propiedad, por ejemplo para usarlo en la parte derecha de una asignación o para usarlo en una expresión, tal es el caso de que queramos hacer algo como esto:

Dim s As String = p.Nombre o Console.WriteLine(p.Nombre)

En estos dos casos, lo que queremos es recuperar el contenido de la propiedad.

Pero si lo que queremos es asignar un nuevo valor, esa propiedad normalmente estará a la izquierda de una asignación, como sería el caso de hacer esto: p.Nombre = «Lolo». En este caso estaríamos asignando un nuevo valor a la propiedad Nombre.

Si queremos que Nombre sea realmente una propiedad (un procedimiento del tipo Property) para que podamos hacer ciertas comprobaciones tanto al asignar un nuevo valor como al recuperar el que ya tiene asignado, tendremos que crear un procedimiento:

```
Public Property Nombre() As String
  ' la parte Get es la que devuelve el valor de la propiedad
  Get
    Return elNombre
  End Get
  ' la parte Set es la que se usa al asignar el nuevo valor
  Set(ByVal Value As String)
    If Value <> «» Then
      elNombre = Value
    End If
  End Set
End Property
```

Es decir, declaramos un procedimiento del tipo Property, el cual tiene dos bloques internos:

- El primero es el bloque Get, que será el código que se utilice cuando queramos recuperar el valor de la propiedad, por ejemplo para usarlo en la parte derecha de una asignación o en una expresión.
- El segundo es el bloque Set, que será el código que se utilice cuando queramos asignar un nuevo valor a la propiedad. Tal sería el caso de que esa propiedad estuviera en la parte izquierda de una asignación.

Como puedes comprobar, el bloque Set recibe un parámetro llamado Value que es del mismo tipo que la propiedad, en este caso de tipo String. Value representa el valor que queremos asignar a la propiedad y representará lo que esté a la derecha del signo igual de la asignación.

Por ejemplo, si tenemos esto: p.Nombre = «Lolo», «Lolo» será lo que Value contenga.

Fíjese que al declarar la propiedad no se indica ningún parámetro; esto lo veremos en otra ocasión, pero lo que ahora nos interesa saber es que lo que se asigna a la propiedad está indicado por el parámetro Value del bloque Set.

Fíjese también que cuando creamos un procedimiento Property siempre será necesario tener un campo (o variable) privado que sea el que contenga el valor de la propiedad. Ese campo privado lo usaremos para devolver en el bloque Get el valor de nuestra propiedad y es el que usaremos en el bloque Set para conservar el nuevo valor asignado.

El tipo de datos del campo privado debe ser del mismo tipo que el de la propiedad. La ventaja de usar propiedades declaradas como Property en lugar de usar variables (o campos) públicos es que podemos hacer comprobaciones u otras cosas dentro de cada bloque Get o Set, tal como hemos hecho en el ejemplo de la propiedad Nombre para que no se asigne una cadena vacía al Nombre. Si nuestra intención es que dentro de una propiedad se ejecute un código que pueda consumir mucho tiempo o recursos, deberíamos plantearnos crear un método, ya que las propiedades deberían asignar o devolver los valores de forma rápida.

Debido a que en Visual Basic .NET los campos públicos son tratados como propiedades, no habría demasiada diferencia en crear una propiedad declarando una variable pública o usando un procedimiento Property, pero deberíamos acostumbrarnos a crear procedimientos del tipo Property si nuestra intención es crear una propiedad, además de que el uso de procedimientos Property nos da más juego que simplemente declarando una variable pública.

8.15. Propiedades de solo lectura

Una de las ventajas de usar un procedimiento Property es que podemos crear propiedades de solo lectura, es decir, propiedades a las que no se pueden asignar valores nuevos, simplemente podemos acceder al valor que contiene.

Para poder conseguir que una propiedad sea de solo lectura, tendremos que indicárselo al Visual Basic .NET de la siguiente forma:

```
Private valorFijo As Integer = 10
'
Public ReadOnly Property Valor() As Integer
  Get
    Return valorFijo
  End Get
End Property
```

Es decir, usamos la palabra clave (o modificador) ReadOnly al declarar la propiedad y tan solo especificamos el bloque Get. Si declaramos un procedimiento ReadOnly Property no podemos indicar el bloque Set, eso dará error.

8.16. Propiedades de solo escritura

De la misma forma que podemos definir una propiedad de solo lectura, también podemos crear una propiedad de solo escritura, es decir, una propiedad que solo aceptará que se asignen nuevos valores, pero que no permitan obtener el valor que tienen.

Veamos cómo tendríamos que declarar una propiedad de solo escritura:

```
Private valorEscritura As Boolean
'
Public WriteOnly Property Escribir() As Boolean
  Set(ByVal Value As Boolean)
    valorEscritura = Value
  End Set
End Property
```

Es decir, usamos el modificador WriteOnly al declarar la propiedad y solo debemos especificar el bloque Set. Si declaramos un procedimiento WriteOnly Property no podemos indicar el bloque Get, ya que eso dará error. Cuando declaramos una propiedad de solo lectura no podemos declarar otra propiedad con el mismo nombre que solo sea de escritura. Si nuestra intención es crear una propiedad de lectura/escritura, simplemente declaramos la propiedad sin indicar ni ReadOnly ni WriteOnly.

8.17. Campos de solo lectura

Lo mismo que existen propiedades de solo lectura, podemos crear campos de los que solo podamos leer el valor que contiene y no asignar ninguno nuevo. Los campos de solo lectura, a diferencia de las constantes, se pueden cambiar de valor, pero solo en la definición, lo cual no se diferenciaría de la forma de

declarar una constante, o dentro del constructor de la clase. Esto último es algo que no se puede hacer con una constante, ya que las constantes siempre tienen el mismo valor, el cual se asigna al declararla.

En el siguiente código vamos a declarar una constante y también un campo (o variable) de solo lectura:

Public Const PI As Double = 3.14159
Public ReadOnly Len As Integer = 50

En este código tenemos declarada una constante llamada PI que tiene un valor fijo. Las constantes siempre deben declararse con el valor que contendrán. Por otro lado, tenemos un campo de solo lectura llamado Len, que es del tipo Integer y tiene un valor de 50.

Cuando declaramos una constante pública, esta estará accesible en las nuevas instancias de la clase además de ser accesible «globalmente», es decir, no tendremos que crear una nueva instancia de la clase para poder acceder a la constante. Por tanto podríamos decir que las constantes declaradas en una clase son «variables» compartidas por todas las instancias de la clase. Es como si declarásemos la constante usando Shared o como si estuviese declarada en una clase de tipo Module.

La diferencia entre una clase de tipo Module y una de tipo Class es que en la primera, todos los miembros están compartidos (Shared), mientras que en la segunda, salvo que se indique explícitamente, cada miembro pertenecerá a la instancia de la clase, es decir, de cada objeto creado con New.

Suponte que cambiamos la declaración de Len de la siguiente forma:

Public Shared ReadOnly Len As Integer = 50

En este caso, no habría diferencia con una constante.

Pero, esta no sería la forma habitual de declarar un campo de solo lectura. Lo habitual es declararlo sin un valor inicial, aunque haciéndolo así nos aseguramos que tenga un valor predeterminado, en caso de que no se asigne ninguno nuevo.

La forma de asignar el valor que tendrá un campo de solo lectura, sería asignándolo en el constructor de la clase. Por tanto, podríamos tener un constructor (Sub New) que reciba como parámetro el valor que tendrá ese campo de solo lectura. En el siguiente código vamos a declarar una clase que tendrá un campo de solo lectura, el cual se asigna al crear una nueva instancia de la clase:

```
Public Class Kiki
  Public ReadOnly Len As Integer = 50
  '
  Public Sub New()
    '
  End Sub
  Public Sub New(ByVal len_nueva As Integer)
    Len = len_nueva
  End Sub
  '
  ' Este será el punto de entrada del ejecutable
  Public Shared Sub Main()
    '
    ' si creamos la clase sin indicar la nueva
longitud...
    Dim p As New Kiki()
    ' el valor será el predeterminado: 50
    Console.WriteLine(«p.Len = {0}», p.Len)
    '
    ' si creamos la clase sin indicar la nueva longitud...
    Dim obj1 As New Kiki(25)
    ' el valor será el indicado al crear la instancia
    Console.WriteLine(«obj1.Len = {0}», obj1.Len)
    '
    Console.WriteLine()
    Console.WriteLine(«Pulsa Intro»)
    Console.ReadLine()
  End Sub
End Class
```

Esta clase tiene definidos dos constructores: uno sin parámetros y otro que recibe un valor de tipo Integer; ese valor será el que se use para el campo de solo lectura.

En el Sub Main, el cual está declarado como Shared para que se pueda usar como punto de entrada del ejecutable, declaramos dos objetos del tipo de la clase: el primero se instancia usando New sin ningún parámetro, mientras que el segundo se crea la nueva instancia indicando un valor en el constructor; ese valor será el que se utilice para darle valor al campo de solo lectura, cosa que se demuestra en la salida del programa:

p.Len = 50

obj1.Len = 25

En los comentarios está aclarado por qué el objeto p toma el valor 50 y porqué usando obj1 el valor es 25.

Una vez que hemos asignado el valor al campo de solo lectura, ya no podemos modificar dicho valor, salvo que esa modificación se haga en el constructor. Por tanto, solo podemos asignar un nuevo valor a un campo de solo lectura en el constructor de la clase.

8.18. El ámbito de los miembros de una clase

Hasta ahora hemos estado usando dos instrucciones (o modificadores) que permiten indicar el ámbito de un miembro de una clase (o módulo): uno de ellos es Public y el otro es Private. Al usar el modificador Public, permitimos que el miembro al que se le ha aplicado ese modificador sea visible desde cualquier sitio y por tanto estará accesible para que podamos usarlo desde la propia clase o desde fuera de ella; por otro lado, cuando usamos Private, estamos indicando que ese miembro es «privado» a la clase en la que se ha declarado y, por tanto, solo podremos usarlo desde la propia clase.

Ejemplo:

```
Public Class Dede                    Private elNombre As String
  Public Apel As String
  '
  Public Property Nombre() As String
    Get
      Return elNombre
    End Get
    Set(ByVal Value As String)
      If Value <> «» Then
        elNombre = Value
      End If
    End Set
  End Property
  Public Sub Mostrar()
    Console.WriteLine(elNombre & « « & Apel)
  End Sub
End Class
```

En esta clase hemos definido dos campos: elNombre y Apel. El primero está declarado como Private; por tanto, solo será visible *dentro* de la clase, es decir, no podemos acceder a ese campo desde *fuera* de la clase (ahora lo comprobaremos). El segundo está declarado como Public; por tanto será accesible (o visible) tanto desde dentro de la clase como desde fuera de ella.

Tendremos que crear una nueva instancia, pero a los miembros a los que podemos acceder, solo serán los que estén declarados como Public. Además de esos dos campos, también tenemos una propiedad y un método, ambos declarados como Public.

El método Mostrar, que es un procedimiento de tipo Sub, puede acceder tanto a la variable declarada como Private como a la que hemos declarado como Public, ya que ambas variables (en este caso también campos de la clase) tienen la «cobertura» suficiente para que sean vistas (o accesibles) desde cualquier sitio «*dentro*» de la propia clase.

La propiedad Nombre, tiene dos bloques de código: uno será la parte Get (la que devuelve el valor de la propiedad) y la otra es el bloque Set (el que asigna el valor a la variable privada). Desde los dos bloques podemos acceder a la variable privada elNombre, ya que el ámbito de dicha variable es: toda la clase.

```
Module Module1
  Sub Main()
    Dim n As New Dede()
    '
    n.Nombre = «Lolo»
    n.Apel = «Som»
    n.Mostrar()
  End Sub
End Module
```

En este módulo tenemos un procedimiento en el cual declaramos una variable del tipo Dede. Creamos una nueva instancia en la memoria y usamos los miembros públicos. Las variables en un procedimiento, (Sub, Function o Property), tenemos que declararlas con Dim. Esto significa que las variables son privadas (o locales) al procedimiento; por tanto, solo se podrán usar dentro del procedimiento y desde fuera de él no sabrán de la existencia de esas variables. Pero aún hay más: si hubiese una variable de «nivel superior» que se llamase igual que una variable «local» al procedimiento, esta última ocultaría a la que está declarada en un nivel superior.

Por ejemplo, en la clase Dede tenemos la variable elNombre, declarada a nivel de módulo; esta variable podrá usarse dentro del procedimiento Mostrar, pero si el procedimiento Mostrar lo sobreescribimos de esta forma:

```
Public Sub Mostrar()
  ' func declarando una variable «local»
  ' que se llama igual que otra declarada a nivel de módulo
  Dim elNombre As String
  '
  Console.WriteLine(elNombre & « « & Apel)
End Sub
```

La variable declarada dentro del procedimiento «ocultaría» a la declarada a nivel de módulo; por tanto, ahora no se mostraría el nombre que hubiésemos asignado a la clase, sino una cadena vacía, ya que no hemos asignado nada a esa variable.

Si el parámetro de un procedimiento recibe una variable que se llama como otra declarada a nivel superior, esa variable también ocultaría a la de nivel superior.

Por ejemplo, si tenemos este procedimiento:

```
Public Sub Mostrar2(ByVal elNombre As String)
  Console.WriteLine(elNombre & « « & Apel)
End Sub
```

El nombre de la variable usada como parámetro también se llama igual que la declarada a nivel de módulo; por tanto, esa variable (la del parámetro) ocultará a la otra.

Realmente son dos variables diferentes y cada una tendrá su propia dirección de memoria; por tanto el valor asignado a esas variables locales no afectará en nada a la variable declarada a nivel de módulo.

Cuando declaramos variables locales (al procedimiento), estas durarán (o existirán) mientras dure el procedimiento; es decir, cuando se sale del procedimiento, el valor que tuviera asignado, se perderá. Cuando se vuelva a entrar en el procedimiento se volverán a crear esas variables y el valor que antes tenían ya no será recordado. Habrá ocasiones en que no queramos que esos valores de las variables locales se pierdan, en Visual Basic podemos hacerlo declarando las variables usando la instrucción Static en lugar de Dim. Static le indica al compilador que esa variable debe mantener el valor entre distintas llamadas al procedimiento, para que de esa forma no se pierda el valor que tuviera. Las variables declaradas con Static siguen siendo locales al procedimiento y también «ocultarán» a variables declaradas a nivel de módulo que se llamen de igual forma. Es decir, funcionan como las declaradas con Dim, pero mantienen el valor, además de que Static solo se puede usar para declarar variables dentro de procedimientos.

En los bloques podemos declarar variables «privadas» a esos bloques, pero en esta ocasión solo pueden declararse con Dim. Las variables declaradas en un bloque solo serán visibles «dentro» de ese bloque; por tanto, no podrán ser accedidas desde fuera del bloque.

Las limitaciones que tenemos en Visual Basic .NET respecto a las variables «locales» declaradas en un bloque, es que no pueden llamarse de igual forma que una declarada en el mismo «nivel» en el que se usa el bloque, normalmente en un procedimiento, ya que no podemos usar un bloque fuera de un procedimiento.

Una variable declarada en un bloque sí se puede llamar como otra declarada a nivel de módulo; en ese caso, la variable «local» ocultará a la declarada a nivel de módulo.

Lo que debemos tener presente es que las variables declaradas en un bloque «mantienen» el valor mientras dure la vida del procedimiento en el que se encuentran. Es como si fuesen «estáticas» mientras dure el procedimiento, aunque, cuando termina el procedimiento, ese valor se pierde.

También debemos saber que esto solo es aplicable si no hemos asignado un valor a la variable al declararla, ya que si se asigna un valor al declararla, siempre usará ese valor.

```
Module Modulo3
  Sub Main()
    Dim i As Integer
    For i = 1 To 2
      Console.WriteLine(« i vale {0}», i)
      Func3()
    Next
  End Sub
```

```
Sub Func3()
    Dim i As Integer
    For i = 1 To 5
      If i > 1 Then
        Dim j As Integer
        Dim k As Integer = 2
        k += 1
        j += 1
        Console.WriteLine(«j = {0}, k = {1}», j, k)
      End If
    Next
  End Sub
End Module
```

En este ejemplo, dentro del bloque If i > 1 Then... tenemos dos variables locales al bloque. La variable "j" se ha declarado de forma «normal», mientras que la variable "k" se ha declarado usando un valor inicial. Cada vez que llamemos al procedimiento Func3, el bucle se repetirá 5 veces y la condición para entrar en el bloque se ejecutará 4 de esas 5 veces, y como podemos comprobar la variable "j" va tomando valores desde 1 hasta 4, mientras que "k" siempre muestra el valor 3 (dos que le asignamos al declarar y uno del incremento).

La salida de esta func sería la siguiente:

i vale 1	i vale 2
j = 1, k = 3	j = 1, k = 3
j = 2, k = 3	j = 2, k = 3
j = 3, k = 3	j = 3, k = 3
j = 4, k = 3	j = 4, k = 3

8.19. Las variables declaradas a nivel de módulo

En las declaraciones de las variables declaradas a nivel de módulo pueden entrar en juego otros modificadores de visibilidad; estos «modificadores», también serán aplicables a los procedimientos.

Cuando declaramos una variable a nivel de módulo, esta será visible en todo el módulo (clase) en el que se ha declarado; además, dependiendo del modificador de nivel de visibilidad que tenga, podrá ser «vista» (o estará accesible) desde otros sitios. Recuerde que las variables declaradas en los módulos se les llama «campos» y si son accesibles desde fuera de la clase, pueden llegar a confundirse con las propiedades.

8.20. Los niveles de visibilidad (accesibilidad o ámbito)

Veamos un resumen de los modificadores de visibilidad que podemos usar en los programas de Visual Basic .NET

MODIFICADOR (VB)	DESCRIPCIÓN
Dim	Declara una variable en un módulo, procedimiento o bloque. Cuando se usa para declarar una variable a nivel de módulo, se puede sustituir por Private.
Private	El elemento declarado solo es visible dentro del nivel en el que se ha declarado.
Public	El elemento es visible en cualquier parte.
Friend	El elemento es visible dentro del propio ensamblado (proyecto).
Protected	El elemento es visible solo en las clases derivadas.
Protected Friend	El elemento es visible en las clases derivadas y en el mismo ensamblado.

Hay que tener en cuenta que el ámbito que tendrán los elementos declarados (por elemento entenderemos que es una variable, un procedimiento, una clase, módulo, estructura o una enumeración), dependerán del nivel en el que se encuentran. Por ejemplo, cuando declaramos un elemento con el modificador Public dentro de un nivel que ha sido declarado como Private, el elemento declarado como público no podrá «sobrepasar» el nivel privado. Por otro lado, si se declara un elemento como Private, será visible desde cualquier otro elemento que esté en el mismo nivel o en un nivel inferior.

Nota: *En la documentación de Visual Studio .NET,* ***ámbito*** *es el nivel de visibilidad que puede tener, ya sea a nivel de bloque, procedimiento, módulo o espacio de nombres. Y la* ***accesibilidad*** *es la «visibilidad» de dicho elemento, si es público, privado, etc.*

Otra cosa a tener en cuenta es que cuando declaramos en Visual Basic .NET una clase, una estructura, una enumeración o un procedimiento sin especificar el ámbito, de forma predeterminada será Public.

Los Namespace no pueden tener modificadores de acceso, siempre serán accesibles.

Los módulos (o clases) declaradas como Module, tampoco pueden tener modificadores de acceso, y también se consideran como Public, por tanto, siempre accesibles.

Además de que cualquier elemento declarado en un Module podrá usarse sin necesidad de indicar el nombre del módulo; dónde puedan usarse dependerá del ámbito que le hayamos dado.

8.21. Diferencia entre Class y Module y miembros compartidos

Por regla general, se usarán las clases de tipo Class cuando «realmente» queramos crear una clase. Para poder acceder a los miembros de una clase, tendremos que crear una nueva instancia en la memoria.

Por otro lado, usaremos las clases de tipo Module cuando queramos tener «miembros» que puedan ser usados en cualquier momento, sin necesidad de crear una nueva instancia de la clase, de hecho no podemos crear una instancia de un tipo declarado como Module. El que podamos acceder a los miembros de un Module sin necesidad de crear una nueva instancia, es por el hecho de que esos miembros son miembros compartidos.

Bueno, realmente no solo porque son miembros compartidos, sino porque el compilador de Visual Basic .NET trata de forma especial a los «tipos de datos» declarados como Module. Los cuales se pueden acceder sin necesidad de usar el nombre del módulo en el que están incluidos.

Pero los Module no son los únicos «poseedores» de la exclusiva de los miembros compartidos, ya que podemos declarar clases, (del tipo Class), que tengan miembros compartidos.

Para que un miembro de una clase esté compartido, hay que usar el modificador Shared.

Esta instrucción le indicará al runtime de .NET que ese miembro puede ser accedido sin necesidad de crear una nueva instancia de la clase. Para simplificar, diremos que es como si el CLR hubiese creado una instancia de esa clase y cada vez que queramos acceder a un miembro compartido, usará esa instancia para acceder a ese miembro. En los miembros compartidos no se pueden usar variables u otros miembros que no estén compartidos.

8.22. Cómo declarar miembros compartidos en una clase, y cómo usarlos

Veamos un pequeño código en el que tenemos una clase con elementos compartidos y no compartidos.

```
Class Dede
 ' miembros compartidos
 Public Shared elNombreShared As String
 Public Shared Sub MostrarCompartido()
  Console.WriteLine(«Este procedimiento está declarado como compartido (Shared)»)
  Console.WriteLine(«Solo podemos acceder a miembros compartidos:
{0}», elNombreShared)
 End Sub
```

```
'
' miembros de instancia
Public elNombreInstancia As String
Public Sub MostrarInstancia()
  Console.WriteLine(«Este procedimiento es de instancia.»)
   Console.WriteLine(«Podemos acceder a miembros de instancia: {0}», elNombreInstancia)
  Console.WriteLine(«y también a miembros compartidos: {0}», elNombreShared)
 End Sub
 '
End Class
```

En esta clase, hemos declarado un «campo» y un procedimiento compartido (usando Shared) y otro campo y otro procedimiento no compartido (de instancia).

Desde el procedimiento compartido solo podemos acceder al campo compartido.

Pero desde el procedimiento de instancia podemos acceder tanto al miembro compartido como al de instancia. Recuerde que una instancia es un nuevo objeto creado en la memoria. Por tanto, cada instancia (u objeto en la memoria) tendrá su propia copia de los miembros de instancia, mientras que de los miembros compartidos solo existe una copia en memoria que será común para todos y cada uno de los objetos instanciados.

Esto lo podemos comprobar en el siguiente código:

```
Module Modulo
 Sub Main()
  ' para acceder a miembros compartidos
  ' usaremos el nombre de la clase:
  Dede.elNombreShared = «Shared»
  Dede.MostrarCompartido()
  '
  Console.WriteLine()
  ' creamos una instancia de la clase
  Dim obj As New Dede()
  '
  obj.elNombreInstancia = «Instancia»
  obj.MostrarCompartido()
  obj.elNombreShared = «Shared cambiado»
```

```
    obj.MostrarInstancia()
    '
    Console.WriteLine()
    '
    ' creamos otro objeto del mismo tipo (otra instancia)
    Dim obj2 As New Dede_5()
    '
    obj2.elNombreInstancia = «Instancia 2»
    obj2.MostrarInstancia()
    'obj2.MostrarCompartido()
    Dede.MostrarCompartido()
  End Sub
End Module
```

La salida de este código será la siguiente:

```
Este procedimiento está declarado como compartido (Shared)
Solo podemos acceder a miembros compartidos: Shared

Este procedimiento está declarado como compartido (Shared)
Solo podemos acceder a miembros compartidos: Shared
Este procedimiento es de instancia.
Podemos acceder a miembros de instancia: Instancia
y también a miembros compartidos: Shared cambiado

Este procedimiento es de instancia.
Podemos acceder a miembros de instancia: Instancia 2
y también a miembros compartidos: Shared cambiado
Este procedimiento está declarado como compartido (Shared)
Solo podemos acceder a miembros compartidos: Shared cambiado
```

Como puede comprobar, para acceder a los miembros compartidos, hemos usado el nombre de la clase.

De esta forma solo podemos acceder a los miembros compartidos.

Por otro lado, al crear una instancia de esa clase, podemos acceder tanto a los miembros compartidos como a los de instancia, en este ejemplo, se asigna un valor al campo de instancia, se llama al procedimiento compartido y des-

pués cambiamos el contenido del campo compartido y llamamos al procedimiento de instancia. En la segunda clase, al mostrar el contenido del campo compartido, mostrará el valor que el objeto (de instancia) le asignó, ya que ese campo está compartido por todas las instancias. En la última línea de código se ha usado el método compartido usando la clase, pero el resultado sería el mismo si se usara el objeto para llamar al método compartido.

8.23. Sobre el código y los datos de una clase

Cada vez que creamos (instanciamos) un objeto en la memoria, estamos reservando espacio para ese objeto. Se reserva espacio tanto para los datos como para el código de los procedimientos. El código siempre es el mismo para todos los objetos creados en la memoria, es decir, solo existe una copia en la memoria del código, no se «copia» ese código una vez por cada objeto que hemos creado. Por otro lado, los datos si que ocupan espacios de memoria diferentes, uno para cada instancia.

Cuando se usa un procedimiento de una clase y en ese procedimiento se usan valores de instancia, (los que existen de forma independiente para cada objeto creado), el runtime realmente hace una llamada al código indicándole dónde están los datos que tiene que manipular, es decir, le pasa al código la dirección de memoria en la que se encuentran los datos a manipular, de esta forma, solo existe una copia del código (igual que ocurre con los datos compartidos) y varias direcciones de memoria para los datos no compartidos, (según el número de instancias que se hayan creado).

9. Assembly (ensamblado)

Un ensamblado es el bloque constructivo primario de una aplicación de .NET Framework. Se trata de una recopilación de funcionalidad que se construye, versiona e instala como una única unidad de implementación (como uno o más archivos).

Para que nos entendamos, podríamos decir que un assembly es una librería dinámica (DLL) en la cual pueden existir distintos espacios de nombres. Aunque esto es simplificar mucho por ahora nos vale.

Un ensamblado o assembly puede estar formado por varios ficheros DLLs y EXEs, pero lo más importante es que todos los ensamblados contienen un manifiest.

Cada assembly contiene un manifiesto en el cual se indica:

- El nombre y la versión del assembly.
- Si este assembly depende de otros ensamblados, con lo cual se indica hasta la versión de dichos ensamblados.
- Los tipos expuestos por el assembly (clases, etc.).
- Permisos de seguridad para los distintos tipos contenidos en el assembly.

También se incluyen en los assemblies los datos del copyright, etc. La ventaja de los ensamblados es que realmente no necesitan de una instalación y un registro correcto en el registro del sistema de Windows, ya que es el intérprete de .NET el que se encarga de hacer las comprobaciones cuando tiene que hacerlas. Por tanto, podríamos distribuir una aplicación sin necesidad de crear un programa de instalación. Pero si la aplicación usa ensamblados compartidos, puede que sea necesario usar una instalación. Los ensamblados compartidos se pueden usar por varias aplicaciones diferentes y deben estar «debidamente» instalados en el directorio asignado por el propio .NET Framework. Ejemplo de ensamblados compartidos son los que definen las clases (tipos) usados por el propio .NET Framework.

10. Espacios de nombres (Namespace)

Un Namespace es una forma de agrupar clases, funciones, tipos de datos, etc. que están relacionadas entre sí. Por ejemplo, entre los Namespaces que podemos encontrar en el .NET Framework encontramos uno con funciones relacionadas con Visual Basic: Microsoft.VisualBasic. Microsoft y VisualBasic están separados por un punto, esto significa que Microsoft a su vez es un Namespace que contiene otros «espacios de nombres», tales como el mencionado VisualBasic, CSharp y Win32 con el cual podemos acceder a eventos o manipular el registro del sistema.

Por regla general, se deberían agrupar en un Namespace funciones o clases que estén relacionadas entre sí, de esta forma, será más fácil saber que estamos trabajando con funciones relacionadas entre sí.

Pero el que distintos espacios de nombres pertenezcan a un mismo Namespace no significa que todos estén dentro de la misma librería o assembly. Un Namespace puede estar repartido en varios assemblies o librerías. Por otro lado, un assembly puede contener varios Namespaces.

Nota. *En JAVA a los espacios de nombres, se les llama paquetes.*

El uso de los espacios de nombres nos permitirá tener nuestros ensamblados agrupados según la tarea para la que los hemos programado. De esta forma, si escribimos ensamblados que accederán al disco, los podemos agrupar para tenerlos en sitio diferente a los que, por ejemplo, accedan a una base de datos. Eso mismo es lo que se hace en las clases base de .NET Framework y por esa razón existen los espacios de nombres System.IO y System.Data, por poner solo dos.

Cada vez que creamos un proyecto con Visual Studio .NET de forma automática se crea un espacio de nombres para nuestra aplicación. Por tanto todos los tipos (clases, estructuras, enumeraciones, etc.) que definamos en dicho proyecto estarán incluidos dentro de ese espacio de nombres.

Aunque el editor de Visual Studio no nos muestra que todo esto es así, ya que para crear o definir un espacio de nombres debemos usar la instrucción Namespace y marcar el final de dicha definición usando End Namespace, por

tanto, es como si el IDE añadiera esas instrucciones por nosotros. Todos nuestros proyectos estarán dentro de un espacio de nombres. Dicho Namespace se llamará de la misma forma que nuestro proyecto, salvo si tiene espacios, en ese caso se cambiarán los espacios (o caracteres no válidos en un nombre de .NET) por un guion bajo.

El escribir nuestro código dentro de un bloque Namespace tiene por finalidad, tal como se ha comentado al principio, poder mantener una jerarquía. Podríamos comparar los espacios de nombres con los directorios de un disco, en cada directorio tendremos ficheros que de alguna forma están relacionados, de esta forma no mezclaremos los ficheros de música con los de imágenes ni con los proyectos de Visual Basic, por poner algunos ejemplos.

Pues lo mismo ocurre con las jerarquías creadas con los Namespace, de forma que podamos tener de alguna forma separados unos tipos de datos (clases, etc.) de otros.

Anidar espacios de nombres: espacios de nombres dentro de otros espacios de nombres.

Es habitual que dentro de un directorio podamos tener otros directorios, de forma que tengamos ficheros que estando relacionados con el directorio principal no queremos que se mezclen con el resto de ficheros.

Pues con los Namespace ocurre lo mismo, podemos declarar bloques Namespace dentro de otro bloque existente. Esto se consigue definiendo un bloque dentro de otro bloque.

Por ejemplo:

```
Namespace CursoLolo
  Namespace Espacio
    Module Module1
      Sub Main()
        '...
      End Sub
      Sub Func()
        '...
      End Sub
    End Module
  End Namespace
End Namespace
```

En este código, tenemos dos espacios de nombres anidados, el espacio de nombres Espacio está dentro del espacio de nombres CursoLolo.

10.1. Tipos declarados dentro de un Namespace

De igual forma que para acceder a los ficheros de un directorio debemos indicar el path en el que se encuentran, con los espacios de nombres ocurre lo mismo. Por ejemplo, si queremos usar la clase StringBuilder que se encuentra declarada en el espacio de nombres System.Text, tendremos que indicar el espacio de nombres en el que se encuentra: *Dim sb As New System.Text.StringBuilder()*

Es decir, debemos indicar el path completo de la clase, para poder acceder a ella.

En teoría, si declaramos un método dentro del módulo Module1 (mostrado en el código anterior), deberíamos llamar a dicho método usando el espacio de nombres completo, pero cuando estamos dentro de un espacio de nombres, no hace falta indicar el nombre completo, ocurre lo mismo que cuando estamos dentro de en un directorio, no tenemos que indicar el path completo para acceder a un fichero de ese mismo directorio, pero si queremos también podemos hacerlo, tal como se muestra en el siguiente código:

```
Namespace CursoLolo
  Namespace Espacio
    Module Module1
      Sub Main()
        CursoLolo.Espacio.Module1.Func()
      End Sub
      Public Sub Func()
        Console.WriteLine(«Saludos desde el Sub Func»)
      End Sub
    End Module
  End Namespace
End Namespace
```

Por tanto, para acceder a los miembros declarados dentro de un espacio de nombres, debemos usar la sintaxis: *Espacio.nombre_clase.nombremetodo*

Si dicha clase está dentro de un espacio de nombres anidado, también debemos indicar dicho espacio de nombres. Es decir, siempre debemos indicar los espacios de nombres en los que se encuentran las clases, salvo que la clase (o tipo) esté definido en el mismo espacio de nombres desde el que se va a usar. Aunque existen ciertas formas de hacer que esto no siempre tenga que ser así, ya que el código se puede convertir en algo más engorroso de leer si siempre tenemos que indicar los espacios de nombres que contienen las clases.

10.2. Importaciones

Pues además de no tener que especificar los espacios de nombres cuando la clase (o tipo) está definido en el mismo en el que queremos usarla, podemos usar lo que se llama importaciones de espacios de nombres, de esa forma podemos acortar el código que tenemos que escribir.

Las importaciones de los espacios de nombres se hacen mediante la instrucción Imports. Dicha instrucción se usará al principio del fichero de código y lo que realmente hace es indicarle al compilador que, si no encuentra una clase en el espacio de nombres actual, la busque en el espacio de nombres que hemos importado. Por ejemplo, la clase Console está definida en el espacio de nombres System, si usamos una importación de System en nuestro código (el Visual Studio lo hace de forma automática) no tendremos que indicar la ruta completa de dicha clase: System.Console, sino que podemos usarla indicando simplemente Console. La instrucción Imports se usa de la siguiente forma: Imports System.

Si queremos importar un espacio de nombres que está anidado en otro, lo haremos así:Imports System.Text. Es como si definiéramos un directorio dentro de la variable de entorno PATH, si queremos acceder a aun ejecutable que esté en cualquiera de esos directorios, no nos veremos obligados a indicarlo.

Por tanto el uso de Imports lo que hace es indicarle al compilador dónde buscar la definición de los tipos que estamos usando en nuestro código, para así ahorrarnos la escritura de los espacios de nombres que habitualmente usamos.

Para importar, por tanto, un espacio de nombre, usamos Imports, seguido del espacio de nombres, al principio de dicho fichero.

Una cosa que no hay que confundir son las importaciones de espacios de nombres y las referencias.

Las referencias son las que le indican al compilador dónde encontrar las clases (y espacios de nombres) que queremos usar en nuestro proyecto, mientras que las importaciones simplemente hacen que no tengamos que escribir los espacios de nombres importados.

Por supuesto, las importaciones (Imports) las podemos realizar no solo con los espacios de nombres definidos en .NET Framework, sino que podemos hacerlas para importar nuestros propios espacios de nombres.

Si hacemos una importación de un espacio de nombres, podemos acceder no solo a las clases (tipos) definidos en ese espacio de nombres, sino también a otros espacios de nombres anidados.

Por ejemplo si queremos acceder a la clase StringBuilder que, como hemos visto, está definida en System.Text, es decir, en el espacio de nombres Text que a su vez está definido dentro de System, en Visual Basic podríamos hacerlo así:

Dim sb As New Text.StringBuilder()

10.3. Crear alias a los espacios de nombres

Un alias es una forma abreviada de usar algo. En el caso de las importaciones de espacios de nombres, podemos crear un alias para un espacio de nombres, de forma que en lugar de escribir todo el espacio de nombres (y subespacios) simplemente usemos ese alias. Por ejemplo, imagine que quiere acceder a las clases del espacio de nombres System.IO pero no quieres realizar una importación completa, con idea de que al escribir dicho alias te muestre todas las clases y demás tipos relacionadas con ese espacio de nombres. En ese caso, podríamos crear un alias al espacio de nombres System.IO de esta forma:

```
Imports alias = System.IO
```

A partir de ese momento podemos acceder a los tipos definidos en dicho espacio de nombres usando el alias que hemos creado: alias.

10.4. Ejemplos de cómo usar los namespace

En el ejemplo que vamos a usar, vamos a definir dos espacios de nombres y en cada uno de ellos declararemos algunas una clase, con idea de que veamos cómo usarlos desde cada uno de esos espacios de nombres. El código sería el siguiente:

```
Namespace Espacio
  Module Module1
    Sub Main()
      ' declarar una clase del espacio de nombres ClaseUno
      Dim c1 As New Lolito.FuncA(«Ismael», «Rodriguez»)
      '
      ' declarar una clase del espacio de nombres ClaseDos
      Dim c2 As New ClasesDos.FuncB(«Maria», «Luz»)
      '
      Console.WriteLine(«c1 = {0}», c1)
      Console.WriteLine(«c2 = {0}», c2)
      '
      Console.ReadLine()
    End Sub
  End Module
End Namespace
'
Namespace Lolito
  Public Class ClaseA
    Private _nombre As String
```

```
    Private _Apel As String
    '
    Public Sub New()
    End Sub
    Public Sub New(ByVal nombre As String, ByVal Apel As String)
      Me.Nombre = nombre
      Me.Apel = Apel
    End Sub
    '
    Public Property Nombre() As String
      Get
        Return _nombre
      End Get
      Set(ByVal value As String)
        _nombre = value
      End Set
    End Property
    Public Property Apel() As String
      Get
        Return _Apel
      End Get
      Set(ByVal value As String)
        _Apel = value
      End Set
    End Property
    '
    Public Overrides Function ToString() As String
      Return Apel & «, « & Nombre
    End Function
  End Class
End Namespace
'
Namespace ClasesDos
  Public Class ClaseB
    Inherits Lolito.ClaseA
```

```
    '
    Public Sub New()
    End Sub
    Public Sub New(ByVal nombre As String, ByVal Apel As String)
      Me.Nombre = nombre
      Me.Apel = Apel
    End Sub
  End Class
End Namespace
```

Para poder acceder a las clases desde Main, tenemos que indicar en qué espacio de nombres está cada una de las clases; esto es así porque el módulo que contiene el método Main está declarado dentro de un espacio de nombres diferente. Lo mismo ocurre con la instrucción Inherits Lolito.ClaseA de la clase ClaseB, ya que cada una de las clases está dentro de un espacio de nombres distinto y si no indicamos expresamente ese espacio de nombres, el compilador no sabrá dónde localizar la clase ClaseA. Por supuesto, si hacemos una importación del espacio de nombres Lolito, podríamos acceder a la clase ClaseA sin necesidad de indicar el espacio de nombres que la contiene.

En un proyecto (ensamblado) podemos definir varios espacios de nombres. Un espacio de nombres puede estar definido en varios ensamblados y todas las clases (tipos) definidos se podrán acceder como si estuviesen en el mismo proyecto. Podemos importar espacios de nombres para que el compilador sepa dónde buscar las clases.

11. Excepciones

11.1. Definición de error y excepción

Los términos error y excepción se suelen utilizar indistintamente. De hecho, un error, que es un evento que sucede durante la ejecución del código, interrumpe el flujo normal del mismo y crea un objeto de excepción. Al producirse la interrupción, el programa intenta buscar un controlador de excepciones, que es un bloque de código que indica cómo reaccionar ante el problema y que ayudará a reanudar el flujo. Es decir, **el error es el evento y la excepción es el objeto que crea dicho evento**.

Con la expresión «iniciar una excepción» los programadores indican que el método en cuestión encontró un error y reaccionó con la creación de un objeto de excepción que contiene información acerca del error y el momento y lugar donde sucedió. Entre los factores que causan el error y las posteriores excepciones se incluyen errores de usuario, de recursos y de lógicas de programación. Dichos errores se relacionan con el modo en que el código realiza una tarea determinada y no con el propósito de esta.

11.2. Control estructurado frente al no estructurado y cuándo utilizar cada uno de ellos

El control de excepciones estructurado consiste simplemente en utilizar una estructura de control que contiene excepciones, bloques de código aislados y filtros para crear un mecanismo de control. Con ello se permite que el código realice una distinción entre diferentes tipos de errores y reaccione según las circunstancias. En el control de excepciones no estructurado una instrucción On Error al principio del código controla todas las excepciones.

No se puede combinar el control de excepciones de ambos tipos en la misma función. Si se utiliza una instrucción On Error, no se podrá emplear Try...Catch en la misma función.

Independientemente del modo que se elija para controlar las excepciones en el código, se debe considerar la situación con cierta perspectiva y analizar qué suposiciones realiza el código. Por ejemplo, si la aplicación solicita al usuario que introduzca un número de teléfono, se tendrán en cuenta las siguientes suposiciones:

- El usuario introducirá un número en lugar de caracteres.
- El número presentará un formato determinado.
- El usuario no introducirá una cadena nula.
- El usuario dispone de un solo número de teléfono.

En la introducción de datos por parte del usuario se pueden infringir todas las suposiciones o cualquiera de ellas. Para que el código resulte eficaz se requiere un control de excepciones apropiado que permita que la aplicación se recupere de forma adecuada frente a infracciones de este tipo.

A menos que se pueda garantizar que un método no iniciará una excepción bajo ninguna circunstancia, se debe considerar disponer de un control de excepciones informativo. Dicho control debe resultar útil. Además de indicar que existe algún problema, los mensajes que procedan del mismo deben informar del motivo y lugar donde se ha producido el error. Con los mensajes que no incluyen información alguna además de «ha ocurrido un error» solo se consigue la frustración del usuario.

11.3. Control de excepciones estructurado

Con este tipo de control cuando se produce la excepción, el código del control de excepciones se adapta a las circunstancias causantes de dicha excepción. En aplicaciones de gran tamaño este método resulta mucho más rápido que el control de excepciones no estructurado; asimismo, permite una respuesta más flexible a los errores y una mayor fiabilidad de la aplicación.

La estructura de control Try...Catch...Finally resulta fundamental en el control de excepciones estructurado. Comfunc, una parte del código, filtra las excepciones creadas mediante la ejecución dicho código y reacciona de forma diferente en función del tipo de excepción iniciada.

11.4. Bloque Try...Catch...Finally

Las estructuras de control Try...Catch...Finally comprueban una parte de código y dirigen el modo en que la aplicación debe controlar las distintas categorías de error. Cada una de las tres partes que componen la estructura realiza una función específica en este proceso.

- La instrucción Try proporciona el código en el que se están comprobando las excepciones.
- La cláusula Catch identifica bloques de código que se encuentran asociados con excepciones específicas. Un bloque Catch When hace que el código se ejecute en circunstancias específicas. Una cláusula Catch sin otra When reacciona ante cualquier excepción. Por lo tanto, el código debe disponer de una serie de instrucciones Catch...When determinadas que reaccionen, cada una de ellas, ante un tipo específico de excepción, seguidas de un bloque general Catch que reaccione ante cualquier excepción que no se haya interceptado mediante las cláusulas Catch...When anteriores.
- La instrucción Finally contiene código que se ejecuta independientemente de si ocurre o no una excepción en el bloque Try. Esta instrucción se ejecutará incluso después de Exit Try o Exit Sub. Este código suele realizar tareas de limpieza como el cierre de archivos o la eliminación de búffers.

11.5. Funciones de la cláusula Catch

Esta cláusula puede adoptar tres formas: Catch, Catch...As y Catch...When.

Una cláusula Catch sin la instrucción When permite que el bloque de instrucciones asociado controle cualquier excepción. Las cláusulas Catch...As y Catch...When interceptan una excepción específica y permiten que el bloque de instrucciones asociado indique a la aplicación qué medidas adoptar. Asimismo, estas dos cláusulas se pueden combinar en una sola instrucción como, por ejemplo, Catch ex As Exception When intResult <> 0.

Si la excepción es el resultado de un error de recursos, se debe identificar el recurso en cuestión y proporcionar sugerencias para la solución de problemas. Si la excepción procede de un error de lógicas de programación, es muy probable que la cláusula permita que la aplicación se cierre de la forma más elegante posible. Sin embargo, si un error de usuario ha sido la causa de la excepción, el código debe permitir al usuario corregirlo y poder continuar.

Las cláusulas Catch se comprueban en el orden en el que aparecen en el código. Por lo tanto, este tipo de cláusulas se debe enfocar desde lo específico a lo general a medida que progresan a través de la secuencia de código. Por ejemplo, compruebe un tipo antes de realizar la comprobación de su tipo de base. Un bloque catch que controla a System.Exception solo debe aparecer como bloque final una vez que se hayan agotado las demás posibilidades.

Ejemplo:

```
Public Class Forcad1
  Inherits System.Windows.Forms.Form
  Public Sub h()
    Dim a,b,c as Integer
    Try
     a = InputBox(«Introduce el primer número»)
      b = InputBox(«Introduce el segundo número»)
      If b = 1 Then Throw New MethodAccessException
      c = a / b
    Catch ex As ArithmeticException 'Tipo aritmetico
      c = 0
      MsgBox(«Dentro de Aritmética»)
    Catch ex As InvalidCastException 'Tipo formato
      c = 0
      MsgBox(«Dentro de Formato»)
    Catch ex As Exception
      MsgBox(«Dentro de Excepción»)
   Finally ' Todo Try debe llevar obligatoria/ un Catch o un Finally
      MsgBox(«Dentro de Finally»)
   End Try
   MsgBox(«El resultado es: « & c)
  End Sub
  Private Sub Button1_Click(ByVal sender As
      System.Object, ByVal e As System.EventArgs)
      Handles Button1.Click
       h()
  End Sub
End Class
```