**Programación Orientada a Objetos**

**mayo, 2003**

# INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS EN VISUAL BASIC .NET

**Tomás Montero Ripoll**

**Raúl del Nogal Sánchez**


Departamento de Informática y Automática

Universidad de Salamanca

Información de los autores:

Tomás Montero Ripoll

Estudiante de 3º de Ingeniería Técnica Informática de Sistemas

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 - Salamanca

sete25@latinmail.com

Raúl del Nogal Sánchez

Estudiante de 3º de Ingeniería Técnica Informática de Sistemas

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 - Salamanca

raulillofiesta@hotmail.com

## Resumen

En este documento se pretende hacer un pequeño acercamiento a la programación orientada a objetos en Visual Basic .NET, centrándonos en la creación de clases y jerarquía existente entre ellas.

Se hará especial hincapié en la sintaxis que este lenguaje utiliza, así como en sus características propias; basándonos en los conceptos estudiados en la asignatura y que tienen en común todo los lenguajes de programación orientados a objetos.



## Abstract

In the work, we present an introduction to Object-Oriented Programming in visual Basic .NET. The main topic was can create different classes and its hierarchical analysis.

The syntaxes of this language and the main features are studied in this work. We have used the concepts learnt in this subject. All the languages of objects are based on the same concepts.

# Tabla de Contenidos

# 1. INTRODUCCIÓN

Visual Basic es un descendiente de Basic, que ha existido durante varias décadas. BASIC (el acrónimo de *Beginners´All-Purpose Symbolic Instruction Code*) fue originariamente desarrollado por la *Dartmouth University* en 1964 como lenguaje para programadores principiantes. BASIC se convirtió en el primer lenguaje que la mayoría de los programadores aprendía para familiarizarse con los fundamentos de la programación antes de pasar a lenguajes más potentes.

Visual Basic apareció en mayo de 1991 y supuso una revolución en la metodología de desarrollo de aplicaciones, ya que permitía la creación de programas arrastrando y soltando componentes en lugar de tener que codificar manualmente los elementos. Desde muchos puntos de vista Visual Basic .NET supone la madurez de ese proyecto que nació hace mas de diez años.

La nueva versión, Visual Basic .NET, incorpora características de orientación a objetos de tal manera que podría considerarse un lenguaje totalmente diferente a sus predecesores y totalmente orientado a objetos, ya que en versiones anteriores Visual Basic era considerado un lenguaje de programación con sintaxis de objetos, pero no un verdadero lenguaje orientado a objetos.

Las principales novedades que incorpora son:

- La herencia, tan solicitada por miles de programadores.
- Desarrollo de programas con múltiples hilos de ejecución.
- Control estructurado de excepciones.
- Inicializadores, que permiten establecer los valores iniciales de las variables.

# 2. ÁMBITOS CON NOMBRE

En una aplicación participan un buen número de componentes mediante los cuales se facilita el acceso a bases de datos, la conectividad en redes, comunicación con otras aplicaciones, etc. La existencia de tantos componentes puede causar conflictos si hay coincidencias de nomenclatura, y por ello, la plataforma .NET ha optado por el uso de espacios o ámbitos de nombre.

Un ámbito con nombre, del inglés *namespace*, es un ámbito delimitado explícitamente al que se ha asignado un identificador. En su interior es posible incluir tanto definiciones de tipos como otros ámbitos con nombre, creando una anidación que dará lugar a una jerarquía.

Para crear un ámbito con nombre, incluyendo en él las definiciones que nos interesen, se utiliza la palabra clave `Namespace`. Hay que tener en cuenta que todas las definiciones incluidas en su definición sólo existen dentro del espacio con nombres. Por ejemplo:

```
Namespace Espacio1
      Namespace Espacio2
            Public Class HolaMundo
                  Public Shared Sub Main()
                        System.Console.WriteLine("Hola   desde
Espacio1.Espacio2")
                  End Sub
            End Class
      End Namespace
End Namespace

Namespace Espacio1.Espacio2
            Public Class HolaMundo
                  Public Shared Sub Main()
                        System.Console.WriteLine("Hola   desde
Espacio1.Espacio2")
                  End Sub
            End Class
      End Namespace
End Namespace
```

Ejemplo 2.1

En ambos casos tenemos un ámbito con nombre, llamado `Espacio1`, que contiene en su interior otro, denominado `Espacio2`.

Al haberse definido la clase `HolaMundo` como pública, es posible crear un objeto de dicha clase desde otro programa, siempre y cuando se componga una referencia completa que permitiese al compilador identificarla de manera unívoca. Esto significaría anteponer al nombre de la clase el del ámbito o ámbitos con nombre donde se encuentra definida, como se hace en este ejemplo:

```
Class UsaHolaMundo
    Shared Sub Main()
       Dim MiHolaMundo As Espacio1.Espacio2.HolaMundo = New
Espacio1.Espacio2.HolaMundo()
       End Sub
End Class
```

Ejemplo 2.2

Una alternativa a esta composición de referencias cualificadas consiste en utilizar la sentencia `Imports`, seguida del ámbito cuya referencia no deseamos repetir continuamente. El resultado sería, como se puede ver a continuación, un código más legible.

```
Imports Espacio1.Espacio2
Class UsaHolaMundo
    Shared Sub Main()
       Dim MiHolaMundo As HolaMundo = New HolaMundo()
    End Sub
End Class
```

Ejemplo 2.3

# 3. CLASES

En la programación orientada a objetos (POO), utilizamos los objetos para encapsular la información asociada a las entidades con las que trabaja el programa.

Una clase es un molde que define los atributos y comportamientos de los objetos que se crean como instancias de esta clase. En Visual Basic .NET, la definición de clase es muy similar a la definición de la misma en cualquier lenguaje orientado a objetos. Esta versión .NET incluye como novedad que no es necesario crear un módulo independiente para cada clase.

Una clase puede ser base de otra, así como estar derivada de otra. En ella, puede definirse su comportamiento -métodos- , sus atributos -propiedades-, campos de datos y eventos, e incluso anidar unas clases dentro de otras.

```
Public class ClaseBase
...
End class
```

Ejemplo 3.1

# 3.1. CAMPOS DE DATOS DE LA CLASE

Los campos de datos son variables con un ámbito de visibilidad reducido a la clase. El acceso a dichas variables, por parte del usuario de la clase, puede ser directo o bien a través de métodos o propiedades, todo dependiendo de la visibilidad del campo. En su declaración, podemos controlar su visibilidad mediante unos modificadores, que no sólo pueden aplicarse a la definición de variables, sino también de clases, métodos, estructuras...

Por defecto, los miembros de una clase tienen visibilidad pública o privada dependiendo de su tipo. Si a una variable le anteponemos la palabra `Dim` sin ningún modificador, será privada a la clase. Por el contrario, si hacemos esto mismo a un método, éste será público. Para resaltar la privacidad de un miembro, usaremos la palabra `Private`, asegurando que únicamente podrá ser manipulado dentro de la propia clase, en el código de los métodos y propiedades, pero nunca fuera, ni siquiera en clases derivadas.

Por otro lado, si queremos presentar las variables en modo público, antepondremos la palabra `Public`, y así podrán usarse tanto dentro de la clase como fuera, ya sea por clases derivadas u otras que no tengan parentesco alguno.

Destacan al mismo tiempo otros modificadores:

- ✓ `Protected:` Son accesibles internamente, y también por aquellas clases que hayan derivado de ésta.
- ✓ `Friend:` Permite el acceso desde cualquier punto del proyecto al que pertenece la clase.
- ✓ `Protected Friend:` Los miembros son visibles en el proyecto actual y en las clases derivadas, aunque estén en otro proyecto.

```
Public class ClaseBase
  Dim AmbitoDefecto As Integer
  Private AmbitoPrivado As Integer
  Protected AmbitoProtegido As Integer
  Friend AmbitoInterno As Integer
  Protected Friend AmbitoProtegidoeInterno As Integer
  Public AmbitoPublico As Integer
End class
```

Ejemplo 3.1.1

Desde cualquier método que pertenezca a la clase `ClaseBase` se puede acceder a todas las variables independientemente de su modificador. Si tuviésemos una clase derivada, la cual heredaría todas las variables, no podría acceder a las dos primeras por tener un ámbito privado. Si tuviésemos una clase totalmente ajena a `ClaseBase`, sólo podríamos acceder a las tres últimas variables.

## 3.2. PROPIEDADES DE LA CLASE

Las clases almacenan información que, en muchas ocasiones, no sólo es para uso interno sino que también puede interesar al programador que las utilice.

Si, por ejemplo, se desea que el valor de una cierta variable pueda ser leído externamente pero no modificado, o bien que cada vez que se modifique su valor éste pueda ser controlado para saber si es válido, es adecuado definir una propiedad.

Una propiedad es una variable que tiene vinculado internamente unos métodos de acceso que son los que controlan la lectura o asignación de valores.

Dentro del bloque de código de propiedades existe un bloque `Get`, que devuelve al usuario de la clase el valor de la instancia privada de la variable; y un bloque `Set`, que asigna el valor indicado por el código cliente a la correspondiente instancia privada de la variable.

Al mismo tiempo, es posible restringir el acceso a las propiedades utilizando el comando `ReadOnly`, y eliminando el bloque `Set`, para convertirla en una propiedad de sólo lectura; o utilizar el comando `WriteOnly` y eliminar el bloque `Get`, para convertir la propiedad en una propiedad de sólo escritura. Por ejemplo:

```
Class Ficha
   Private Pnombre As String
   Private Pdireccion As String
   Private Pdepartamento As String
   Private Pidempleado As Integer

   //La propiedad Nombre sólo puede leerse, al igual que sucede con Direccion
   Public ReadOnly Property Nombre() As String
        Get
             Return Pnombre
        End Get
   End Property

  // La propiedad Departamento puede leerse y escribirse, al igual que sucede con
Idempleado.
   Public Property Departamento() As String
        Get
             Return Pdepartamento
        End Get
        Set (ByVal Value As String)
             Pdepartamento = Value
        End Set
   End Property
End Class
```

Ejemplo 3.2.1

## 3.3. MÉTODOS DE LA CLASE

Los métodos definen el comportamiento de las clases. Son bloques de código delimitados, a los cuales se les asigna un identificador para poder efectuar una o más llamadas y ejecutar el código las veces necesarias.

En un método hay dos posibilidades, que devuelva algún dato, o que por el contrario no devuelva nada. Si del primer caso se trata se antepone la palabra `Function`, para el segundo caso se usa la palabra `Sub`.

Los parámetros que recibe el método pueden estar pasados por valor o por referencia:

-Valor: El método recibe una copia del valor almacenado en la variable original. El paso por valor se usa por defecto, pero si queremos resaltarlo se antepone la palabra `ByVal`.

-Referencia: El método recibe una referencia a esa variable, lo cual permite modificarla. Para indicar que el paso es por referencia se antepone la palabra `ByRef`.

```
Public Sub asignacion ( ByVal Pnombre As String, ByVal
Pdireccion As String)
    If Pnombre = "David" And Pdireccion = "Santa Marta" Then
      Pidempleado = 1
      Departamento = "IS"
    ElseIf Pnombre = "Jesus" And Pdireccion = "Salamanca" Then
      Pidempleado = 2
      Departamento = "AC"
    Else
     Throw New Exception ("Acceso incorrecto.")
    End If
End Sub
```

Ejemplo 3.3.1

En cuanto a la devolución de parámetros desde un método, lo único que hay que hacer es sustituir la palabra `Sub`, que hemos puesto delante del nombre del método, por `Fuction` y añadir al final la palabra Al seguida del tipo que corresponda. En el cuerpo del método usaríamos la sentencia `Return` acompañada del valor a devolver.

## 3.4. EVENTOS DE LA CLASE

Se entiende por evento una señal generada por un objeto o componente. Esta señal puede conectarse a un gestor de evento, un método que se ejecutaría automáticamente al recibir esa señal.

Los eventos proporcionan una forma de que un objeto ejecute el código escrito por el usuario del objeto. El objeto desencadena el evento mediante un controlador de eventos.

Para declarar un evento se utiliza la palabra clave `Event`, que indica a la clase el nombre del evento:

```
Public Event SalarioError (ByVal Error As String)
```

Una vez se le ha indicado a la clase la posibilidad de producirse un evento, es necesario introducir el código para producir el evento:

```
Public Property Salario( ) As Decimal
      Get
            Return Psalario
      End Get
      Set (ByVal Valor As Decimal)
            If Valor >= 0 Then
                  Psalario = Valor
            Else
                  RaiseEvent SalarioError (" Salario no puede ser
negativo.")
            End If
      End Set
End Property
```

Ejemplo 3.4.1

## 3.5. CONSTRUCTORES Y DESTRUCTORES

Las clases definidas en Visual Basic .NET pueden contar con unos métodos específicos para controlar la construcción y destrucción de objetos. Una misma clase puede contar con múltiples

constructores, siempre que cuenten con diferentes listas de parámetros; mientras que destructor sólo puede haber uno por clase.

Los constructores tienen por nombre `New()`, no cuentan con valor de retorno, pueden tomar o no parámetros según interese, y pueden inicializar las propiedades del objeto instancia o establecer una conexión con una base de datos.

El destructor tiene por nombre `Finalize()`, no cuenta con valor de retorno y no pueden tomar parámetro alguno.

Al crear un objeto de una clase, utilizando para ello el operador `New`, se ejecuta automáticamente el constructor que corresponda según los parámetros facilitados. Sin embargo, el destructor, se ejecuta automáticamente para un cierto objeto cuando ya no hay ninguna referencia hacia él, es decir, no necesitamos destruir los objetos explícitamente, de esto se encarga el sistema de recogida de basura de la plataforma .NET. La recogida de objetos no útiles se efectúa en momentos concretos en los que no hay ninguna otra actividad pendiente.

```
Class Agenda
  Public Sub New()   //Constructor sin parámetros
  End Sub
  Public Sub New (ByVal Parametro As String)   //y con un parámetro
  End Sub
  Protected Sub Finalize()   //Destructor
  End Sub
End Class

//Creación de un objeto sin entregar ningún parámetro
Dim MiAgenda As Agenda = New Agenda()
//Creación de un objeto facilitando una cadena
Dim Miagenda As Agenda = New Agenda("Parámetro")
```

Ejemplo 3.5.1

Obsérvese que el destructor es un procedimiento protegido, ya que no es accesible desde el exterior de la clase y se ejecuta automáticamente

Visual Basic .NET permite el uso de miembros compartidos que no pertenecen a un objeto en particular, sino a toda la clase, por lo que son creados la primera vez que es utilizada la clase

y no cada vez que se crea un objeto. Para ello disponemos del concepto de **constructor de clase**, el cual no puede tomar parámetros y cuyo único modificador posible, obligatorio además, es `Shared`. Este constructor únicamente se ejecuta cuando se aloja la clase en memoria.

## 3.6. CLASES ANIDADAS

Consiste básicamente en definir otras clases en el interior de una clase, esto es útil cuando una cierta clase tan solo va a ser necesaria en el interior de otra, nunca fuera.

Al anidar una clase en otra es también una forma de ocultarla, ya que si no se la hace publica, seria imposible usarla desde el exterior aun poniendo una referencia completa. Conseguimos la encapsulación de una clase dentro de otra.

```
Class Agenda
      Class Anotación
            Shared Sub New()
                  Console.writeLine("Constructor de la clase
anotación")
            End Sub

            Public Sub New()
                  Console.writeLine("Constructor de anotación")
            End Sub

            Public Sub New(ByVal Parametro As String)
                  Console.writeLine("Constructor {0}",parámetro)
            End Sub
            Protected Sub Finalize()
                  Console.writeLine("Destructor de anotación")
            End Sub
      End class
//Las siguientes declaraciones pertenecen a la clase agenda
      Private MiAnotacion As Anotacion
      Private Sub New()
            Console.writeLine("Constructor agenda")
            Mianotacion = New Anotación ("Nueva anotación")
      End Sub
```

```
      Protected Sub Finalize()
            Console.writeLine("Destructor agenda")
            Mianotacion = Nothing
      End Sub
End Class

//Creamos una que contenga el Main()
Class Principal
      Shared Sub Main()
            Dim MiAgenda As Agenda = New Agenda()
            Console.WriteLine("Objeto creado")
            MiAgenda = Nothing
            Console.WriteLine("Fin de la aplicación")
      End Sub
End Class
```

Ejemplo 3.6.1

El resultado que se obtendría por pantalla sería:

Constructor de Agenda

Constructor de la Clase Anotación

Constructor de Nueva Anotación

Objeto creado

Fin de la aplicación

Destructor de Anotación

# 4. JERARQUÍA DE CLASES

La herencia es una de las funciones más potentes y fundamentales de cualquier lenguaje de programación orientado a objetos, ya que es posible crear una clase base que encapsule las propiedades y métodos que serán necesarios en múltiples clases derivadas del mismo tipo. Otra de estas características fundamentales es el polimorfismo, gracias al cual se pueden definir en una clase base métodos que serán implementados por las clases que se deriven de ella.

## 4.1. HERENCIA

Gracias a la herencia podemos crear clases base en las que se encapsulen las funciones más comunes. Posteriormente, podemos crear otras clases que se deriven de las clases base. Las clases derivadas heredan las propiedades y métodos de las clases base y pueden ampliar o complementar sus funciones para adaptarse a los requerimientos del programa.

Para crear una clase derivada en Visual Basic .NET se incluye en su definición el comando `Inherits` junto con el nombre de la clase base. Por ejemplo:

```
 Public Class Cuenta
       Public PnumeroCuenta As Long
       Public Property NumeroCuenta() As Long
       Get
            Return PnumeroCuenta
       End Get
       Set (ByVal Value As Long)
            PnumeroCuenta = Value
       End Set
       End Property
       Public Function HacerBalance () As Double
       //Código para obtener el balance de la cuenta de la base de datos.
       End Function
End Class

Public Class CuentaCorriente
       Inherits Cuenta
       Private MinBalance As Double
       Public Sub Retirar (ByVal Cantidad As Double)
       //Código para retirar dinero de la cuenta.
       End Sub
End Class

Dim Ahorros As CuentaCorriente = New CuentaCorriente ()
Ahorros.NumeroCuenta = 2000
```

```
Ahorros.HacerBalance = ()   //Método definido por la clase Cuenta heredado
                            //Por la clase CuentaCorriente.
Ahorros.Retirar (500)       //Método definido por la clase CuentaCorriente.
```

Ejemplo 4.1.1

En ciertas ocasiones, interesa que no se tenga autorización para crear instancias de una clase base, forzando que el acceso a los métodos y propiedades de la clase se realice a través de una clase derivada. En este caso, construiríamos la clase base utilizando el modificador `MustInherit`. En el siguiente código se muestra la definición de la clase Cuenta con el modificador `MustInherit`:

```
Public MustInherit Class Cuenta
```

Esta definición convierte a `Cuenta` en una clase abstracta, ya que define las interfaces de los métodos y propiedades que serán heredadas por las clases derivadas. Así, para poder acceder al método `HacerBalance` será necesario crear una instancia de la clase derivada `CuentaCorriente`.

Por defecto, todas las clases pueden tener herencia. Si no se tiene cuidado, es posible que generemos cadenas de herencia muy complicadas que resultan difíciles de administrar y depurar. Utilizando el modificador `NotInheritable`, es posible crear clases con la total certeza de que no darán lugar a clases derivadas. Este tipo de clases suele denominarse clase sellada o final. Ejemplo:

```
Public NotInheritable Class CuentaCorriente
```

## 4.2. POLIMORFISMO

El polimorfismo es la habilidad que tienen objetos basados en diferentes clases para responder a la misma llamada de método utilizando sus propias implementaciones. Los métodos heredados por las clases derivadas pueden someterse a sobrecarga, para ello utilizamos la palabra clave `OverLoads`. La signatura de método de la clase sobrecargada debe utilizar el mismo nombre que el método sobrecargado, pero la lista de parámetros debe ser diferente. Es exactamente lo

mismo que sobrecargar métodos de la misma clase, excepto la palabra clave `Overloads` es opcional y, normalmente, se omite.

```
Public Class Cuenta
      Public Sub Retirar (ByVal Cantidad As Double)
      //Código de implementación
      End Sub
End Class

Public Class CuentaCorriente
      Inherits Cuenta
      Public Overloads Sub Retirar (ByVal Cantidad As Double,
ByVal MinnumBalance As Double)
      //Código de implementación
      End Sub
End Class
```

Ejemplo 4.2.1

En el siguiente ejemplo, se realizará una iteración a través de la colección de clases de tipo cuenta, y el compilador determinará en tiempo de ejecución qué implementación concreta de tipo cuenta debe ejecutar. De este modo, no hay que preocuparse de a qué tipo de clase estamos haciendo referencia, pues los tipos de clase implementan las mismas interfaces de método. Así, nos interesa que todas las clases de cuentas contengan un método `ObtenerInfoCuenta` con la misma definición de interfaz, pero con diferentes implementaciones que dependerán del tipo de cuenta.

Para poder reemplazar un método heredado en la clase derivada, usamos la palabra clave `Overrides` en la definición del mismo.

```
Public MustInherit Class Cuenta
    Public MustOverride Function ObtenerInfoCuenta () As String
End Class
```

```
Public Class CuentaCorriente Inherits Cuenta
     Public Overrides Function ObtenerInfoCuenta () As String
          Return "Imprimiendo información de cuenta corriente"
     End Function
End Class

Public Class CuentaAhorro Inherits Cuenta
     Public Overrides Function ObtenerInfoCuenta () As String
          Return "Imprimiendo información de cuenta de ahorros"
     End Function
End Class
```

Ejemplo 4.2.2

También se puede obtener un resultado similar utilizando una interfaz, que definirá las firmas de los métodos. En lugar de heredar de una clase base `Cuenta`, definiremos una interfaz `ICuenta`, y las clases que la implementen deben aportar el código de implementación de todos los métodos definidos por ella:

```
Public Interface ICuenta
    Function ObtenerInfoCuenta () As String
End Interface

Public Class CuentaCorriente Implements ICuenta
     Public Function ObtenerInfoCuenta () As String Implements
ICuenta.ObtenerInfoCuenta
          Return "Imprimiendo información de cuenta corriente"
     End Function
End Class

Public Class CuentaAhorro Implements ICuenta
     Public Function ObtenerInfoCuenta () As String Implements
ICuenta.ObtenerInfoCuenta
          Return "Imprimiendo información de cuenta de ahorros"
     End Function
End Class
```

Ejemplo 4.2.3

# 5. CONCLUSIONES

• Al igual que sus predecesores, Visual Basic .NET sigue siendo un lenguaje potente y fácil de aprender.

• Las clases pueden definir en su interior campos de datos, propiedades, métodos, eventos e incluso permite en anidamiento entre clases.

• Con la introducción de la herencia en Visual Basic .NET, se puede considerar esta versión totalmente orientada a objetos.

• Permite tanto sobrecarga (redefinición) de métodos heredados, como reemplazo del cuerpo de dichos métodos.

# 6. REFERENCIAS

**Francisco Charte Ojeda.** "Microsoft Visual Basic .NET" Ediciones Anaya Multimedia (Grupo Anaya, S.A.). 2001.

**Dan Clark.** "Introducción a la Programación Orientada a Objetos con Visual Basic .NET" Ediciones Anaya Multimedia (Grupo Anaya, S.A.). 2003.

**Luis Miguel Blanco.** "Programación en Visual Basic .NET" Editorial Eidos. 2002.

**Web Site**. http://guille.costasol.net/NET/cursoVB.NET