palasi.com
IT and management

Introducción a Visual Basic .NET

Dr. Vicent-Ramon Palasí Lallana
Aurum Solutions, S.A. de C.V.

Presentación del profesor

- Dr. Vicent-Ramon Palasí Lallana.
  - Licenciado en Informática por Universidad Politécnica de Cataluña (Barcelona, España)
  - Doctor en Ingeniería Informática por Universidad Jaume I (Castellón, España).
  - Diez años de experiencia profesional en el desarrollo de sistemas empresariales y en docencia pública y privada.
  - En la actualidad, Gerente General de Aurum Solutions.

Aurum Solutions, S.A. de C.V.

- Empresa que se dedica al desarrollo de aplicaciones de alto nivel.
- Plataformas más usadas: Java (J2EE) y .NET.
- Algunos clientes:
  - Empresa Bonal (España)
  - Secretaría Técnica de la Presidencia.
  - Geotérmica Salvadoreña.
  - Algunos Ministerios de El Salvador.

Presentación de los participantes

- Nombre, a qué se dedican.

Temario del curso

- 1. Introducción a .NET y Visual Basic .NET
- 2. Un primer programa en VB .NET
- 3. Introducción a la programación orientada a objetos.
- 4. Otros conceptos básicos de VB.NET.
- 5. Más sobre la programación orientada a objetos con VB.NET.
- 6. Conclusión. Los siguientes pasos.

Temario del curso

- 1. Introducción a .NET y Visual Basic .NET
- 2. Un primer programa en VB .NET
- 3. Introducción a la programación orientada a objetos.
- 4. Otros conceptos básicos de VB.NET.
- 5. Más sobre la programación orientada a objetos con VB.NET.
- 6. Conclusión. Los siguientes pasos.

Mail: palasi@palasi.com Web: www.palasi.com

Objetivos

- Introducir a la programación de sistemas de computadora mediante el lenguaje Visual Basic .NET
- Presentar la plataforma .NET y explicar su relación con Visual Basic .NET.

Prerrequisitos

- Este curso supone que el asistente tiene conocimiento de Windows (NT, XP o 2000) y Visual Basic 6.
- No hay otros prerrequisitos a parte de éstos.

Advertencia

- Visual Basic .NET tiene una curva de aprendizaje mucho más lenta que Visual Basic 6.
- Por esto, no se pretende que, al acabar este curso, los asistentes sean capaces de realizar aplicaciones de nivel empresarial.
- Para ello hacen faltan cursos adicionales: éste es un curso de introducción, que explica lo más básico.

En pocas palabras

- 1. ¿Qué es Visual Basic .NET? Es un nuevo lenguaje de programación.
- 2. ¿Qué es .NET? Es una nueva plataforma de programación, para programar y ejecutar programas.

Con más detalle

- 1.1. El lenguaje Visual Basic .NET
- 1.2. La plataforma .NET

Con más detalle

- 1.1. El lenguaje Visual Basic .NET
- 1.2. La plataforma .NET

1.1 El lenguaje Visual Basic .NET

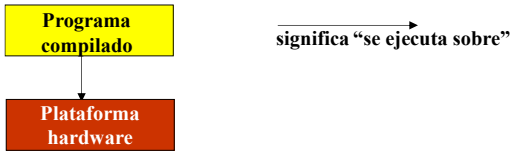
- Es un nuevo lenguaje de programación desarrollado por Microsoft para los sistemas operativos Windows.
- Contrariamente a lo que se podría pensar, Visual Basic .NET no es el mismo lenguaje que Visual Basic. En este caso, el nombre engaña.
- Es un lenguaje completamente diferente y con una nueva filosofía.

La primera diferencia

- VB.NET es un lenguaje orientado a objetos.
- VB 6 no lo era. Le faltaba el concepto de herencia y además se podía programar ignorando el concepto de clase.
- En VB.NET, como en cualquier lenguaje orientado a objetos, hay herencia y todo es una clase.
- Definiremos las clases más adelante. Por ahora, piensen en una clase como una especie de módulo de programación.
- Un programa VB.NET no es más que un conjunto de clases.

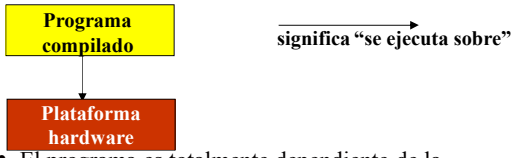
Más diferencias

- Otra diferencia es la ejecución. En un lenguaje convencional.



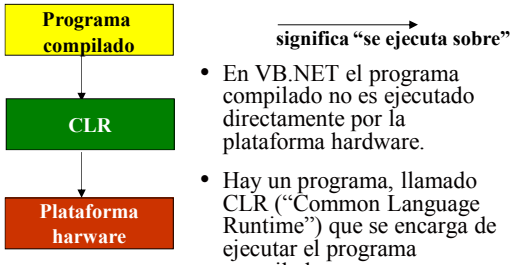
- La plataforma hardware es el conjunto del hardware y del sistema operativo.
- El programa está compilado específicamente para una determinada plataforma hardware, por lo tanto es ejecutable directamente sobre la plataforma.

Desventajas de este enfoque



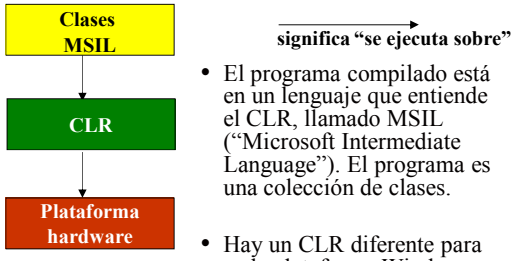
- El programa es totalmente dependiente de la plataforma hardware.
- Si conseguimos un programa en Visual Basic 6 y lo queremos ejecutar en un Pocket PC no funciona.
- Esto es malo, pues cada vez las plataformas son más diversas (celulares, tablet PC, handhelds, etc.)

Ejecución de Visual Basic .NET



- En VB.NET el programa compilado no es ejecutado directamente por la plataforma hardware.
- Hay un programa, llamado CLR ("Common Language Runtime") que se encarga de ejecutar el programa compilado.
- Es decir, se añade una capa de indirección.

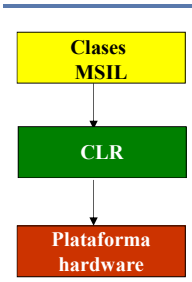
Una cuestión importante



- El programa compilado está en un lenguaje que entiende el CLR, llamado MSIL ("Microsoft Intermediate Language"). El programa es una colección de clases.
- Hay un CLR diferente para cada plataforma Windows y todos los CLRs entienden el mismo lenguaje MSIL.

Mail: palasi@palasi.com Web: www.palasi.com

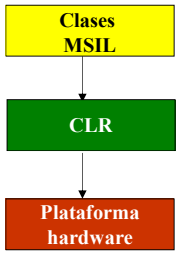
¿Qué ventajas da eso?



significa "se ejecuta sobre"

- El programa se convierte en independiente de la plataforma hardware.
- El programa compilado en MSIL se puede ejecutar en cualquier plataforma para la cual haya un CLR.
- Se consigue una independencia de la plataforma harware.

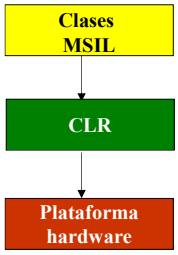
Esto es útil



significa "se ejecuta sobre"

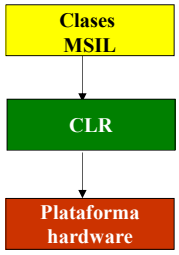
- Ahora puedo compilar mi programa VB.NET en una PC y copiarlo en una Pocket PC.
- El programa funcionará sin cambios, pues ambas plataformas tienen un CLR que entiende el MSIL.
- Ahora resulta tan fácil programar en una plataforma como en todas las plataformas que tienen CLR.

Observación



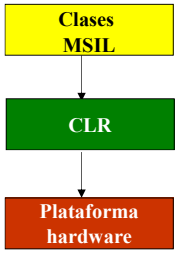
- Si el CLR estuviera implementado en "todas" las plataformas (al estilo de Java), esto sería fantástico, pues sólo habría que hacer un programa y correría en todas las computadoras y aparatos similares.
- Sin embargo, Microsoft sólo ha implementado el CLR en las plataformas Windows.
- Ximian tiene un proyecto para implementarla en Linux (Mono).

Por ello



- Al menos por ahora (y probablemente en el futuro), la independencia de la plataforma que el CLR provee sólo está disponible para Windows.
- Plataforma = SO+ hardware. VB.NET sólo nos da independencia del hardware y del sistema operativo mientras sea Windows (Windows XP, Windows Tablet PC, Windows CE, etc).

Otra ventaja

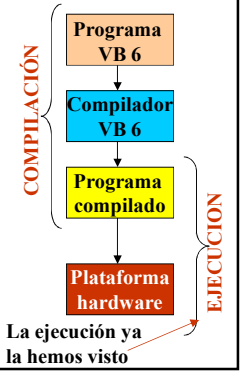


significa "se ejecuta sobre"

- Como ahora la ejecución, está controlada por el programa CLR, ahora puede detectarse si el código es seguro.
- CLR no dejará ejecutar código malicioso. Además provee otros servicios útiles, como multithreading, recolección de basura, etc.
- Los programas son más seguros y de mejor calidad. Están mejor controlados.

Eso en cuanto la ejecución, pero ¿qué hay de la compilación?

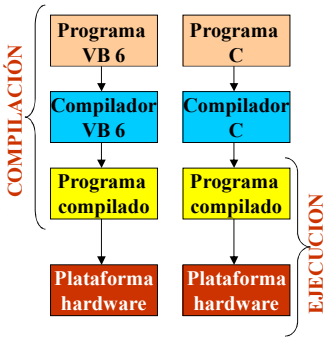
- En un lenguaje convencional, el compilador traduce el código fuente a un programa compilado específicamente para la plataforma.



Mail: palasi@palasi.com Web: www.palasi.com

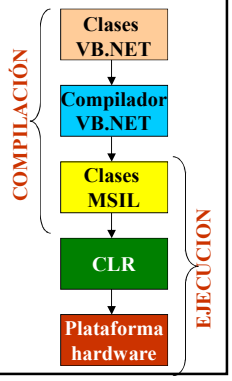
Esto quiere decir que si hay dos programas en diferentes lenguajes

- Cada uno es compilado en un binario diferente y su código no se puede combinar.
- Los programas de un lenguaje de programación son un mundo diferente de los del otro lenguaje. No se pueden combinar.



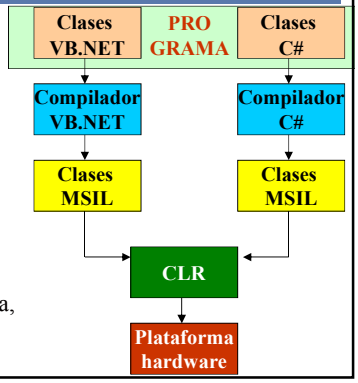
Esto cambia en VB.NET

- Como ahora, el programa compilado no se ejecuta sobre la plataforma sino sobre un programa llamado CLR.
- Esto nos permite programar en el CLR una serie de aspectos interesantes.



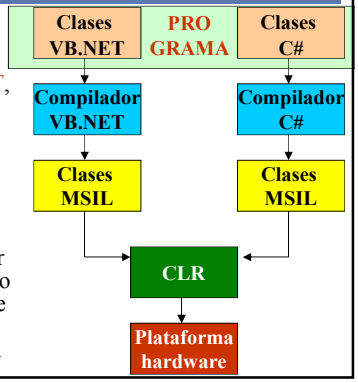
Por ejemplo, programas con varios lenguajes

- Así, podemos tener un programa que combine clases programadas en VB .NET, con clases programadas en C#, con clases programadas en COBOL. NET.
- Todo ello forma un solo programa, que es ejecutado por el CLR.



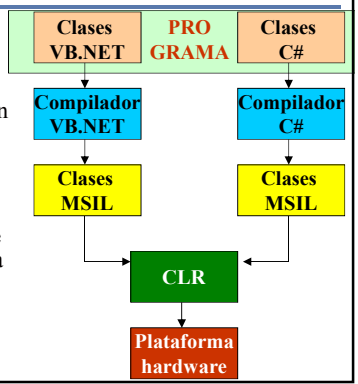
Por ejemplo, programas con varios lenguajes

- Mientras el lenguaje sea adaptado a la plataforma .NET, podemos programar en el lenguaje que prefiramos.
- Así, un programador puede programar en VB. NET pero usar las clases de un colega que programa en C#.



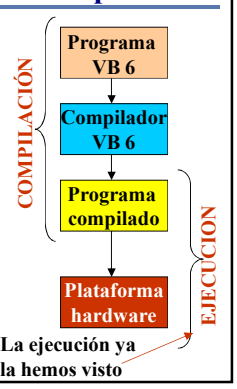
De ello, viene el nombre del CLR

- Common Language Runtime quiere decir "Rutina de Ejecución común a los lenguajes".
- Todos los lenguajes de programación se ejecutan sobre la misma rutina: el CLR.



Eso en cuanto la ejecución, pero ¿qué hay de la compilación?

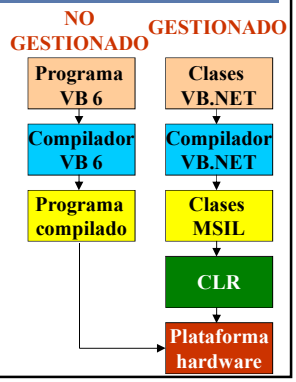
- En un lenguaje convencional, el compilador traduce el código fuente a un programa compilado específicamente para la plataforma.



Mail: palasi@palasi.com Web: www.palasi.com

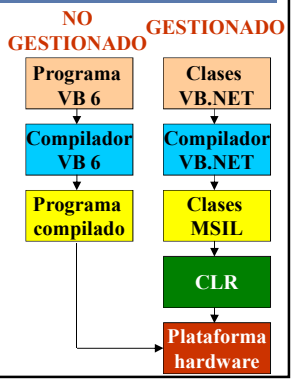
Código gestionado frente a código no gestionado

- Se llama código gestionado ("managed code") a los programas que se ejecutan sobre el CLR.
- Se llama código no gestionado a los programas que se ejecutan directamente sobre la plataforma.



Código gestionado frente a código no gestionado

- El código de los lenguajes .NET es código gestionado.
- Los lenguajes que no son .NET tienen código no gestionado.



Relación de Visual Basic .NET con otros lenguajes

- Visual Basic .NET adopta los principales conceptos y filosofía de Java, que es el lenguaje anterior a él al que más se parece.
- Visual Basic .NET usa una sintaxis parecida a Visual Basic 6. También los controles son parecidos.
- Visual Basic .NET se relaciona con otros lenguajes como C#, Visual C++.NET, etc. pues todos se pueden combinar en un solo programa, como acabamos de ver.

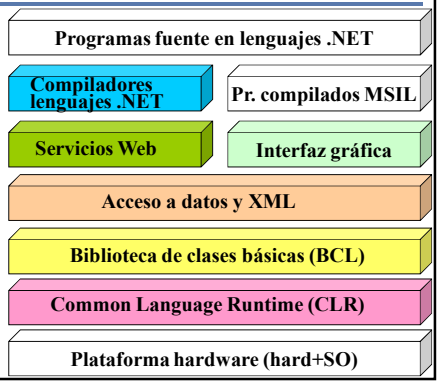
Con más detalle

La plataforma .NET (".NET framework")

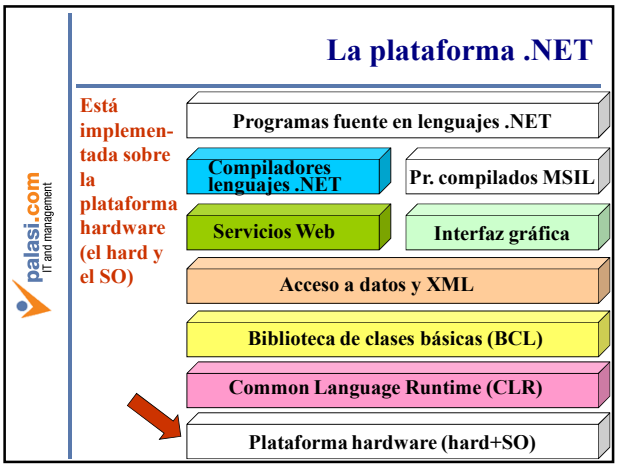
- Es una plataforma de programación, es decir, una plataforma para programar y ejecutar programas.
- Contiene una serie de recursos que hacen sencillo programar y ejecutar programas, desde el CLR, compiladores, librerías de clases predefinidas.
- La idea es que gran parte de las cuestiones más difíciles (concurrencia, acceso al Web, etc.) ya estén programadas en la plataforma y listas para utilizar.
- Así, la programación es más sencilla y los programas resultantes son más robustos, potentes y adecuados a los nuevos tiempos.

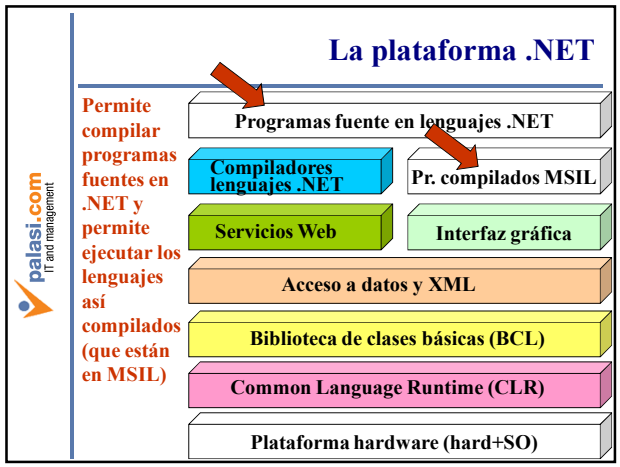
La plataforma .NET

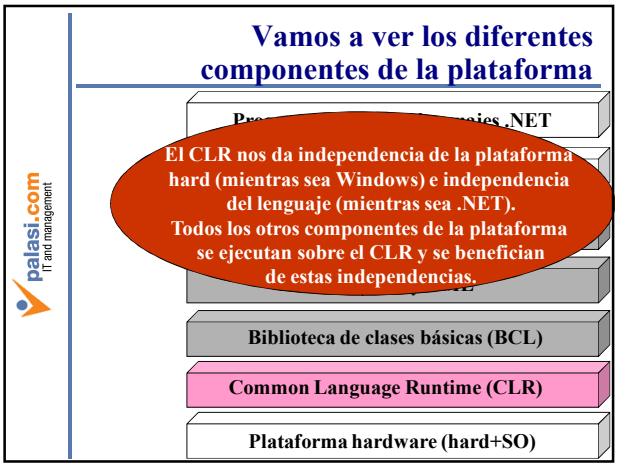
Es todo lo que está en color. Lo que está en blanco, sólo se incluye para ver cómo se relaciona con otros elementos.

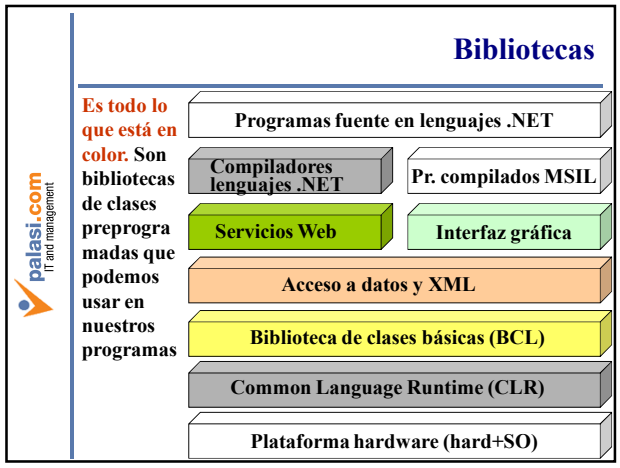


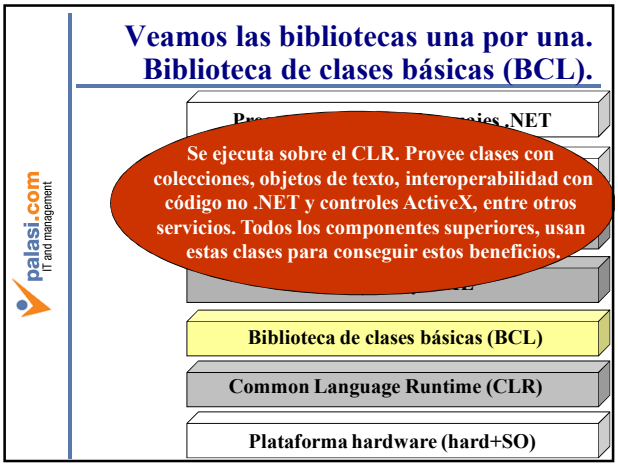
Mail: palasi@palasi.com Web: www.palasi.com

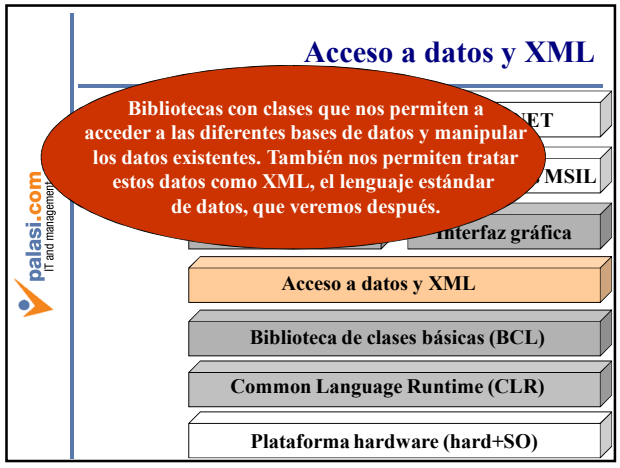




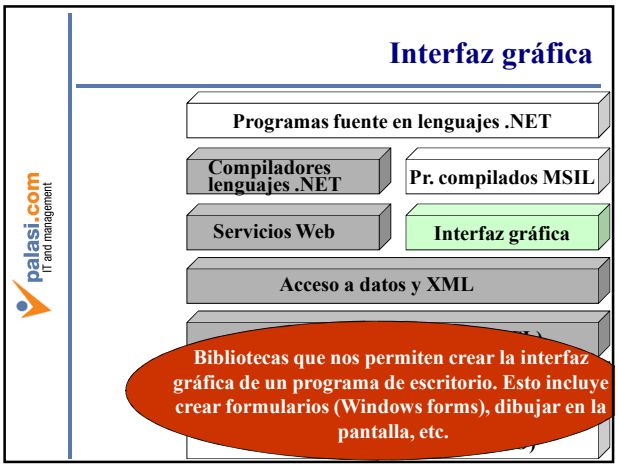


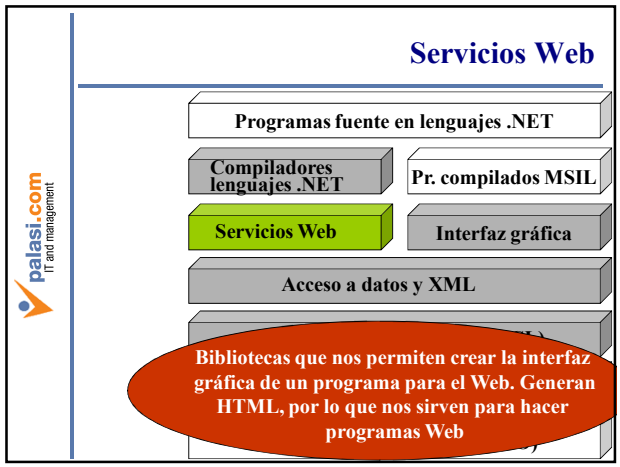


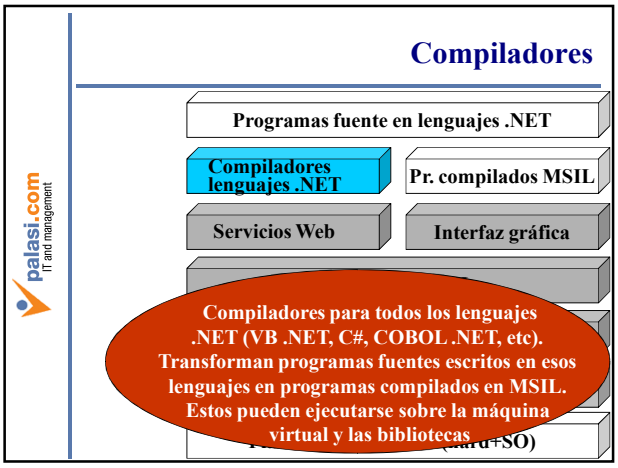


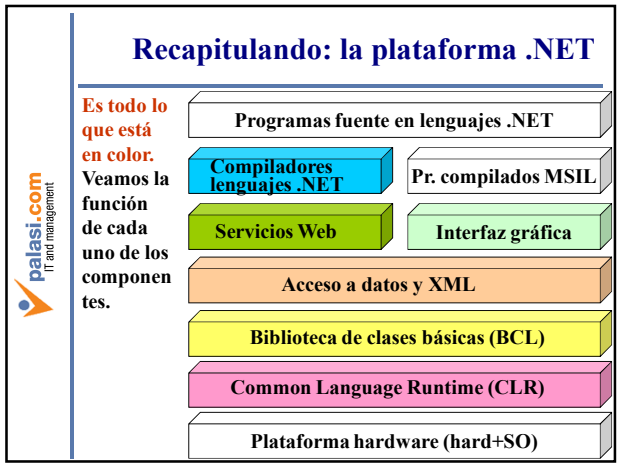


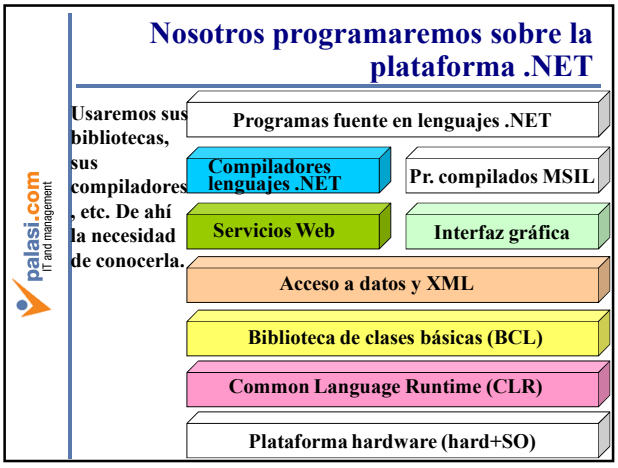
Mail: palasi@palasi.com Web: www.palasi.com

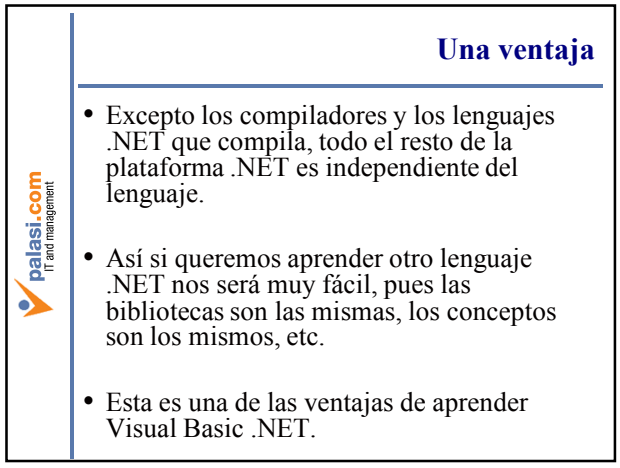












Mail: palasi@palasi.com Web: www.palasi.com

Un inconveniente

- Parte de la plataforma .NET es necesaria para ejecutar programas .NET
- Esto plantea dos tipos de problemas:
  - Técnicos. ¿Cómo hacemos para instalar esa parte de la plataforma en las computadoras que ejecutan programas .NET?
  - Legales. ¿Tenemos permiso para instalar esta parte de la plataforma?

Primero tratemos los problemas técnicos

- La parte de la plataforma que necesitamos para ejecutar los programas .NET se encuentra en un archivo de instalación llamado `dotnetfx.exe`
- Este archivo se puede descargar del sitio de Internet de Microsoft (Microsoft Windows Update) o bien está en el último disco de instalación de Visual Studio .NET.
- Para instalarlo, sólo debe ejecutarse y ya podremos ejecutar programas .NET en la computadora.
- Por supuesto, sólo hace falta instalarlo en las máquinas que no tengan Visual Studio .NET

Después soluciones a los problemas legales

- Este archivo es distribuible libre y gratuitamente.
- Tiene algunas condiciones en la licencia (EULA). Sobre todo, limitaciones de responsabilidad a Microsoft por cualquier daño.
- En general, puede distribuirse libremente sin problemas mientras aceptamos y cumplamos la licencia.

Temario del curso

2. Un primer programa en VB.NET

2. Un primer programa en VB.NET

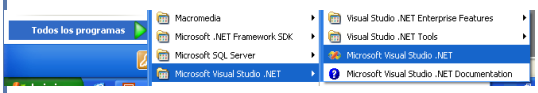
Mail: palasi@palasi.com Web: www.palasi.com

EL IDE de Visual Basic.NET

- Un IDE (Integrated Development Environment o entorno integrado de desarrollo) es aquel entorno gráfico en el que se crean y compilan programas de un determinado lenguaje de programación.
- Igual que casi todos los lenguajes de programación actuales, Visual Basic.NET tiene un IDE. VB6 también lo tenía.
- Muchas veces cuando hablamos de VB.NET, nos estamos refiriendo al IDE de VB.NET.

Entrando al (IDE de) Visual Basic .NET

- Entren a **Todos los programas| Microsoft Visual Studio .NET |Microsoft Visual Studio .NET**

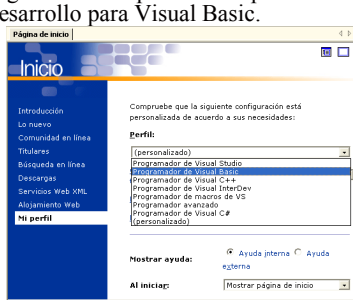


En la barra azul de la izquierda

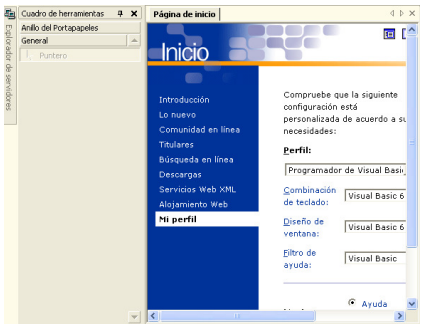
- Hagan clic en "Mi perfil".

Despliegue la lista que hay bajo de "Perfil"

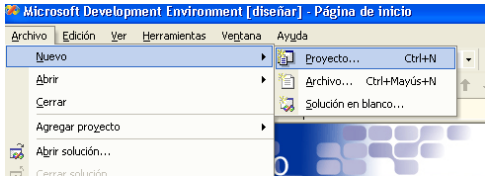
- Elijan "Programador de Visual Basic". Con ello, configurarán la disposición en pantalla del entorno de desarrollo para Visual Basic.



Aparecerá una pantalla así.



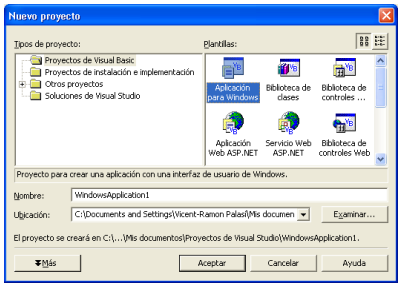
Seleccionamos Archivo|Nuevo|Proyecto



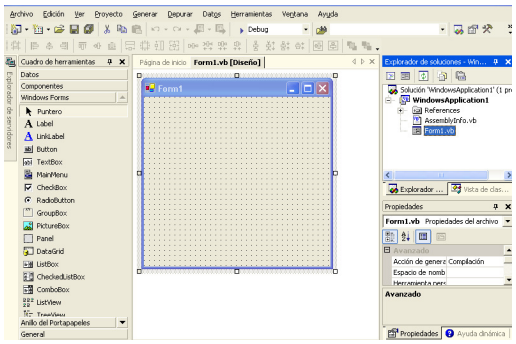
- En la barra de menús superior.

En la ventana que aparece

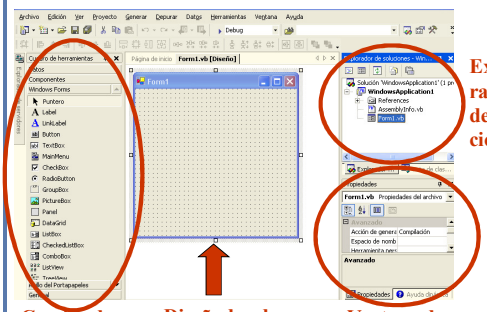
- Seleccionamos “Aplicación para Windows” y hacemos clic en “Aceptar”.



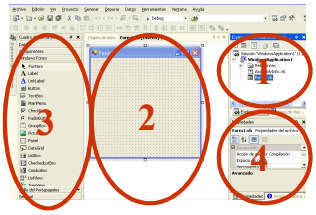
Aparece la ventana principal de Visual Basic .NET



Elementos de la ventana principal de Visual Basic .NET



Elementos de la ventana principal de Visual Basic .NET

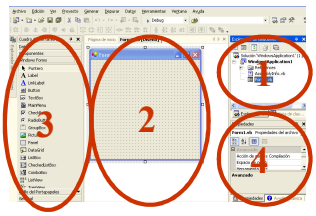


- **1. Explorador de soluciones.** Sirve para gestionar todos los archivos que hay en la programación de un sistema. Es análogo al explorador de proyectos de Visual Basic 6.

¿Por qué se llama “explorador de soluciones”?

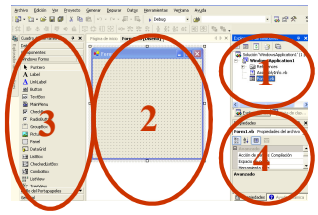
- En VB6, los archivos para el desarrollo de un sistema, se agrupaban en proyectos y, por lo tanto, el explorador se llamaba “explorador de proyectos”.
- Se podían manejar varios proyectos al mismo tiempo con un “grupo de proyectos”, pero esto era muy incómodo y engorroso.
- En Visual Basic .NET, al grupo de proyectos se le llama “solución”. Un desarrollo estará compuesto de varios proyectos que se agrupan en una solución. De ahí el nombre.

Elementos de la ventana principal de Visual Basic .NET



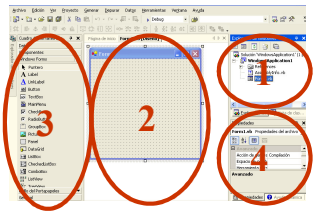
- **2. Diseñador de formularios.** Es aquí donde diseñaremos los formularios de nuestro programa. Es análogo al diseñador de formularios de Visual Basic 6.

Elementos de la ventana principal de Visual Basic .NET



- **3. Cuadro de herramientas.** Como en VB6, aquí es donde están los controles que podemos añadir a nuestro formulario: etiquetas, cuadros de texto, etc.

Elementos de la ventana principal de Visual Basic .NET



- **4. Ventana de Propiedades.** Como en VB6, aquí tendremos las propiedades de nuestros controles gráficos y formularios.

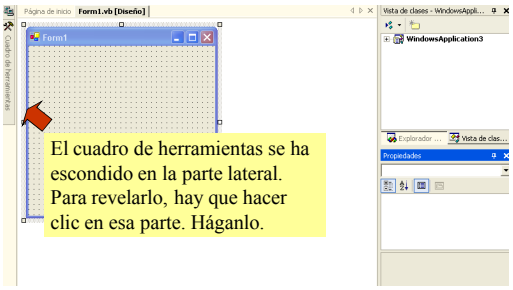
Organizando esos elementos

- Son muchas ventanas para una sola pantalla. Es difícil trabajar con tantas ventanas y tan pequeñas.
- Afortunadamente, hay una solución para esto.
- En las ventanas laterales (en todas menos el Diseñador de formularios), hay un icono en forma de chincheta 
- Si hacemos clic sobre ella, la chincheta apunta a un lado 

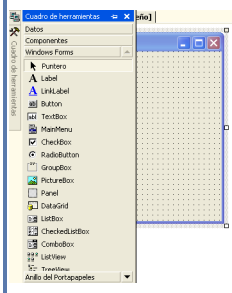
Organizando esos elementos

- Cuando la chincheta apunta a un lado, quiere decir que las ventanas laterales, se esconden a un lado de la pantalla 
- Para ello, sólo debemos hacer clic en el Diseñador de formularios. Háganlo con el cuadro de herramientas.

Aparece una ventana así



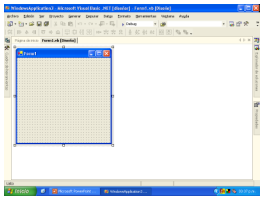
Aparece de nuevo el cuadro de herramientas



- Si queremos ocultarlo de nuevo, sólo hay que hacer clic en el Diseñador de formularios. Háganlo.
- De esta manera, el cuadro de herramientas no nos ocupa espacio y lo tenemos disponible cuando lo queremos.
- Hagan lo mismo con el explorador de soluciones y la ventana de propiedades.

Mail: palasi@palasi.com Web: www.palasi.com

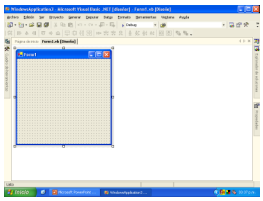
Así obtendremos una ventana totalmente limpia



- Sólo aparecerá el Diseñador de formularios, que será el componente más utilizado.
- Las otras ventanas estarán a un lado y podremos expandirlas cuando lo deseemos.

2. Un primer programa en VB.NET

Por ahora todo parece como en Visual Basic 6



- Hemos entrado al programa. Y tenemos un formulario en blanco donde no hemos hecho nada.
- Veamos el código del formulario, haciendo clic derecho y "ver código", o bien pulsando F7.

Aparece el siguiente código

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Código generado por el Diseñador de Windows Forms
End Class
```

- Esto ya es una diferencia con VB6. En éste último, si no hacíamos nada, no había código. Ahora hay código y no parece muy claro.
- ¿Por qué hay código si no hemos hecho nada?

En VB.NET, cada vez que definimos un formulario con el Diseñador

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Código generado por el Diseñador de Windows Forms
End Class
```

- Se genera el código que define un formulario en VB.NET.
- Esto es diferente de VB6. En él, los formularios se definían gráficamente pero no podían programarse con el mismo lenguaje.
- Ahora tenemos las dos posibilidades.

En VB.NET, hay dos formas de diseñar un formulario

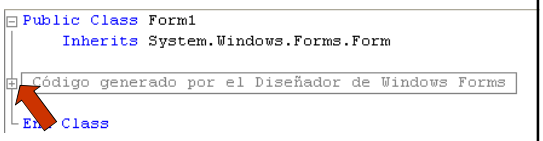
- **1. Programándolo con VB.NET**, ya que se puede. Esta es una forma sólo texto de diseñar el formulario.
- **2. Diseñando gráficamente con el Diseñador de formularios.** Es lo mismo que hacíamos con VB 6.

Mail: palasi@palasi.com Web: www.palasi.com

Las dos formas son equivalentes e intercambiables

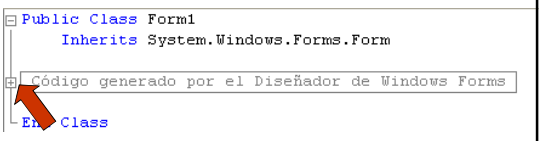
- 1. Si programamos el formulario con VB.NET (sólo texto), después podemos entrar en el Diseñador de formularios y verlo gráficamente.
- 2. Si lo diseñamos gráficamente, se genera automáticamente el código de programación en VB.NET.

En nuestro caso



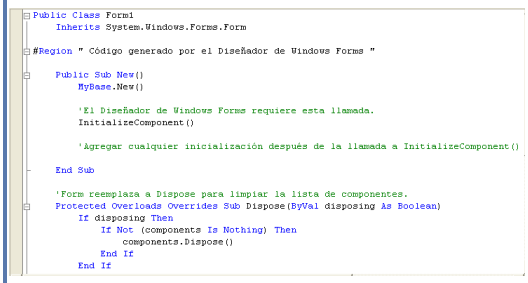
- En nuestro caso, en el Diseñador de formularios tenemos un formulario en blanco y éste es el código en Visual Basic .NET, que lo define.

De hecho, esto no es del todo cierto



- Hagan clic en el signo “más” (+) que señala la flecha y se llevarán una sorpresa.

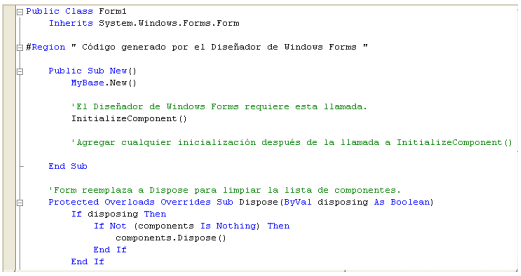
Aparece aún más código



```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Código generado por el Diseñador de Windows Forms "

    Public Sub New()
        MyBase.New()

        'El Diseñador de Windows Forms requiere esta llamada.
        InitializeComponent()

        'Agregar cualquier inicialización después de la llamada a InitializeComponent()

    End Sub

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
```

- ¿Qué es esto? ¿Por qué ha cambiado el código?

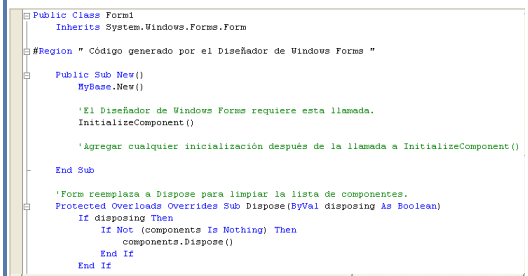
El código que aparece es la forma de definir el formulario en blanco



```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Código generado por el Diseñador de Windows Forms "

    Public Sub New()
        MyBase.New()

        'El Diseñador de Windows Forms requiere esta llamada.
        InitializeComponent()

        'Agregar cualquier inicialización después de la llamada a InitializeComponent()

    End Sub

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
```

- O sea que no es tan sencillo como parecía.

¿Por qué antes aparecía un código sencillo y ahora aparece mucho?



```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Código generado por el Diseñador de Windows Forms "

    Public Sub New()
        MyBase.New()

        'El Diseñador de Windows Forms requiere esta llamada.
        InitializeComponent()

        'Agregar cualquier inicialización después de la llamada a InitializeComponent()

    End Sub

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
```

- ¿Hay dos códigos diferentes? ¿Qué pasa?

Mail: palasi@palasi.com Web: www.palasi.com

La palabra clave #Region

- Es nueva en Visual Basic .NET. Sirve para definir bloques de código que se pueden ver con poco detalle o mucho detalle, según desee el programador.
- Su sintaxis es

```
#Region "FraseResumen"
  <aquí todo el código que queramos>
#End Region
```

- Todo lo que hay entre **#Region** y **#End Region** se puede llamar región (de código).

¿Cómo funciona una región?

```
#Region "FraseResumen"
  <aquí todo el código que queramos>
#End Region
```

- La región puede verse de dos maneras:
  - Toda la región entera. Así se ve con todo detalle todo el código. Se dice que la región está expandida.
  - Sólo su frase resumen. Así sólo vemos una descripción general de toda la región, sin entrar en detalles. Se dice que la región está contraída.

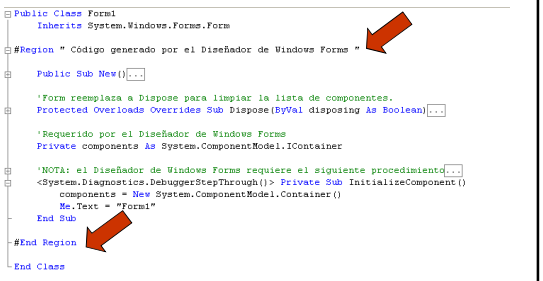
El usuario puede decidir el nivel de detalle

```
#Region "FraseResumen"
  <aquí todo el código que queramos>
#End Region
```

- La región puede verse de dos maneras:
  - Si expande la región, verá todo el código con todo el detalle. Esto es conveniente si está programando este fragmento de código.
  - Si contrae la región, verá sólo su frase resumen. Esto es conveniente si sólo da un vistazo general al código, sin entrar en detalles.

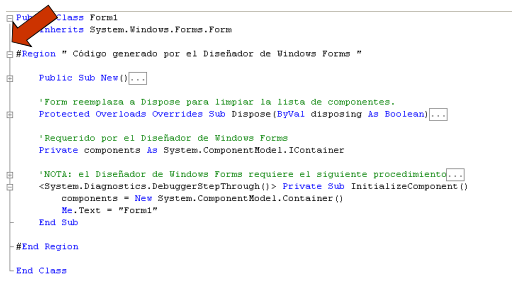
En nuestro ejemplo

- Hay una región que abarca casi todo el código. Su frase resumen es "Código generado por el Diseñador de Windows Forms".



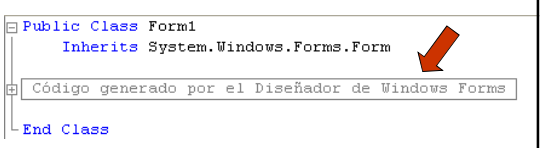
Si hacemos clic en el signo menos que hay a la izquierda del #Region

- Toda la región se contraerá y sólo aparecerá su frase resumen.



Aparecerá el siguiente texto

- Toda la región se contraerá y sólo aparecerá su frase resumen.



- De esta forma, podemos ver toda la región de una manera general, sin entrar en los detalles.
- Esto es lo único que necesitamos por ahora.

Mail: palasi@palasi.com Web: www.palasi.com

Si la región está contraída

- Podemos hacer clic en el signo + al lado de su frase resumen.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Código generado por el Diseñador de Windows Forms
End Class
```

- La región se expandirá.

El texto se expandirá

- Esto nos servirá para ver todos los detalles. Por ahora, no nos interesan

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Código generado por el Diseñador de Windows Forms "

    Public Sub New()...

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)...

    'Requerido por el Diseñador de Windows Forms
    Private components As System.ComponentModel.IContainer

    'NOTA: el Diseñador de Windows Forms requiere el siguiente procedimiento...
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        components = New System.ComponentModel.Container()
        Me.Text = "Form1"
    End Sub

#End Region

End Class
```

Nosotros podemos definir nuestras propias regiones

- Simplemente usando la sintaxis que acabamos de ver.

```
#Region "FraseResumen"
  <aquí todo el código que queramos>
#End Region
```

- Podemos decidir qué fragmento de código queremos incluir en la región y cuál será la frase resumen.

Ejercicio

- Definan una región que comprenda todo el código que aparece y cuya frase resumen sea “Todo el código del formulario”.

No sólo las regiones pueden expandirse o contraerse

- Si miramos el código del formulario, veremos que las clases también pueden expandirse y contraerse.

```
Public Class Form1...
```

- Los métodos (procedimientos y funciones), también pueden hacerlo.

```
Public Sub New()...

'Form reemplaza a Dispose para limpiar la lista de componentes.
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)...
```

Siempre se sigue la misma regla

- Si el bloque de código tiene un signo menos (-) a la izquierda es que está expandido.

```
Public Sub New()
    MyBase.New()

    'El Diseñador de Windows Forms requiere esta llamada.
    InitializeComponent()

    'Agregar cualquier inicialización después de la llamada a InitializeComponent()

End Sub
```

- Si el bloque de código tiene un signo más (+) a la izquierda es que está contraído.

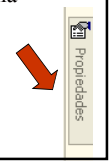
```
Public Sub New()...

'Form reemplaza a Dispose para limpiar la lista de componentes.
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)...
```

¿Para qué todas estas posibilidades de expandir y contraer?

- Esto es una facilidad que nos da el IDE de VB.NET para manejar grandes fragmentos de código.
- El problema de los grandes fragmentos de código es que muchas veces "los árboles no dejan ver el bosque". La abundancia de detalles no deja ver la estructura general.
- Expandiendo y contrayendo podemos ver el código al nivel de detalle que deseemos. Podemos prescindir de los detalles que no nos interesan en ese momento.
- Es decir, se obtiene una mejor comprensión del código.

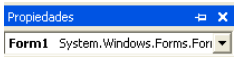
Cambiar el título del formulario (1)

- Cambien el título del formulario desde el Diseñador de formularios. Para ello, deberán seguir los siguientes pasos:
- 1. Vuelvan al diseñador de formularios con el menú **Ver|Diseñador**
- 2. Hagan clic en el icono de la parte derecha que pone "Propiedades". De esta manera, aparecerá la ventana de propiedades.

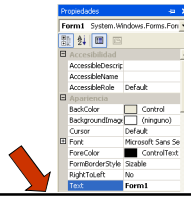


Cambiar el título del formulario (2)

- 3. En la ventana de propiedades, fíjense que éstas son las propiedades del formulario (Form1).

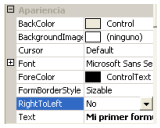


- 4. Vayan hasta la propiedad "**Text**", que es la que contiene el título de un formulario.



Cambiar el título del formulario (3)

- 5. Cambien el valor de la propiedad "Text", por "Mi primer formulario".



- 6. Pulsen ENTER. El valor de las propiedades no queda fijado hasta que pulsan ENTER o se mueven a otra propiedad.

Cambiar el título del formulario (4)

- 7. Verán que el formulario en pantalla ha cambiado de nombre.

Veamos el código generado

- Expandan todo el código y vayan hasta el final.

```
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.S(292, 266)
Me.Name = "Form1"
Me.Text = "Mi primer formulario"
```

- Verán la línea siguiente:

```
Me.Text = "Mi primer formulario"
```

- Esta línea no aparecía antes. ¿Por qué aparece ahora?

Mail: palasi@palasi.com Web: www.palasi.com

Recordemos: dos formas de diseñar un formulario

- 1. Programándolo en código VB.NET (textualmente).
- 2. Diseñándolo con el Diseñador de formularios (gráficamente).

- Las dos formas son intercambiables.
  - Si se programa con código VB.NET, automáticamente los cambios se reflejan en el Diseñador.
  - Si se programa con el Diseñador, automáticamente los cambios se reflejan en el código VB:NET

En nuestro caso

- Hemos cambiado el nombre del formulario con el Diseñador y esto se refleja en el código.

```
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(292, 266)
Me.Name = "Form1"
Me.Text = "Mi primer formulario"
```

- Vamos a realizar la operación contraria. En la línea que pone

```
Me.Text = "Mi primer formulario"
```

cámbienla para que diga

```
Me.Text = "Formulario en VB.NET"
```

Abran el Diseñador de formularios

- Verán que ahora el título es "Formulario en VB.NET". Se ha actualizado automáticamente con los cambios que hemos hecho en el código.

¿Usar el Diseñador, programar el código o ambos?

- No importa el método que usemos para diseñar el formulario, mientras lo hagamos bien.
- Siempre los cambios hechos en el Diseñador se reflejarán en el código y viceversa.
- Sin embargo, ahora estamos aprendiendo. Por momento, sólo usaremos el Diseñador, ya que, para diseñar el formulario en código, aún no tenemos los conceptos necesarios para hacerlo.

2. Un primer programa en VB.NET

- 2.1. El IDE de Visual Basic.NET
- 2.2. Código de un formulario.
- 2.3. Procedimientos y eventos.
- 2.4. Compilación y ejecución Archivos PE.
- 2.5. Archivos fuente de un programa.

Vamos a crear un botón de comando

- Un botón de comando es un control gráfico que, al hacer clic sobre él, realiza una determinada acción.
- Esto es lo mismo que en VB6 o otros lenguajes.

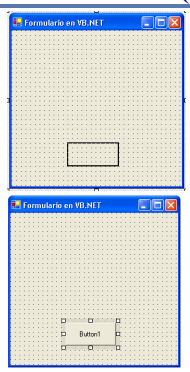
Pasos para crear un botón de comando (1)

- 1. Expandan el cuadro de herramientas, haciendo clic en su icono en la parte izquierda.

- 2. Dentro de la barra de herramientas hagan clic en el icono "Button". Después contraigan el cuadro de herramientas.

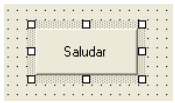
Pasos para crear un botón de comando (2)

- 3. Hagan clic en la parte del formulario que quieran colocar el botón y arrastren hasta que dibujar un cuadrado con las medidas del mismo.
- 4. Suelten el botón del mouse. Aparecerá el botón.



Cambien el texto dentro del botón de comando

- Para ello, expandan la ventana de propiedades y cambien el valor de la propiedad "Text" para que sea "Saludar". El botón deberá tener el texto "Saludar".



Fíjense que el nombre del botón es "Button1"

- Esto se ve en la ventana de propiedades, en la propiedad (Name).

- "Saludar" es sólo el texto, no es el nombre.

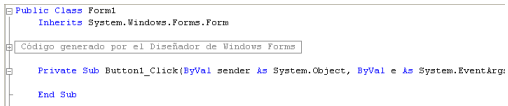
Abran el código y expandan todo

- Verán que se ha añadido un código para el botón. Fíjense que el texto "Saludar", aparece en ese código.

```
'
'Button1
'
Me.Button1.Location = New System.Drawing.Point(96, 192)
Me.Button1.Name = "Button1"
Me.Button1.Size = New System.Drawing.Size(88, 40)
Me.Button1.TabIndex = 0
Me.Button1.Text = "Saludar"
'
'Form1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
```

Hagan doble clic en el botón

- Con el código contraído, deben obtener:



- Fíjense que ha aparecido una estructura del estilo.

```
Private Sub Button1_Click(...)

End Sub
```


Mail: palasi@palasi.com Web: www.palasi.com

Recordando

- Esta estructura que ha aparecido:

```
Private Sub Button1_Click(...)

End Sub
```

- Debería ser familiar por VB6, pero la explicaremos para recordar un poco.

Procedimiento

- Fragmento de código que
  - agrupamos y al que ponemos un nombre
  - para que pueda ser ejecutado sin tener que repetir todas sus instrucciones.
- Existen en VB6 y en muchos otros lenguajes.

Sintaxis de un procedimiento en VB6

- Su sintaxis en VB6 es:

```
Sub NombreProcedimiento (ListaParametros)
   FragmentoDeCodigo
End Sub
```

- ***FragmentoDeCodigo*** es el conjunto de instrucciones que agrupamos en el procedimiento.
- ***ListaParametros*** es una lista de parámetros separada por comas. Un parámetro es un dato que se le envía al procedimiento desde el programa principal.
- A veces el procedimiento lleva **`Private`** o **`Public`** antes del primer **`Sub`**. Esto no nos importa por ahora.

La lista de parámetros en VB6

- Tiene la siguiente sintaxis:

```
NombreParam1 As TipoParam1, . . . ,
NombreParamn As TipoParamn
```

- Es decir, para cada parámetro se indica su nombre y su tipo, separados por la palabra **`As`**.
- A veces, antes del nombre del parámetro aparece la palabra **`ByVal`**. Esto no nos importa por ahora.

Advertencia

- Todo lo que hemos visto hasta ahora son los procedimientos en VB6.
- **En VB.NET ya no hay procedimientos, sino algo llamado métodos**. El concepto es algo diferente.
- Sin embargo, estas diferencias las veremos más adelante. Por ahora podemos suponer que son procedimientos como los de VB6.

De hecho, el código que hemos visto que se ha generado

```
Private Sub Button1_Click(ByVal
  sender As System.Object, ByVal e
  As System.EventArgs) Handles
  Button1.Click

End Sub
```

- Como vemos es un procedimiento. Tiene el **`Sub`** o el **`End Sub`**, la lista de parámetros con sus tipos.
- Lo que no tiene el fragmento de código: está vacío, por ahora.

Eventos gráficos en VB6

- Acciones que el usuario realiza con la interfaz gráfica y que producen una reacción del programa.
- Por ejemplo, hacer clic en un botón, escribir algo en un cuadro de texto, desplazar una ventana, cerrar una ventana, etc.
- Todo botón tiene un evento, llamado "Click", que es el que se produce cuando el usuario hace clic en ese botón.

¿Qué pasa cuando el usuario hace clic en un evento?

- Depende del evento. En un programa, cuando se hace clic en un botón, algunos botones hacen unas cosas y otros botones hacen otras.
- El programador debe especificar qué es lo que debe pasar cuando se produce un evento.
- ¿Cómo lo especifica?

Cada evento tiene un procedimiento asociado

- En ese procedimiento es donde el programador escribe el código que se ejecuta cada vez que se produce el evento.
- Así, si tenemos un botón, en VB6, el procedimiento asociado al evento "Click", se llama ***nombrebotón*_Click**
- Será en ese procedimiento donde escribimos el código que se ejecutará cada vez que se haga clic en el botón.

En VB.NET es un poco diferente

- En VB6, el procedimiento asociado al evento "Click" de un botón, se llamaba ***nombrebotón*_Click**
- En VB.NET el nombre es lo de menos. Lo importante es que después de la lista de parámetros aparezca

```
Handles nombrebotón.Click
```

De hecho, el código que hemos visto que se ha generado

```
Private Sub Button1_Click(ByVal
  sender As System.Object, ByVal e
  As System.EventArgs) Handles
  Button1.Click

End Sub
```

- Es el procedimiento asociado al evento de hacer clic en el botón **Button1** (que acabamos de crear).
- Es decir, cuando se hace clic en el botón, se ejecutará el código que hay en el procedimiento.
- Como no hay código, no se ejecuta nada.

Esto va a cambiar

- Queremos que cuando hagamos clic en el botón "**Button1**" aparezca un mensaje de saludo.

Mail: palasi@palasi.com Web: www.palasi.com

Para ello, usaremos el "procedimiento" `MessageBox.Show`

- En su versión más sencilla, escribir
  `MessageBox.Show("texto")`
  muestra una pantalla con el texto que hay entre las comillas.
- Fíjense que la versión de este "procedimiento" en VB6 es **`MsgBox().`**
- **`MsgBox()`** sigue existiendo en VB.NET, pero se recomienda no usarla porque está "deprecada" (es decir, puede desaparecer en futuras versiones del lenguaje).

Ejercicio

- Escribir el código para que cada vez que se haga clic sobre el botón con el texto "Saludar", aparezca una ventana con texto "Hola a todos".

Solución

- Basta con escribir un **`MessageBox`** en el procedimiento que se ejecuta cuando se hace clic en el botón.

```
Private Sub Button1_Click(...)
 Handles Button1.Click
   MessageBox.Show("Hola a todos")
End Sub
```

Nota

- Si queremos poner un título a la ventana del **`MessageBox`**, lo podemos incluir como segundo parámetro del **`Show`**.

```
Private Sub Button1_Click(...)
 Handles Button1.Click
   MessageBox.Show("Hola a todos",
     "Mi primer Messagebox")
End Sub
```

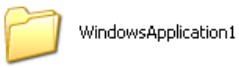
2. Un primer programa en VB.NET

Ya tenemos un programa "importante"

- Queremos ejecutarlo. Para ello usaremos **Depurar|Iniciar** o bien F5.

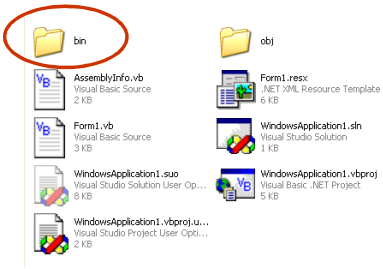
Queremos ejecutarlo fuera del IDE

- Al ejecutarlo dentro del IDE con F5, se ha generado automáticamente el programa compilado.
- Para verlo, abran la carpeta donde VB.NET guarda sus proyectos. Por defecto, se encuentra en **`Mis documentos\Proyectos de Visual Studio`**
- Allá verán una carpeta con el nombre de su proyecto. En nuestro caso, **`WindowsApplication1`**



Abramos esta carpeta

- Dentro hay una serie de archivos y un directorio **`bin`**. Abramos el directorio **`bin`**.



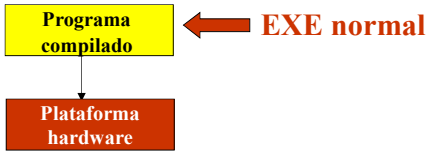
Dentro del directorio `bin`

- Hay un archivo llamado **`WindowsApplication1.exe`**.
- Este es el programa compilado.
- Para ejecutarlo hagan doble clic sobre él.

Las apariencias engañan

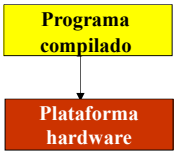
- Esto parece un ejecutable normal de Windows
  - Tiene la extensión .EXE.
  - Se ejecuta al hacer doble clic.
- Pero no lo es. Es algo muy diferente.

Un ejecutable de Windows está escrito en lenguaje máquina



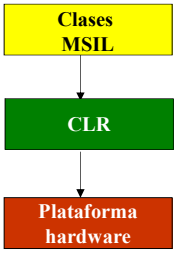
- En nuestro caso, en un código máquina de los procesadores de la gama Pentium.
- Para ejecutarlo, el sistema operativo sólo lo carga en memoria y le pasa el control.
- Es la plataforma hardware que lo ejecuta.

Sin embargo el EXE que hemos generado



- En nuestro caso, en un código máquina de los procesadores de la gama Pentium.
- Para ejecutarlo, el sistema operativo sólo lo carga en memoria y le pasa el control.
- Es la plataforma hardware que lo ejecuta.

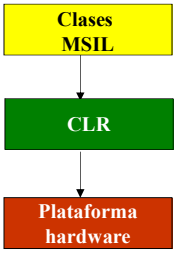
Sin embargo el EXE que hemos acabado de generar con VB.NET



EXE generado

- No está escrito en código máquina y, por lo tanto, no es ejecutable directamente sobre la plataforma.
- Está escrito en el lenguaje MSIL y, por lo tanto, lo ejecuta el CLR, con todas las ventajas que esto supone.

Para diferenciarlo, de los EXEs normales, le daremos un nuevo nombre



Archivo PE

- Le llamaremos archivos PE ("portable executable").
- El archivo PE es un archivo .exe, formado por clases escritas en lenguaje MSIL que es ejecutado sobre el CLR.

¿De verdad pasa esto?

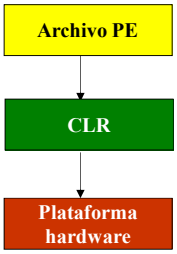
- La verdad es que resulta difícil de creer, pues hacemos doble clic y el programa se ejecuta.
- Esto es porque todo el proceso de ejecución de un archivo PE es transparente y se hace silenciosamente. Por ello no se ven las diferencias.
- Vamos a verlo para tener claros los conceptos.

Proceso de ejecución de un archivo PE

- 1. Se hace doble clic sobre nuestro archivo PE.

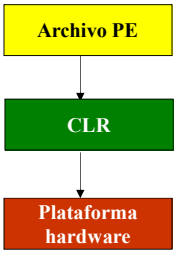
- 2. El archivo se ejecuta y detecta que él mismo no es un EXE normal, sino un archivo PE
  - que está escrito en lenguaje MSIL.
  - que necesita el CLR para ejecutarse.

Proceso de ejecución de un archivo PE



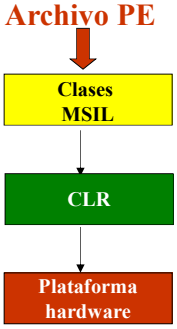
- 3. El archivo PE llama al CLR para que lo ejecute.
- 4. El CLR ejecuta el archivo PE y aparecen las ventanas.

Lo importante es que el archivo PE es ejecutado por el CLR



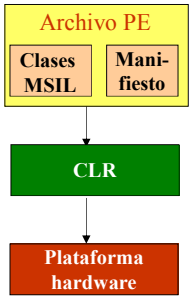
- Aunque esto no se vea directamente, pero es así.
- Más adelante veremos que esto se complica más, pero por ahora es suficiente.

Otra pregunta: ¿De que está compuesto el archivo PE?



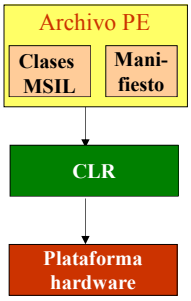
- El archivo PE está compuesto de clases escritas en el lenguaje MSIL.
- En este sentido, el archivo PE se parece a un archivo ZIP: es un archivo que contiene otros archivos dentro de él.

Pero aún hay más



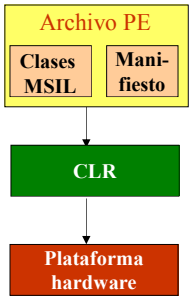
- El archivo PE no sólo está compuesto por clases MSIL, sino que contiene también un archivo especial llamado **"el manifiesto".**
- ¿Qué es esto?

El manifiesto tiene los datos necesarios para que el arc. PE se pueda ejecutar



- El manifiesto tiene información sobre los tipos definidos por el archivo PE, los tipos externos que utiliza, la versión del archivo PE, etc.
- En versiones anteriores de VB, esta información se guardaba en el registro y en librerías de tipos. Esto hacía que pudieran haber conflictos y hacía que el ejecutable fuera poco portable.

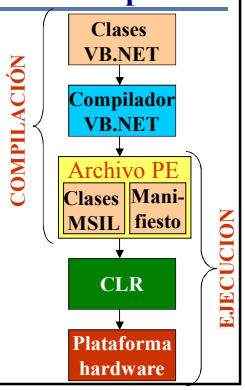
Ahora toda esta información está en el archivo PE



- El ejecutable contiene toda la información necesaria para ejecutarse, sin hacer referencias a archivos externos o al registro.
- Por lo tanto, es completamente portable sin ningún problema.
- Es por eso que se llama PE ("portable executable", un ejecutable que se puede portar).

El manifiesto es generado automáticamente por el compilador

- El compilador lo genera y lo incluye en el archivo PE junto con las clases compiladas de nuestro programa.
- Por lo tanto, no nos debemos preocuparnos de definirlo pues su generación es automática.



Ejercicio

- Creen un programa con un formulario con cuatro botones.
- Al apretar un botón, el formulario se mueve 5 píxels a la derecha.
- Los otros botones hacen lo mismo, pero a la izquierda, arriba y abajo.
- Utilicen Me.Left (que indica la posición X en que empieza el formulario) y Me.Top (que indica la posición Y).
- Ejecuten el programa desde dentro y fuera del IDE.

Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- Creen un formulario con un cuadro de texto y un botón. Cuando se haga clic en el botón, se debe mostrar en un **`MessageBox`** el contenido del cuadro de texto.
- Nota: si el cuadro de texto se llama **`TextBox1`**, el contenido se obtiene con **`Me.TextBox1.Text`**

Ejercicio

- Creen un formulario con dos cuadros de texto y tres botones (llamados "Suma", "Resta" y "Producto").
- En los cuadros de texto se introducirán números y cada vez que se oprima un botón debe dar el resultado de la operación correspondiente.

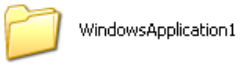
Ejercicio

- Creen un formulario con dos cuadros de texto y un botón.
- Cuando se pulse el botón, un **`MessageBox`** debe mostrar una frase que concatene el contenido los dos cuadros de texto.
- Para concatenar dos cadenas, usen el operador **`&`**

2. Un primer programa en VB.NET

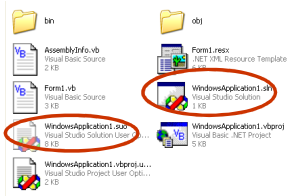
Los archivos fuentes de nuestro proyecto

- Acabaremos el estudio de este primer programa, viendo algunos archivos fuentes que genera VB.NET.
- Abran la carpeta donde VB.NET guarda el proyecto. Por defecto, se encuentra en **`Mis documentos\Proyectos de Visual Studio\WindowsApplication1`**



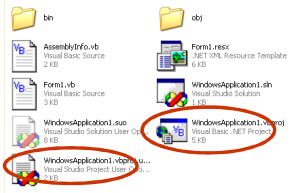
Hay dos archivos que contienen información sobre la solución

- Recordemos que una solución es un grupo de proyectos.
- **`WindowsApplication1.sln`** (texto) tiene información sobre nuestra solución
- **`WindowsApplication1.suo`** (binario) contiene las opciones de personalización del IDE asociadas con la solución, de forma que, cuando volvamos a abrir el IDE, se muestre como lo hemos configurado.



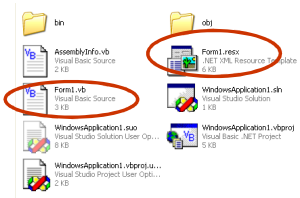
Hay dos archivos que contienen información sobre el proyecto

- **`WindowsApplication1.vbproj`** (XML) tiene información sobre cada proyecto
- **`WindowsApplication1.vbproj.user`** (XML) contiene las opciones de personalización del IDE asociadas con el proyecto, de forma que, cuando volvamos a abrir el IDE, se muestre como lo hemos configurado.



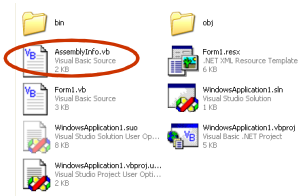
Hay dos archivos que contienen información sobre el proyecto

- **`Form1.vb`** (texto) tiene el código fuente del formulario en VB.NET
- **`Form1.resx`** (XML) contiene los recursos externos que usa el formulario para funcionar (imágenes, datos no ejecutables, etc).



Hay un archivo que no veremos por ahora

- **`AssemblyInfo.vb`** (texto). Lo veremos más adelante cuando estudiemos los ensamblados.

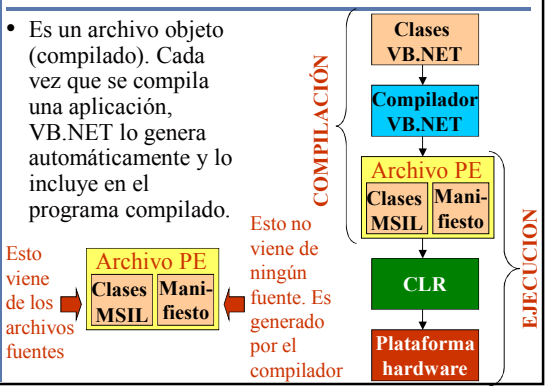


Pregunta

- ¿Dónde está el archivo del manifiesto?

El manifiesto no es un archivo fuente

- Es un archivo objeto (compilado). Cada vez que se compila una aplicación, VB.NET lo genera automáticamente y lo incluye en el programa compilado.



Temario del curso

Mail: palasi@palasi.com Web: www.palasi.com

3. Introducción a la programación orientada a objetos

- 3.1. Qué es la programación orientada a objetos.
- 3.2. El concepto de objeto.
- 3.3. Ciclo de vida de un objeto.
- 3.4. Programación de clases.

3. Introducción a la programación orientada a objetos

- 3.1. Qué es la programación orientada a objetos.
- 3.2. El concepto de objeto.
- 3.3. Ciclo de vida de un objeto.
- 3.4. Programación de clases.

La crisis del software

- Es el nombre que damos al estado insatisfactorio en que se encuentra el desarrollo de software.
- Las tareas que nos gustaría resolver mediante las computadoras son en la práctica:
  - demasiado difíciles de resolver.
  - suelen extenderse más allá del costo y el tiempo previsto.
  - es muy probable que contengan errores.

La crisis del software

- Los proyectos de programación de gran tamaño consumen 150% del tiempo previsto.
- El 25% de estos proyectos son cancelados.
- El 75% de los proyectos no cancelados:
  - O bien no funcionan como se quería.
  - O bien no se utilizan para nada.

Estadísticas sobre proyectos de software

Tamaño	Temprano	A tiempo	Retraso	Cancelados
1 PF	14.68%	83.16%	1.92%	0.25%
10 PF	11.08%	81.25%	5.67%	2.00%
100 PF	6.06%	74.77%	11.83%	7.33%
1000 PF	1.24%	60.76%	17.67%	20.33%
10000 PF	0.14%	28.03%	23.83%	48.00%
100,000PF	0.00 %	13.67%	21.33%	65.00%

- Fuente: Patterns of Software Systems Failure And Success, Capers Jones.

Soluciones a la crisis del software

- La crisis se detectó en la conferencia de la OTAN de 1968 sobre Ingeniería del Software y sigue vigente.
- Se han investigado varias soluciones:
  - Derivación y verificación automática de software (de interés más teórico).
  - Programación con componentes reusables de software.

Programación con componentes reusables de software

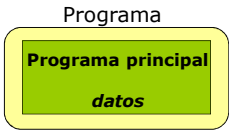
- Una de las técnicas para mitigar los problemas del desarrollo de software.
- Se basa en la idea extendida en otras ingenierías que, para construir un producto, sólo deben ensamblarse piezas preconstruidas.
- La idea básica es:
  - existen componentes reusables
  - programar una aplicación consiste en ensamblarlos.

Programación con componentes reusables de software

- Objetivo perseguido durante muchos años.
- Para conseguirlo se han desarrollado varias técnicas de programación:
  - programación no estructurada.
  - programación procedimental.
  - programación modular.
  - programación orientada a objetos.

Programación no estructurada

- Es el primer tipo de programación que apareció.
- Hay un único programa principal que trabaja con unos datos globales.

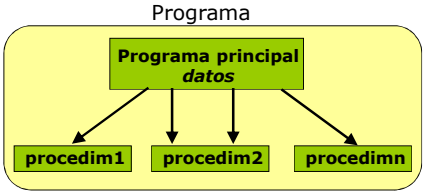


Características de la programación no estructurada

- Hace muy difícil la programación a gran escala:
  - no hay ninguna reusabilidad del código.
  - si hay que realizar una tarea dos veces, se debe volver a escribir el código.
- Prácticamente no se utiliza en el mundo real.

Programación procedimental

- El código de tareas que se repiten se escribe en procedimientos ("subprogramas"). El programa principal los llama cada vez que debe hacer dichas tareas.



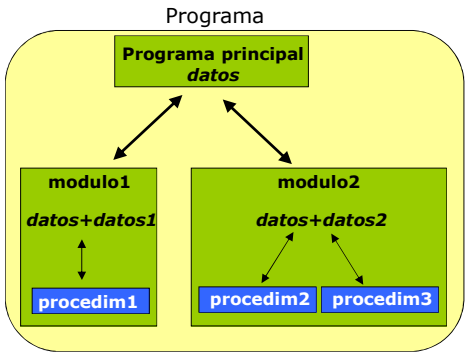
Características de la programación procedimental

- Reusabilidad a nivel elemental:
  - se reusa el código de los procedimientos
  - se evita la tarea de repetir código dentro de un programa
- El problema es que los procedimientos no pueden ser reusados por otros programas.

Programación modular

- Los procedimientos referentes a un mismo conjunto de datos se agrupan en un módulo (compilado separadamente).
- Cada programa consta de varios módulos. El programa principal coordina las llamadas a los procedimientos de los otros módulos.

Programación modular



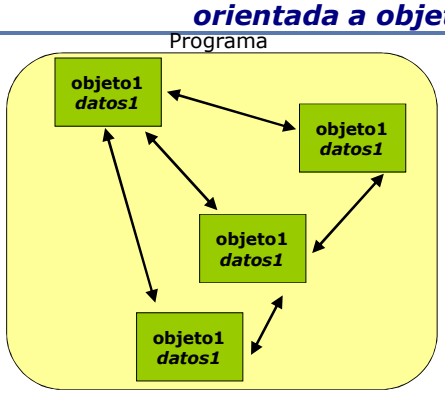
Características de la programación modular

- Hay una reusabilidad de los módulos.
- De hecho, los módulos tienen sus propios datos y se convierten en parecidos a un tipo de datos con sus operaciones (ejemplo: lista).
- Pero hay inconvenientes:
  - los módulos no se acaban de comportar como un tipo de datos.
  - se vuelve complicado tener y gestionar varias instancias del módulo (varias listas).

Programación orientada a objetos

- Resuelve los problemas mencionados.
- Desaparece el concepto de programa principal.
- Lo que hay es una red de ***objetos*** interactuantes.
- Los objetos
  - contienen datos y operaciones sobre estos datos.
  - interactúan intercambiando mensajes (análogo a llamadas a procedimiento).

Programación orientada a objetos



3. Introducción a la programación orientada a objetos

Mail: palasi@palasi.com Web: www.palasi.com

Programación orientada a objetos

- Es una de las técnicas para implementar programación con componentes reusables de software.
- Estándar dominante actualmente de programación.
- Evoluciona de otras formas anteriores de programación, pero introduce una manera diferente de pensar (más parecida a la vida real).
- Se basa en el concepto de objeto.

Objeto (de software)

- Concepto fundamental de la programación orientada a objetos.
- Es análogo a un objeto de la vida real.

Un objeto de software es análogo a un objeto de la vida real

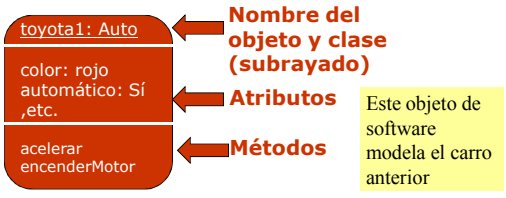
- Por ejemplo, un objeto real es el auto que es propiedad del licenciado.
- De un objeto real, nos interesan dos cosas.

Las dos cosas que nos interesan de un objeto

- Sus características (su color, tamaño, su potencia, si tiene caja de cambios automática, etc.). A esto se le llama **atributos o características.**
- El tipo de acciones que pueden realizarse con él (acelerar, encender el motor, apagarlo, etc). A esto se le llama **métodos**.
- A atributos y métodos, se les llama **miembros**

Un objeto de software es parecido

- **Es un conjunto de datos relacionados que un programa trata como una unidad y que modela un objeto real.**
- Nos interesan sus atributos o características (datos) y las cosas que pueden hacerse con esos datos: sus métodos.



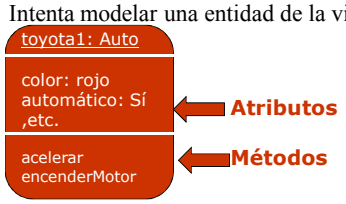
Ejemplo

- Supongamos que estamos creando un programa para gestionar el parqueo de los diferentes carros de los empleados de una empresa.
- Hay una serie de objetos reales, que son el auto del licenciado, el auto de la empresa, etc.
- Nuestro programa puede tener muchos objetos carroDelLicenciado, carroDeLaEmpresa. Cada uno modela a un auto real.
- Cada uno de estos objetos de software tendrá unos atributos (color, modelo) y unos métodos (acelerar, frenar) que se corresponderán con los atributos y métodos de los objetos reales.

Repasando: Objeto

- Concepto principal de la programación O-O.
- Contiene:
  - datos (atributos).
  - métodos para efectuar tareas sobre esos datos.

  A esto se le llama encapsulación.
- Intenta modelar una entidad de la vida real.



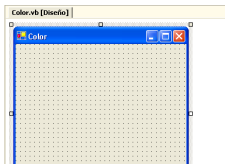
3. Introducción a la programación orientada a objetos

- 3.1. Qué es la programación orientada a objetos.
- 3.2. El concepto de objeto.
- 3.3. Ciclo de vida de un objeto.
- 3.4. Programación de clases.

Vamos a ver el ciclo de vida de un objeto

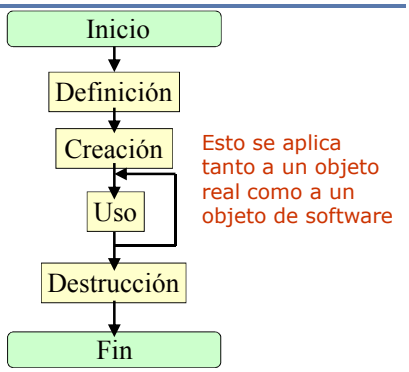
- Para ello, nos vamos a ayudar de un proyecto de VB.NET previamente construido que nos ayudará a aprender los conceptos de forma gráfica.
- Ese proyecto se encuentra en una carpeta llamada “Clases Coloreadas”, que se les provee.
- Copien esa carpeta a su carpeta de Proyectos de Visual Studio y ábranla con **Abrir|Proyecto**

Aparecerá un proyecto

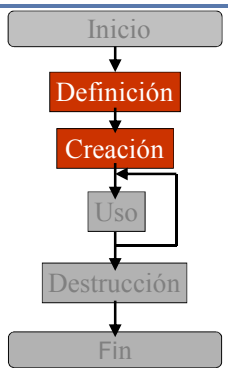


- Pueden abrir el formulario “Color”. Por ahora, este es el único componente del proyecto que usaremos.
- Pueden ejecutarlo y verán que no hace nada.

Ciclo de vida de un objeto



Ciclo de vida de un objeto



palasi.com
IT and management

Definición y creación de un objeto

- La definición y creación de un objeto se realiza en base a partir de una clase previamente definida.
- Esto nos obliga, antes de continuar, a definir con exactitud el concepto de clase.

Clases y objetos

- Un programa utiliza una serie de objetos y normalmente muchos de ellos son similares.
- Al conjunto de estos objetos similares se le llama **clase**

Otra definición de clase

- La descripción o modelo (o molde) de estos objetos similares se le llama **clase**
- Esta definición es equivalente a la anterior.

Una clase es un documento de instrucciones de fabricación

- Una clase es una descripción o modelo de diseño de todos los objetos que forman parte de ella. Así, "Toyota Corola" es una especificación para crear autos de una cierta marca o modelo.

Clases y objetos

- La clase indica cómo serán los objetos que se crearán con ese modelo
- Una clase sería un documento que nos describiera cómo es un "Toyota Corola"
- Un objeto sería el auto "Toy.Corola" de Juan

Resumen: Tres formas de ver lo mismo

- Una clase es el conjunto de todos los objetos del mismo tipo.
- Una clase es un "molde" para hacer objetos del mismo tipo.
- Una clase son las instrucciones para crear objetos de este tipo.

Mail: palasi@palasi.com Web: www.palasi.com

Un poco de terminología

- Cuando un objeto pertenece a una clase, se dice que es una **instancia** de esta clase.
- Así, los objetos autoDePerez, autoDeBonilla, autoDeAlvarado son instancias de la clase Auto.
- Decimos que instanciamos una clase cuando creamos un objeto de la misma.

Un hecho incuestionable

- En VB.NET (como en otros lenguajes O-O) todos los objetos pertenecen a una clase.
- No hay objetos sin clase.
- Por eso, cada vez que creamos un objeto debemos especificar a qué clase pertenece.

Un hecho importante

- **Las clases son tipos de datos.**
- En VB.NET todas las clases son tipos de datos.
- Los objetos de una clase son los valores del tipo de datos que define la clase.

Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa especificación) que es la clase.
- Veamos cómo se hace en VB.NET

¿Cómo se codifica esto?

- Se necesitan dos pasos: definición de una variable y creación del objeto.
- 1. Se declara una variable de la clase que queremos instanciar (definición).

```
Dim objRojo1 As ClaseRoja
```

- 2.- Se crea un nuevo objeto y su referencia se almacena en esta variable.

```
objRojo1 = New ClaseRoja()
```

1. Definición

- Necesitamos declarar una variable para guardar el objeto que creemos.
- En los lenguajes convencionales, hay que declarar una variable para guardar el valor de un tipo de datos.
- Como hemos dicho que un objeto no es el valor de un tipo de datos, necesitaremos una variable para guardarlo.
- Esta variable será del tipo de datos de la clase.

```
Dim objRojo1 As ClaseRoja
```

Declaración de variables

- En VB.NET se puede hacer con diferentes sintaxis.
- Por ahora, sólo veremos la más sencilla. Para declarar una variable ***var*** de tipo ***tip*** se usa la siguiente sintaxis

```
Dim var As tip
```

- En nuestro caso, la variable es de tipo **`ClaseRoja`** y se llama **`objRojo1`**. Por ello, la declaración es:

```
Dim objRojo1 As ClaseRoja
```

2. Creación

- Una vez declarada la variable, la creación consta de dos tareas.
  - Crear el objeto.
  - Asignarlo a la variable.
- Esto se realiza como una única instrucción.

```
var = New tip()
```

  - **`New`** ***tip()*** crea un objeto nuevo del tipo (clase) ***tip*** (no olviden los paréntesis).
  - La asignación, ***var*** =, asigna este nuevo objeto a la variable.

2. Creación

- En nuestro caso, creamos un nuevo objeto de la clase **`ClaseRoja`** y lo asignamos a la variable **`objRojo1`**.
- Esto se hace así .

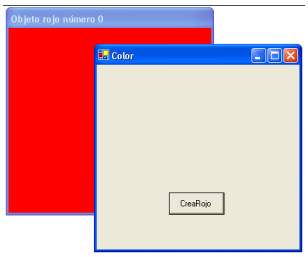
```
Dim objRojo1 As ClaseRoja
```

Ejercicio

- Creen un botón sobre el formulario "Color". Programen de forma que, cuando se haga clic sobre ese botón, se cree un objeto de la clase "ClaseRoja".
- Ejecuten el programa y creen un objeto de esta clase.

Verán que cuando se crea un objeto de la clase ClaseRoja

- Aparece un formulario rojo en pantalla.

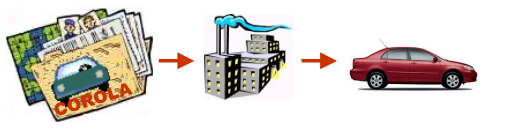


Qué hemos hecho

- La clase **`ClaseRoja`** era un diseño o especificación para crear objetos de esa clase. Lo que se ha hecho es crear un objeto a partir de la clase.

New ClaseRoja()

CLASE ClaseRoja → **OBJETO objRojo1**



Mail: palasi@palasi.com Web: www.palasi.com

El formulario rojo es la representación gráfica del objeto

- Hemos programado que cada vez que se cree un objeto aparezca un formulario en la pantalla.
- Esto no es así por defecto, sino que lo hemos programado así para que entiendan bien el concepto de objeto.
- Por ahora no piensen cómo está programado, sólo piensen que el formulario es una representación gráfica de objeto, **pero no es el objeto mismo.**

Esto no es una pipa (Magritte)

Esto no es un objeto

Es la representación gráfica de un objeto

El formulario es la representación gráfica de un objeto

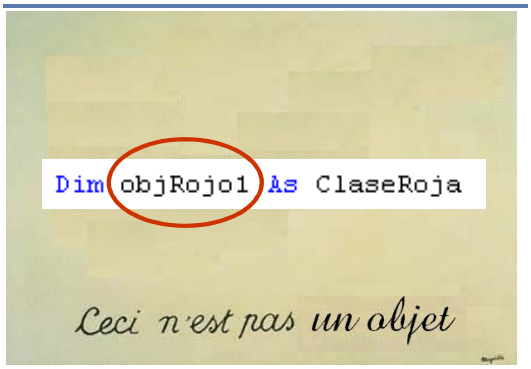
- Igual que un icono es la representación gráfica de un archivo, pero no el archivo mismo.
- Un icono permite manipular un archivo pero no es el archivo mismo.

Atención

```
objRojo1 = New ClaseRoja()
```

- En la variable **objRojo1** no se almacena el objeto sino una **referencia** al objeto.
- La referencia representa al objeto y permite que lo manipulemos pero NO es el objeto.
- Es similar al icono de un archivo.

Es decir, pasa lo mismo que antes

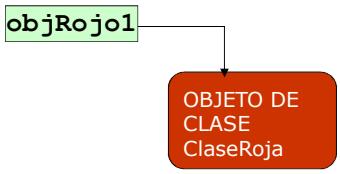


La variable no es un objeto

- La variable es una representación, un icono del objeto.
- Nos permite manipular y utilizar un objeto, pero no es el objeto mismo.
- Se dice que la variable es una **referencia** del objeto. O bien, una referencia que **apunta** al objeto.
- En términos de lenguaje máquina, sería la dirección del objeto.

Referencia al objeto

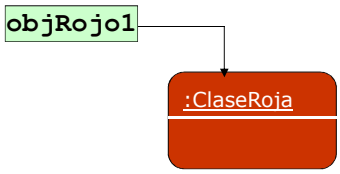
```
objRojo1 = New ClaseRoja()
```



- La variable es una referencia. Permite manipular el objeto, pero no es el objeto mismo.

Para abreviar

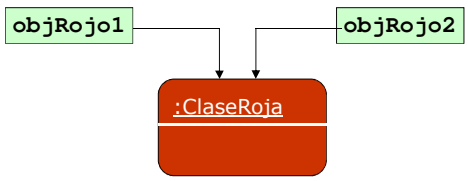
```
objRojo1 = New ClaseRoja()
```



- :ClaseRoja quiere decir en UML, un objeto de la Clase "ClaseRoja"

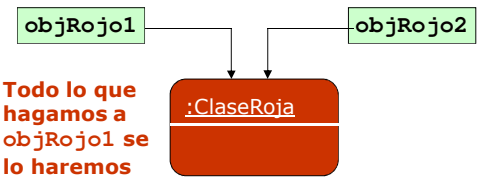
Varias referencias pueden referenciar el mismo objeto

- `objRojo1 = New ClaseRoja()`
- `objRojo2 = objRojo1`



No se ha hecho dos objetos sino dos referencias a él

```
objRojo1 = New ClaseRoja()
objRojo2 = objRojo1
```



Todo lo que hagamos a `objRojo1` se lo haremos también a `objRojo2`

Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- En el programa anterior, incluyan el siguiente código en el botón.

```
objRojo1 = New ClaseRoja()
objRojo2 = objRojo1
```

- Recuerden que tienen que declarar las variables.
- ¿Qué es lo que obtienen al hacer UN SOLO CLIC?

Sólo aparece un objeto

- Y dos referencias (variables) que apuntan a él (esto no se ve).
- No aparecen dos objetos diferentes.

Pregunta

- ¿Cómo haríamos que **objRojo1** y **objRojo2** referenciaran dos objetos diferentes?

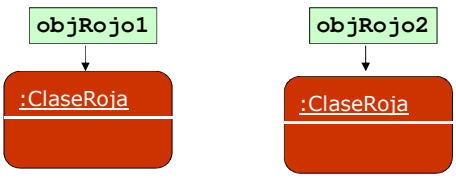
Respuesta

- Invocando dos veces la creación con **New:**

```
objRojo1 = New ClaseRoja()
objRojo2 = New ClaseRoja()
```

Respuesta

```
objRojo1 = New ClaseRoja()
objRojo2 = New ClaseRoja()
```

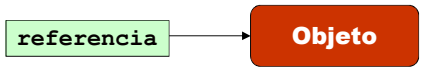


Ejercicio

- Comprueben esta solución, programándola en el ejercicio anterior.

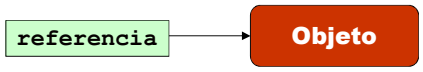
Resumiendo

- En VB.NET hay dos cosas completamente diferentes:
- **Objetos**. Grupo de datos que el programa trata como una unidad.
- **Referencias a objetos (variables).** La forma con que accedemos a los objetos.



Tanto las referencias como los objetos tienen clase

- **Objetos**. La clase del objeto es la que se escribe detrás de la palabra **New**. Se le llama clase de creación.
- **Referencias a objetos (variables).** La clase de una variable es la clase con la cual se declara la variable.



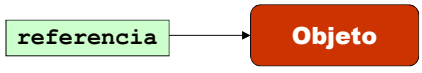
1 variable y el objeto que refer. deben ser de la misma clase

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
```
OK

referencia de clase ClaseRoja.
objeto de clase ClaseRoja.

```
Dim objRojo1 As ClaseAzul
objRojo1 = New ClaseRoja()
```
NO

referencia de clase ClaseAzul.
objeto de clase ClaseRoja.



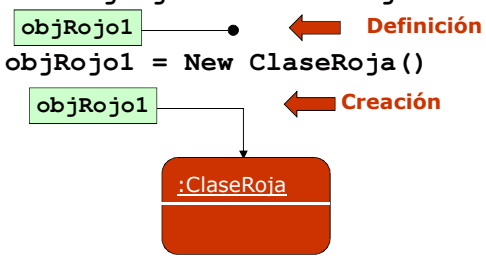
Repasando

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
```
Definición
Creación

- A la creación se le llama a veces instanciación.
- Se dice que instanciamos la clase (ya que creamos una instancia –objeto- de ella).
- A veces, abusamos del lenguaje y decimos que instanciamos el objeto.

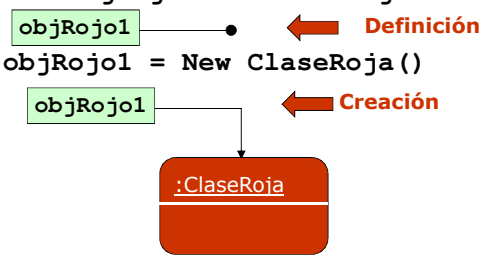
Definición y creación de un objeto

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
```



Pregunta: ¿Qué referencia objRojo1 después de definición?

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
```



Mail: palasi@palasi.com Web: www.palasi.com

Respuesta: No referencia nada

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
```

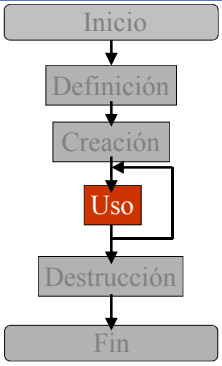
objRojo1 ← Definición

Se dice que referencia a `Nothing`

Ejercicio

- Crear dos objetos de la clase **`ClaseAzul`**, uno de la clase **`ClaseVerde`** y uno de la clase **`ClaseRoja`**.

Ciclo de vida de un objeto



Las dos cosas que nos interesan de un objeto (ejemplo, de clase ClaseRoja).

- Sus características o **atributos.**
  - Su posición vertical.
  - Su posición horizontal
- El tipo de acciones que pueden realizarse con él o **métodos**
  - Mover a la derecha.
  - Mover a la izquierda.
  - Mover arriba.
  - Mover abajo.
- A atributos y métodos se les llama **miembros.**

Uso de un objeto

- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

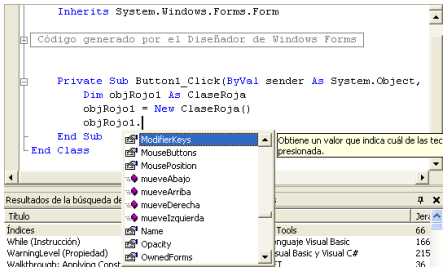
Uso de un objeto

- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

Mail: palasi@palasi.com Web: www.palasi.com

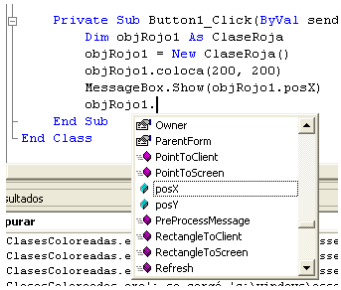
Cómo conocer los métodos de un objeto

- Se escribe un punto al lado de la variable que referencia el objeto y aparece una lista desplegable de miembros.



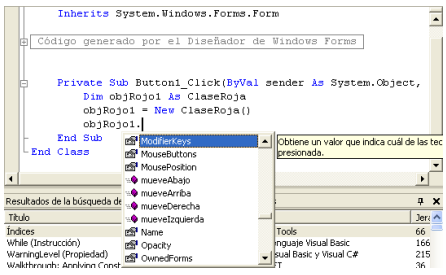
Cómo conocer los métodos de un objeto

- Los atributos aparecen con el icono
- Los métodos aparecen con el icono



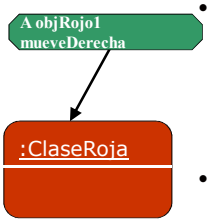
Por ahora, sólo nos fijaremos en unos pocos métodos

- **`mueveAbajo`**, **`mueveArriba`**, **`mueveDerecha`**, **`mueveIzquierda`**.



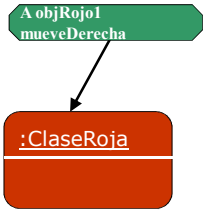
Cómo ejecutar los métodos

- Esta ejecución se llama **llamada a método o mensaje**.



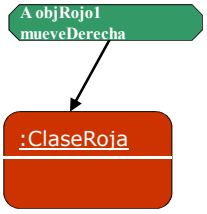
- Es una petición a un objeto para que realice un método (tarea) de las que sabe hacer este objeto.
- El objeto realiza la tarea siguiendo el método que tiene definido.

Método



- El objeto para realizar la tarea que ha pedido el mensaje tiene un método definido.
- Este método está oculto:
  - sólo lo conoce el objeto
  - no se conoce desde el exterior (excepto su nombre).

Desde el punto de vista de la programación



- Los métodos no son más que procedimientos, funciones o subrutinas.
- Por ello tienen
- Nombre.
- Parámetros.
- A veces, resultado.
- Una implementación o cuerpo.

Mail: palasi@palasi.com Web: www.palasi.com

Método o mensaje

- Hay que distinguir entre:
- Método (análogo a procedimiento)
- Mensaje o llamada a Método. (análogo a llamada a procedimiento)

- **Mensaje o llamada a método** es la petición (el clic) que hemos hecho para ejecutar `mueveDerecha.`
- **Método** es el procedimiento que está implementado internamente en el objeto y que se ejecuta, permitiendo al objeto moverse a su izquierda. **Por ahora, no lo conocemos.**

Cómo se haría una llamada a método en VB.NET

- Se utiliza la notación del punto.

objeto.nombremétodo()

- donde ***objeto*** es el nombre de la variable que contiene la referencia al objeto.
- ***nombremétodo*** es el nombre del método al que queremos llamar.

Cómo se haría una llamada a método en VB.NET

- Se utiliza la notación del punto.

objeto.nombremétodo()

- Por ejemplo

```
objRojo1.mueveDerecha()
```

Ejercicio

- Usando el ejercicio anterior, hagan que cuando se haga clic sobre el botón, se cree un objeto y éste se mueva dos veces a la derecha y tres veces abajo.

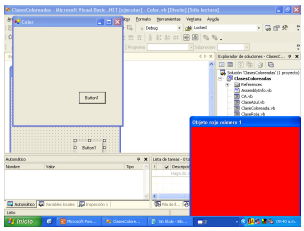
Parámetros

- Cambien el proyecto anterior para que cuando se haga clic se ejecute el siguiente fragmento de código.

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
objRojo1.coloca(500, 300)
```

- Ejecuten el programa y comprueben los resultados.

El objeto rojo creado se coloca en la posición 500 horizontal y 300 vertical



- Esto es porque se ha hecho la llamada a método (mensaje)

```
objRojo1.coloca(500, 300)
```


Mail: palasi@palasi.com Web: www.palasi.com

Parámetros

```
objRojo1.coloca(500, 300)
```

- Al llamar al método **coloca**, hemos especificado la posición horizontal y vertical donde queremos colocar el objeto.
- En general, cuando llamamos a un método, les proporcionamos ("pasamos" en el argot informático) unos datos que necesita para su ejecución.
- Estos datos se llaman parámetros. Como ven es lo mismo que los parámetros de un procedimiento en la programación convencional.

Terminología

- A los datos que proporcionamos a los métodos para que puedan ejecutarse, se les llama "parámetros".
- Al acto de proporcionar un parámetro a un método cuando se le llama, se le llama "pasar un parámetro"

Cómo se pasan parámetros en VB.NET

- Para un procedimiento que tiene parámetros, la forma de llamarlo es la siguiente:

```
objeto.nombremétodo(valorespar)
```

- donde ***objeto*** es la variable que guarda su referencia al objeto.
- donde ***valorespar*** es la lista de los parámetros que queremos pasar separada por comas.
- Por ejemplo

```
objRojo1.coloca(500, 300)
```

Ejercicios

- 1. Creen un objeto verde y colóquenlo en la posición 100 horizontal y 300 vertical.
- 2. En el formulario, creen dos cuadros de texto en los que pondrán la posición donde desean colocar un objeto azul cuando hagan clic.
- 3. Creen un objeto rojo y cámbienle de tamaño para que tenga 100 de alto y 200 de ancho. Para ello, usen el método **cambiaDimension**, cuyos parámetros pueden consultar en el editor VB.NET.
- 4. Hagan lo mismo, pero ahora el ancho y alto del formulario deberán ser introducidos en cuadros de texto.

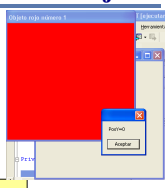
Valor de retorno

- En el código del botón escriban

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
MessageBox.Show("PosX=" &
 objRojo1.ConsiguePosX())
MessageBox.Show("PosY=" &
 objRojo1.ConsiguePosY())
```

- Ejecuten y vean los resultados.

Este programa muestra la posición horizontal y vertical del objeto



Prueben con:

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
objRojo1.coloca(50,30)
MessageBox.Show("PosX=" &
 objRojo1.ConsiguePosX())
MessageBox.Show("PosY=" &
 objRojo1.ConsiguePosY())
```


Mail: palasi@palasi.com Web: www.palasi.com

Valor de retorno

- Algunos métodos proporcionan una información cuando se acaba su ejecución.
- A esta información se le llama **resultado o valor de retorno.**
- Cuando el método proporciona esta información, se dice que **devuelve o retorna un resultado.**

Valor de retorno

- El valor de retorno puede ser usado en cualquier expresión válida en VB.NET.
- En VB.NET, el hecho de retornar un valor divide a los métodos en procedimientos y funciones.
- Los procedimientos son los métodos que NO devuelven resultados. Las funciones son los métodos que devuelven resultados.

Atención

- Los métodos se diferencian de los procedimientos y funciones convencionales en que están ligados a un objeto.
- Los convencionales eran totalmente autónomos.
- Sin embargo, los métodos son procedimientos o funciones que realizamos sobre un objeto y se refieren a ese objeto y a la información de ese objeto.

`ConsiguePosX()` y `ConsiguePosY()`

- Son funciones que retornan como resultado la posición horizontal y vertical de un objeto de los que estamos tratando.
- Hay otras funciones **`ConsigueAncho`** y **`ConsigueAlto`**, que consiguen respectivamente el ancho y alto del objeto.

Ejecutando una función en VB.NET

- En principio es lo mismo que con un procedimiento:

objeto . nombremétodo (valorespar)

- donde ***valorespar*** es la lista de los parámetros que queremos pasar separada por comas.
- Sin embargo ahora, se devuelve un resultado. Con él, hay que hacer algo (integrarlo en una expresión, pasarlo como parámetro a un método, asignarlo, etc)
- Por ejemplo

```
posHorizontal =
  objRojo1.ConsiguePosX()
```

Ejecutando una función en VB.NET

- Normalmente, llamamos a la función y hacemos algo con el resultado

```
posHorizontal =
  objRojo1.ConsiguePosX()
MessageBox.Show(
  objRojo1.ConsiguePosX())
```

- Pero, si no nos interesa el resultado, podemos llamar a la función como si fuera un procedimiento.

```
objRojo1.ConsiguePosX()
```


Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- Creen un formulario con dos cuadros de texto, donde introduciremos dos posiciones (horizontal y vertical).
- El formulario debe crear un objeto azul y después colocarlo en esa posición horizontal y vertical.
- Para ello, usen el método **`coloca`**.

Ejercicio

- Creen un formulario con dos cuadros de texto, donde introduciremos dos porcentajes.
- El formulario debe crear un objeto verde y después reducir proporcionalmente sus dimensiones según los porcentajes introducidos.
- Es decir, si el usuario introduce 50 y 30, el objeto debe reducirse al 50% de ancho y el 30% de alto.
- Usen el método **`cambiaDimension`**.

Uso de un objeto

- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

Recordamos: las 2 cosas que interesan de un objeto

- Sus características o **atributos.**
  - Su posición vertical.
  - Su posición horizontal
- El tipo de acciones que pueden realizarse con él o **métodos**
  - Mover a la derecha.
  - Mover a la izquierda.
  - Mover arriba.
  - Mover abajo.

- A atributos y métodos se les llama **miembros.**

Ya hemos visto los métodos

- Nos queda por ver los atributos, que veremos a continuación.
- Sabemos que una de las cosas que nos interesan de un objeto son sus características, que llamamos atributos.
- Los atributos no son más que una clase de variables.

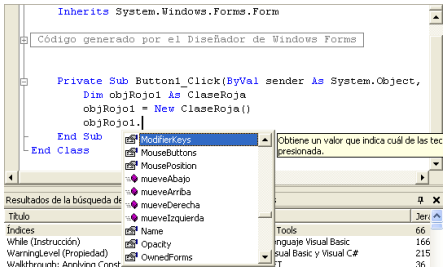
Los atributos son una clase especial de variables

- Son unas variables que están ligadas al objeto (que pertenecen a un objeto), de la misma forma que los métodos son procedimientos y funciones pero ligados al objeto.
- Se diferencian de otro tipo de variables en que:
  - Se definen dentro y asociadas a un objeto.
  - Están accesibles dentro de todo ese objeto.

Mail: palasi@palasi.com Web: www.palasi.com

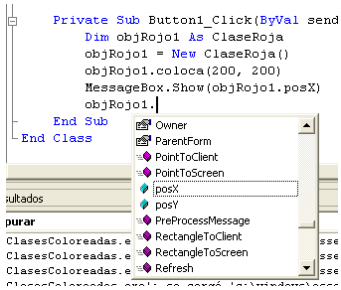
Cómo conocer los atributos de un objeto

- Se escribe un punto al lado de la variable que referencia el objeto y aparece una lista desplegable de miembros.



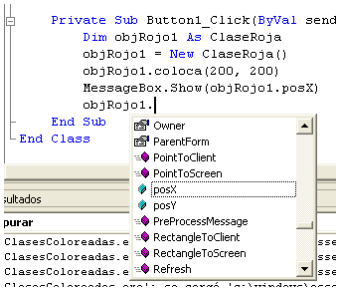
Cómo conocer los atributos de un objeto

- Los atributos aparecen con el icono
- Los métodos aparecen con el icono



En los objetos que manejamos, sólo tenemos 4 atributos

- **`posX`**, **`posY`**, **`alto`**, **`ancho`**.



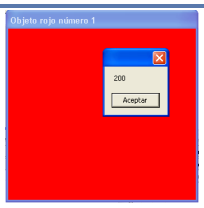
Ejemplo

- Cambien el proyecto anterior para que cuando se haga clic se ejecute el siguiente fragmento de código.

```
Dim objRojo1 As ClaseRoja
objRojo1 = New ClaseRoja()
objRojo1.coloca(200, 200)
MessageBox.Show(objRojo1.posX)
```

- Ejecuten el programa y comprueben los resultados.

Aparece la posición horizontal del objeto



- **`posX`** es el atributo que guarda la posición horizontal del objeto.
- **`posY`**, **`ancho`** y **`alto`** guardan la posición vertical, el ancho y el alto.

Obtener el valor de un atributo

- Para obtener el valor de un atributo, se utiliza la notación del punto:

objeto.nombreatributo

- donde ***objeto*** es el nombre de la variable que contiene la referencia al objeto.
- ***nombreatributo*** es el nombre del atributo del cual queremos saber su valor.

Mail: palasi@palasi.com Web: www.palasi.com

Así, para obtener la posición horizontal

- Obtenemos el valor del atributo **`posX`**, con la siguiente expresión:

```
objRojo1.posX
```

- Esto nos devuelve un valor, que podemos incluir en cualquier expresión, mostrarlo por pantalla, etc.

Estado

- Sabemos que una de las cosas que nos interesan de un objeto son sus características, que llamamos atributos.
- Cada uno de estos atributos tiene un valor.
- A partir de ahora, al conjunto del valor de los atributos, se le llamará **Estado**.
- En nuestro caso, el estado es el conjunto de valores de los atributos **`posX`**, **`posY`**, **`alto`**, **`ancho`**.

Ejercicios

- 1. Cambie la posición de un objeto recién creado con el método **`coloca`** repetidas veces. Después de cada vez que se cambie la posición, imprima el estado por pantalla.
- 2. Cambie el tamaño de un objeto recién creado con el método **`cambiaDimension`**. Después de cada vez que se cambie la dimensión, imprima el estado por pantalla.

Sabemos que los atributos son una clase especial de variables.

- Hasta ahora, hemos recuperado su valor con la notación del punto.
- Pero de una variable, nos interesan dos acciones: recuperar el valor y guardar un valor.
- ¿Cómo guardamos (modificamos) el valor de un atributo?

Modificando el valor de un atributo

- Para guardar o saber el valor de un atributo, se utiliza la asignación:

```
objeto.nombreatributo = expresión
```

- donde ***expresión*** es el nombre del atributo del cual queremos saber su valor.
- Por ejemplo

```
objRojo1.posX = 100
```

Ejercicio

- Cambien la posición de un objeto recién creado, modificando el valor de sus atributos.
- Ejecuten el programa y vean qué es lo que aparece.

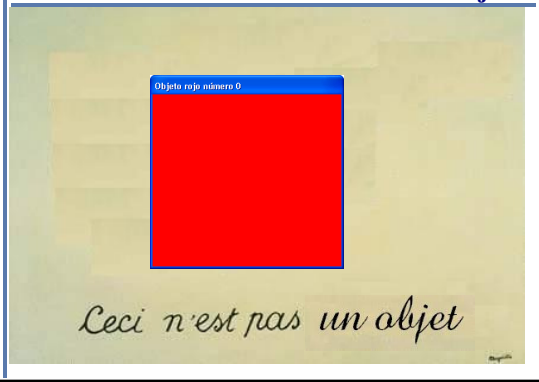
Mail: palasi@palasi.com Web: www.palasi.com

Aquí hay algo extraño

- A pesar de que hemos cambiado la **`posX`** y la **`posY`**, el objeto no se mueve de sitio.

- ¿Qué es lo que pasa?

Recordemos que el cuadrado rojo no es un objeto



Es la representación gráfica de un objeto

En realidad, el estado del objeto sí ha cambiado

- Compruébenlo escribiendo el valor de los atributos por pantalla con un **`MessageBox.Show`**
- Lo que pasa que este cambio no ha sido trasladado a su representación gráfica. NO se ha dibujado.

El método `Dibuja()` dibuja la representación gráfica del objeto

- Según los valores de los atributos del objeto.
- Incluyan el método **`Dibuja()`** y verán cómo el cambio en los atributos se refleja en pantalla.

Ejercicio

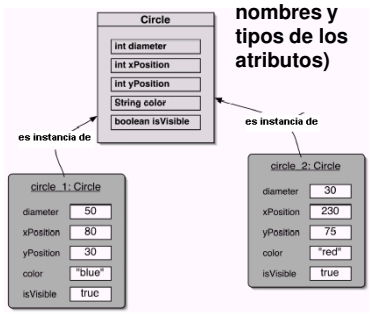
- Cambien el ancho y alto de un objeto, usando los respectivos atributos.
- Ejecuten el programa. Incluyan el método **`Dibuja()`** para que los cambios se trasladen a la representación gráfica en pantalla.

Aclarando las ideas

- Sabemos que los atributos son variables y, por lo tanto, nos interesa su nombre, su tipo y su valor.
- Los objetos de una misma clase (por ejemplo, de la clase **`ClaseRoja`**) pueden tener diferente estado, es decir, diferente valor de los atributos.
- En cambio, el nombre y el tipo de los atributos es el mismo para todos los objetos de la misma clase.

Una clase y sus objetos

CLASE (Define los nombres y tipos de los atributos)

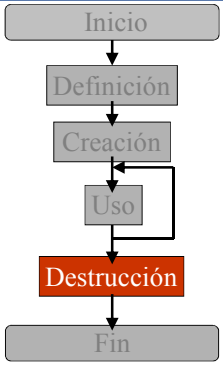


OBJETOS (Definen los valores de los atributos)

Comparación con el mundo real

- La clase (especificación) Auto define cuál son los atributos: motor, color, etc y el tipo de ellos.
- Cada objeto, cada auto, tiene un color diferente, un motor diferente, etc.

Ciclo de vida de un objeto



Destruir un objeto

- Un objeto se destruye porque se destruyen todas las referencias que apuntan a él. Ejemplo:

```
Public Class Color
    Inherits System.Windows.Forms.Form

    Código generado por el Diseñador de Windows Forms

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArg
        Dim objRojo1 As ClaseRoja
        objRojo1 = New ClaseRoja()
    End Sub
End Class
```

Destruir un objeto

- Un objeto se destruye porque se destruyen todas las referencias que apuntan a él. Ejemplo:

```
Private Sub Button1_Click(...
  Dim objRojo1 As ClaseRoja
  objRojo1 = New ClaseRoja()
End Sub
```


Mail: palasi@palasi.com Web: www.palasi.com

Análisis de la ejecución de este evento

```
Private Sub Button1_Click(...
    Dim objRojo1 As ClaseRoja
```

Crea una referencia `objRojo1` ●

```
    objRojo1 = New ClaseRoja()
```

Crea un objeto `objRojo1` → :ClaseRoja

```
End Sub
```

Se acaba el programa. La variable `objRojo1`, que referenciaba el objeto, se libera. Como ya no hay referencias al objeto, el objeto se destruye.

De hecho, esto no es cierto del todo

- Cuando no hay referencias, el objeto queda no accesible. Hemos dicho que se destruye.
- De hecho esto no es del todo cierto, el recolector de basura de VB.NET tarda un tiempo en destruir el objeto y liberar la memoria que usaba.
- Sin embargo, esto no nos importa. Para el programador el objeto no está accesible y debe pensar en él como destruido, pues ya no puede usarlo ni accederlo de ninguna manera.
- A partir de ahora, no pensaremos en estos detalles y consideraremos el objeto destruido cuando no hay referencias.

Consideramos que se destruye cuando no hay referencias

- Así, si queremos destruir un objeto en medio de un programa podemos lograrlo asegurando que todas sus referencias no apunten a nada (o sea, apunten a **`Nothing`**).
- Para hacer que una referencia apunte a **`Nothing`**.

```
varReferencia = Nothing
```

`Nothing` es el literal que indica que una referencia no apunta a nada

- Si queremos saber si la referencia apunta a un objeto basta con saber el valor de la expresión

```
IsNothing(varReferencia)
```

El objeto "se destruye" cuando no tiene referencias

```
Private Sub Button1_Click(...
    Dim objRojo1 As ClaseRoja
    objRojo1 = New ClaseRoja()
    objRojo1 = Nothing

    'objRojo1.mueveDerecha() daría
    'error en este punto
End Sub
```

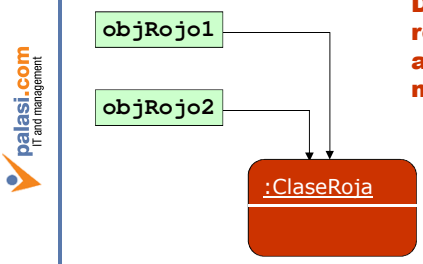
EL OBJETO YA NO ES ACCESI-BLE EN ESTE PUNTO

Asignar a `Nothing` no tiene porque destruir el objeto

```
Private Sub Button1_Click(...
    Dim objRojo1 As ClaseRoja
    objRojo1 = New ClaseRoja()
    objRojo2 = objRojo1
    objRojo1 = Nothing

    objRojo2.mueveDerecha()  OK
End Sub
```

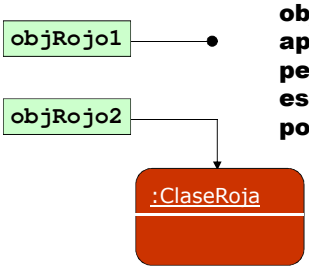
- Se destruye la referencia **`objRojo1`**, pero siempre queda válida la referencia **`objRojo2`**.

Antes de asignar el `Nothing` la situación es la siguiente



Dos referencias apuntan al mismo objeto

Después de asignar el `Nothing` la situación es la siguiente



objRojo1 no apunta a nada pero el objeto es referenciado por objRojo2

NO QUEDA INACCE-SIBLE NI SE DESTRUYE

Sólo se destruye cuando no hay ninguna referencia.

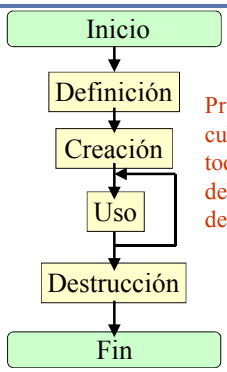
Prográmenlo

- Programen este último fragmento de código y comprueben que el objeto no desaparece.
- Ahora, pongan a **`Nothing`** la segunda referencia y comprueben que el objeto ya no está accesible.

3. Introducción a la programación orientada a objetos

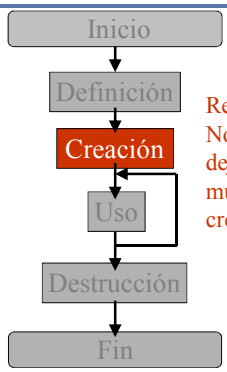
- 3.1. Qué es la programación orientada a objetos.
- 3.2. El concepto de objeto.
- 3.3. Ciclo de vida de un objeto.
- 3.4. Programación de clases.

Ciclo de vida de un objeto



Pregunta: ¿Hemos cubierto todo lo que debemos saber de los objetos?

Ciclo de vida de un objeto



Respuesta: No, nos hemos dejado un aspecto muy importante de la creación de un objeto

Recordemos: Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa especificación) que es la clase.

Recordemos: Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa especificación) que es la clase.

Hasta ahora se crearon objetos de clases ya programadas.

- Hasta ahora habíamos considerado las clases "desde fuera": nos limitábamos a ejecutarlas sin saber cómo estaban implementados. Éramos usuarios de las clases.
- Esto simplifica la programación y es la forma en qué debemos considerarlas cuando estamos programando otras clases.
- El problema es que es lo único que sabemos.

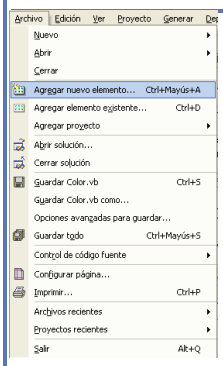
Sólo sabemos crear objetos de clases ya programadas.

- Esto nos limita, ya que no podemos crear un objeto con características diferentes a las que los otros han programado por nosotros.
- En el ejemplo anterior, no podemos crear un objeto con forma de estrella.

Sólo sabemos crear objetos de clases ya programadas.

- Conclusión: Para poder crear cualquier tipo de objeto que deseemos, tenemos que saber programar clases ("moldes").
- Y eso aún no lo sabemos.

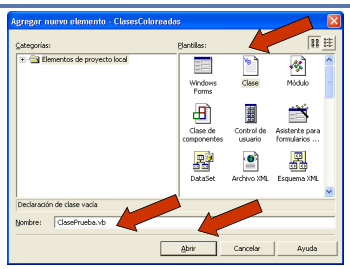
Cómo programamos clases



- Vamos al menú **Archivo** y a la opción **Agregar nuevo elemento**

Mail: palasi@palasi.com Web: www.palasi.com

Cómo se crea una nueva clase



- Hacemos clic en el icono de clase, escribimos el nombre de la clase y hacemos clic en "Abrir".

Cómo se crea una nueva clase

- Aparecerá una ventana, donde podemos escribir el código de la clase.
- Pero, ¿qué código es éste?

Código para declarar (programar) clases en VB.NET

```
Public Class NombreClase
    Declaración de atributos
    Declaración de métodos
End Class
```

- Este es un código simplificado. Más adelante veremos que hay más opciones.

Nombre de la clase

```
Public Class NombreClase
    Declaración de atributos
    Declaración de métodos
End Class
```

- Por convención, escribiremos el nombre de las clases con la primera letra en mayúscula y el nombre de los objetos en minúsculas.
- Ya veremos que después esto nos será útil.

Declaración de atributos

```
Public Class NombreClase
    Declaración de atributos
    Declaración de métodos
End Class
```

- Los atributos se declaran parecido a una declaración de variables pues no son más que un caso especifico de variables.
- Se incluye el nombre, el tipo y (opcionalmente) un valor inicial.

Sintaxis de declaración de atributos

ámbito nombre **As** *tipo* = *valorinicial*

- Es una declaración de variables y, cómo tal, tiene un nombre del atributo y un tipo.
- Opcionalmente, se puede añadir un valor inicial.
- También se especifica un ámbito del atributo, como veremos a continuación.

Mail: palasi@palasi.com Web: www.palasi.com

Modificadores de acceso (ámbito)

- **`Public`**. El acceso al atributo es público. Se puede utilizar el atributo en cualquier circunstancia.
- **`Protected`**. El atributo es accesible desde la clase que lo define y las clases que derivan de ella.
- **`Friend`**. El atributo es accesible dentro del mismo proyecto.
- **`Protected Friend`**. El atributo es accesible dentro del mismo proyecto o bien desde la clase que lo define y las clases que derivan de ella.
- **`Private`**. El atributo sólo puede ser usado dentro de la clase que lo define.

Sintaxis de declaración de atributos

ámbito nombre **`As`** *tipo* = *valorinicial*

- Ejemplos de declaraciones válidas:

```
Private nombre As String
Public objRojo As ClaseRoja =
 New ClaseRoja()
Public posX As Integer
Private contador As Integer = 0
```

Declaración de métodos

```
Public Class NombreClase
    Declaración de atributos
    Declaración de métodos
End Class
```

- Ahora vamos a la declaración de métodos.
- Vamos a distinguir entre funciones (métodos que devuelven resultado) y procedimientos (métodos que no devuelven nada).

Declaración de procedimientos

ámbito **`Sub`** *nombre*(*listaparam*)
    instrucciones
`End Sub`

- Los elementos en negro son opcionales
- ***ámbito*** es un modificador de acceso (ya se vio).
- ***nombre*** es el nombre del método.
- ***listaparam*** es la lista de parámetros en formato que ya veremos
- ***instrucciones*** es el conjunto de sentencias que ejecutará el procedimiento. **Se le llama CUERPO del método.**

Declaración de procedimientos

ámbito **`Sub`** *nombre* (*listaparam*)
    instrucciones
`End Sub`

- ***listaparam*** es la lista de parámetros en el formato:
  - **`ByVal`** *nombre1* **`As`** *tipo1*, **`ByVal`** *nombre2* **`As`** *tipo2*, ... **`ByVal`** *nombren* **`As`** *tipon*.
- **`ByVal`** quiere decir que el parámetro se pasa por valor. Puede reemplazarse por **`ByRef`**, que significa que el parámetro se pasa por referencia. Por ahora, no nos ocuparemos de esto.

Ejemplo de un procedimiento

```
Public Sub escribeSuma(ByVal
  num1 As Integer, ByVal num2
  As Integer)
  MessageBox.Show(num1+num2)
End Sub
```


Mail: palasi@palasi.com Web: www.palasi.com

Declaración de funciones

```
ámbito Function nombre(listaparam) As tipo
        instrucciones
End Function
```

- Como ven, es lo mismo que los procedimientos, pero ahora la palabra clave es **`Function`** y se especifica el tipo del resultado.
- Además una de las instrucciones especifica que valor es el resultado de la función. Esta instrucción se escribe

```
Return expresion
```

Ejemplo de una función

```
Public Function suma(ByVal num1
  As Integer, ByVal num2 As
  Integer) As Integer
  Return num1+num2
End Sub
```

Vuelvan a recuperar el proyecto anterior

- El de "ClasesColoreadas".
- Miren como está definida la clase "ClaseRoja" que hemos usado.
- También miren el formulario.

Atención

- Como dijimos al principio del curso, en VB.NET, como en otros lenguajes orientados a objetos, todo es una clase.
- Fijémonos que los formularios también son clases, comienzan con **`Public Class`**.

```
Imports System.Windows.Forms
Public Class Color
    Inherits System.Windows.Forms.Form
    Código generado por el Diseñador de Windows Forms
End Class
```

- Tienen algunos elementos que aún no hemos visto, como **`Imports`** e **`Inherits`**. Ya los veremos.

Definimos un atributo en el formulario que sea de clase `ClaseRoja`

- Lo definimos como privado y creamos el objeto.

```
Private objRojo As ClaseRoja =
  New ClaseRoja()
```

- Creen cuatro botones en el formulario que muevan este objeto a la derecha, izquierda, arriba y abajo.

Ejercicio

- Creen un nuevo proyecto.
- Crear en el proyecto una clase calculadora con métodos para obtener la suma, resta, producto, división, cuadrado y media aritmética de números pasados como parámetros.
- Crear un formulario en el proyecto con dos cuadros de texto y varios botones, que permitan hacer todas estas operaciones usando la clase.
- Ejecutar.

Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- Crear un nuevo proyecto.
- Crear en él una clase que modele una libreta de banco. En principio, sólo nos interesará el saldo de la libreta y su número de libreta.
- Con esta libreta, se pueden inicializar la libreta (entonces se fijará un número de libreta), añadir depósitos, añadir retiros, obtener el saldo y el número de libreta.
- Crear un formulario para hacer todas estas operaciones a una única libreta, a partir de cuadros de texto y botones.

Solución

```
Public Class Libreta
Private Numero As Integer
Private Saldo As Double
Public Sub Iniciar(ByVal nLibreta As Integer)
  Numero = nLibreta
  Saldo = 0
End Sub
Public Function ObtenerNumero() As Integer
  Return Numero
End Function
Public Function ObtenerSaldo() As Double
  Return Saldo
End Function
```

Solución

```
Public Sub Depositar(ByVal monto As Double)
  Saldo = Saldo +monto
End Sub
Public Function Retirar(ByVal monto As
  Double) As Boolean
  If monto > Saldo Then
     Return False
  Else
     Saldo = Saldo - monto
     Return True
  EndIf
End Function
End Class
```

Solución

- Tendríamos un formulario así.

Solución

```
Public Class Form1
    Inherits System.Windows.Forms.Form
Private cuenta As Libreta
Private Sub ButtonIniciar_Click ...
  cuenta = New Libreta()
  cuenta.Iniciar(Me.TextBox1.Text)
End Sub
Private Sub ButtonConsultar_Click ...
  MessageBox.Show ("El numero es " &
  cuenta.ObtenerNumero() & " y el saldo " &
  cuenta.ObtenerSaldo())
End Sub
```

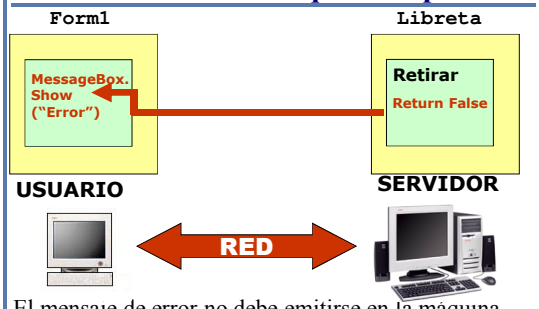
Solución

```
Private Sub ButtonDepositar_Click ...
  cuenta.Depositar(Me.TextBox1.Text)
End Sub
Private Sub ButtonRetirar_Click ...
  Dim correcto As Boolean
  correcto = _
   cuenta.Retirar(Me.TextBox1.Text)
  If Not Correcto Then
     MessageBox.Show ("No se puede retirar
     más  que el saldo. La operación no
     se completó")
  End If
End Sub
```

Observación

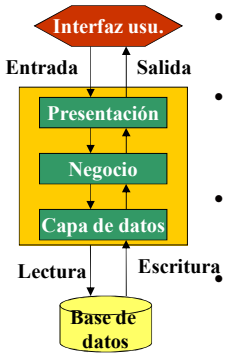
- Como ven, todas las operaciones de entrada/salida se han concentrado en el formulario.
- La clase **`Libreta`** no tiene operaciones de entrada/salida sino sólo cálculo.
- Así, si intentamos retirar más del saldo, la clase no da un mensaje de error, sino que retorna un valor y es el formulario quien se encarga de emitir el mensaje de error.
- ¿Por qué?

Las clases pueden estar en máquinas separadas



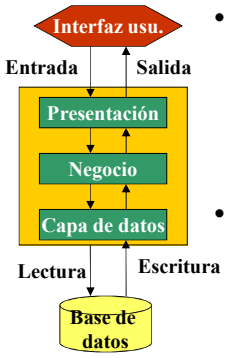
- El mensaje de error no debe emitirse en la máquina donde se detecta el error, sino en la máquina del usuario.

Esto sigue la filosofía de arquitectura en 3 capas



- El programa se divide en capas (conjuntos de clases).
- Una capa (presentación) sólo se ocupa de la E/S en la interfaz de usuario.
- Una capa (negocio) sólo se ocupa del cálculo.
- Una capa (datos) sólo se ocupa del acceso a la base de datos.

Las clases de una capa sólo llaman a las de la capa inferior



- En nuestro caso, el formulario formaría parte de la capa de presentación y la clase **`Libreta`** de la capa de negocio (cálculo).
- El formulario llama a la clase **`Libreta`**. Las clases de una capa sólo llaman a las de la capa inferior.

Accediendo a los miembros de la misma clase

- Sabemos que la forma de acceder a los miembros de una clase es con la notación del punto:

```
objeto.nombreatributo
objeto.nombremétodo(valorespar)
```

- Pero frecuentemente, desde un método de una clase, queremos acceder a los miembros de esta misma clase. Esto permite que los objetos de esa clase puedan acceder a sus propios miembros.

Accediendo a los miembros de la misma clase

- En este caso, puede prescindirse del punto y de la referencia a objeto

```
nombreatributo
nombremétodo(valorespar)
```

- Otra opción es usar la palabra reservada **`Me`**

```
Me.nombreatributo
Me.nombremétodo(valorespar)
```

La palabra reservada Me

- Sirve para referenciar el objeto actual desde el código del mismo. Es una autoreferencia.

Ejercicio

- En el proyecto que estamos utilizando, mirar cómo están programadas las clases **`ClaseRoja`**, **`ClaseVerde`**, **`ClaseAzul`**.

Ejercicio

- Ampliar la clase de la libreta, para añadir una función que determine si una libreta es igual a la que estamos programando.
- Usen la estructura de control

```
If expresión Then
   instrucciones1
Else
   instrucciones2
End If
```

Ejercicio

- Ampliar la misma clase con un procedimiento que, usando la función anterior, escriba por pantalla si las dos libretas son iguales o no

Solución

```
Public Sub ComparaConOtraLibreta(ByVal
  l As Libreta)
   If l.iguales(Me) Then
     MessageBox.Show("Iguales")
   Else
      MessageBox.Show("Diferentes")
   End If
End Sub
```

Fijemos como podemos enviar el objeto actual como parámetro con **`Me`**.

Otra solución

```
Public Sub ComparaConOtraLibreta(ByVal
  l As Libreta)
   If Me.iguales(l) Then
     MessageBox.Show("Iguales")
   Else
      MessageBox.Show("Diferentes")
   End If
End Sub
```

Fijemos como podemos enviar el objeto actual como parámetro con **`Me`**.

Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- Completar la clase de la libreta para que de error en el caso de que se retire más del saldo.
- Intentar que el mensaje de error no se emita desde la clase, sino que esta devuelva un resultado, que sea usado por el formulario para emitir el mensaje.

Ejercicio

- Crear una clase que modele un banco. Para simplificar, supondremos que un banco tiene 3 libretas como máximo.
- Las operaciones que nos interesan de un banco son añadir una nueva libreta, ingresar en una libreta (dado su número), retirar de una libreta, obtener el saldo de una libreta, obtener el saldo total y obtener el número de libretas.
- Deben usar la clase de libreta que han programado anteriormente.
- Consideren todos los errores que se puedan dar e intenten que sea el formulario el que los genere.

Solución (1)

```
Public Class Banco
  Private cuenta1 As Libreta
  Private cuenta2 As Libreta
  Private cuenta3 As Libreta
```

Solución (2)

```
Public Function AgregarLibreta(ByVal nLibreta As
   Libreta) As Boolean
   If IsNothing(cuenta1) Then
     cuenta1 = nLibreta
     Return True
   ElseIf IsNothing(cuenta2) Then
     cuenta2 = nLibreta
     Return True
   ElseIf IsNothing(cuenta3) Then
     cuenta3 = nLibreta
     Return True
  Else
     Return False
  End If
End Function
```

Solución (3)

```
Public Function ObtenerLibreta(ByVal numLib As Integer) As
    Boolean
    If Not IsNothing(cuenta1) Then
      If numLib = cuenta1.ObtenerNumero() Then
            Return cuenta1
      EndIf
    EndIf
    If Not IsNothing(cuenta2) Then
      If numLib = cuenta2.ObtenerNumero() Then
            Return cuenta2
      EndIf
    EndIf
    If Not IsNothing(cuenta3) Then
      If numLib = cuenta3.ObtenerNumero() Then
            Return cuenta3
      EndIf
    EndIf
   Return Nothing
End Function
```

Solución (4)

```
Public Function SaldoTotal() As Double
    Dim totalSaldo As Double
    totalSaldo = 0
   If Not IsNothing(cuenta1) Then
       totalSaldo += cuenta1.ObtenerSaldo()
   EndIf
   If Not IsNothing(cuenta2) Then
       totalSaldo += cuenta2.ObtenerSaldo()
   EndIf
   If Not IsNothing(cuenta3) Then
       totalSaldo += cuenta3.ObtenerSaldo()
   EndIf
   Return totalSaldo
End Function
```

Solución (5)

```
Public Function NumeroLibretas() As Integer
    Dim totalLib As Double
    totalLib = 0
    If Not IsNothing(cuenta1) Then
        totalLib += 1
    EndIf
    If Not IsNothing(cuenta2) Then
        totalLib += 1
    EndIf
    If Not IsNothing(cuenta3) Then
        totalLib += 1
    EndIf
    Return totalLib
End Function
End Class
```

Ejemplo de uso de la clase `Banco` (1)

- Añadimos una nueva libreta con código 123 al banco que está en la variable **`banco1`**.

```
Dim nuevaCuenta As Libreta
nuevaCuenta = New Libreta(123)
banco1.AgregarLibreta(nuevaCuenta)
```

- Como vemos utilizamos la clase **`Libreta`** para crear la libreta antes de agregarla al Banco.

Ejemplo de uso de la clase `Banco` (2)

- Hacemos un retiro a la libreta con código 24 al banco que está en la variable **`banco1`**.

```
Dim cuenta As Libreta
Dim correcto As Boolean
cuenta = banco1.ObtenerLibreta(24)
If IsNothing(cuenta) Then
  MessageBox.Show("La cuenta no existe")
Else
  correcto = cuenta.Retirar(500)
  If Not correcto Then
     MessageBox.Show ("La operación no se
       completó. Saldo insuficiente")
  End If
End If
```

Ejemplo de uso de la clase `Banco` (2)

- Primero se obtiene una referencia a la libreta y se hace el retiro.

```
Dim cuenta As Libreta
Dim correcto As Boolean
cuenta = banco1.ObtenerLibreta(24)
If IsNothing(cuenta) Then
  MessageBox.Show("La cuenta no existe")
Else
  correcto = cuenta.Retirar(500)
  If Not correcto Then
     MessageBox.Show ("La operación no se
       completó. Saldo insuficiente")
  End If
End If
```

Como se ve en estos ejemplos

- Para hacer una determinada tarea, colaboran objetos de la clase **`Banco`** y objetos de la clase **`Libreta`**.
- La clase **`Banco`** no tiene operaciones para retirar, depositar o crear libretas ya que esto corresponde a la clase **`Libreta`**.
- De esta manera, evitamos la repetición de código, lo que es bueno para el mantenimiento.
- Esta es la forma orientada a objetos de programar: cada clase tiene sus responsabilidades y no se mezcla con las otras.

Temario del curso

Mail: palasi@palasi.com Web: www.palasi.com

4. Otros conceptos básicos de VB.NET

- 4.1. Variables locales
- 4.2. Tipos de datos en VB.NET
- 4.3. Paso de parámetros
- 4.4. Estructuras de control
- 4.5. Arreglos.
- 4.6. Propiedades.
- 4.7. Concepto de interfaz e implementación.

Otros conceptos básicos de Visual Basic .NET

- Antes de seguir con conceptos nuevos de la programación orientada a objetos, queremos repasar alguna conceptos básicos de VB.NET, que nos permitan implementar métodos más elaborados que los que hemos hecho hasta ahora.
- Muchos de estos conceptos eran parecidos a los que existían en VB 6, pero otros han cambiado.
- Dada la limitación del tiempo, veremos sólo lo más básico.

4. Otros conceptos básicos de VB.NET

- 4.1. Variables locales
- 4.2. Tipos de datos en VB.NET
- 4.3. Paso de parámetros
- 4.4. Estructuras de control
- 4.5. Arreglos.
- 4.6. Propiedades.
- 4.7. Concepto de interfaz e implementación.

Variables locales

- Observemos la siguiente función:

```
Public Function Media(ByVal n1 As Double,
  ByVal n2 As Double) As Double
  Dim suma As Double
  suma = s1+s2
  Return suma/2
End Function
```

- Hemos utilizado una variable auxiliar **suma** para realizar un cálculo auxiliar.
- Esto mejora la legibilidad y puede ser necesario o muy conveniente en métodos más complicados.

Variables locales

```
Public Function Media(ByVal n1 As Double,
  ByVal n2 As Double) As Double
  Dim suma As Double
  suma = s1+s2
  Return suma/2
End Function
```

- La variable auxiliar **suma** sólo existe dentro del método.
- Se le llama variable local.

Variables locales

```
Public Function Media(ByVal n1 As Double,
  ByVal n2 As Double) As Double
  Dim suma As Double
  suma = s1+s2
  Return suma/2
End Function
```

Son variables que:

- Se declaran dentro de un método.
- Sólo están accessibles dentro de este método.
- Son parte de la implementación, pero no de la interfaz.

Declaración de variables locales

- La declaración de variables locales puede tener la siguiente sintaxis:

```
Dim nombre As tipo
```

- Donde ***nombre*** y ***tipo*** son, respectivamente el nombre y tipo de la variable que deseamos declarar. Normalmente las declaraciones van al principio del método.
- Opcionalmente se puede asignar un valor inicial en el momento de la declaración, de esta forma:

```
Dim nombre As tipo = valorinicial
```

También puede declararse varias variables a la vez

- Simplemente el nombre de las diferentes variables se separa por comas.

```
Dim nombre1, nombre2 As tipo
```

- También variables de diferentes tipos

```
Dim nombre1 As tipo1, nombre2 As tipo2
```

- O se puede asignar varios valores iniciales a la vez.

```
Dim nombre1 As tipo1 = valorinicial1,
    nombretipo2 As tipo2 = valorinicial2
```

- Estas dos últimas posibilidades, sin embargo, no son recomendables por su poca legibilidad.

Acceso a variables locales

```
nombrevariable
nombrevariable = expresión
```

- Se accede a ellas como cualquier otra variable.
- Pero sólo son accesibles dentro del método.
- Si las accedemos fuera, dan error de compilación.

4. Otros conceptos básicos de VB.NET

Declaración de variables

- Tanto si son variables locales como si son atributos, siempre debe especificarse un tipo.

```
Dim nombre As tipo = valorinicial
ambito nombre As tipo = valorinicial
```

- El tipo puede ser una clase que definimos nosotros o bien un tipo predefinido del lenguaje.
- Esto nos hace plantearnos cuáles son los tipos predefinidos de Visual Basic .NET

Tipos de datos primitivos

- Tipos de datos predefinidos en el lenguaje, que podemos utilizar sin definirlos.
- Además, al contrario que otros tipos, permiten especificar literales para ellos.
- Los literales son expresiones de texto que representan valores. Así `2` representa el entero 2 y `"Hola"` representa la cadena Hola.

Mail: palasi@palasi.com Web: www.palasi.com

Tipos de datos primitivos en VB .NET

`Boolean`	`True` o `False`
`Byte`	Números enteros de 0 a 255
`Short`	Números enteros de –32768 a 32767
`Integer`	Enteros de –2,147,483,648 a 2,147,483,647
`Long`	Enteros de -9223372036854775808 a 9223372036854775807
`Decimal`	Números reales de punto fijo de hasta 28 decimales que, al quitarles el punto decimal, su valor absoluto no es mayor de 79.228.162.514.264.337.593.543.950.335
`Single` `Double`	Reales de punto flotante. Inexactos. No los usaremos en ap. de empresa.

Tipos de datos primitivos en VB .NET

`Char`	Un carácter Unicode (2 bytes). Literales entre comillas dobles.
`String`	Cadenas de caracteres Unicode entre 0 y 2000 millones de caracteres . Literales entre comillas dobles
`Date`	Fechas y horas entre el año 1 y el 9999 con precisión de 100 nanosegundos. Literales entre #.
`Object`	No lo veremos por ahora.
Estructuras	No las veremos por ahora.

Operaciones con los tipos de datos numéricos

- Hay los mismos operadores que en VB6: `+`, `–`, `*`, etc.
- Pero ahora hay formas sintácticas adicionales:

Op.	Uso	Equivalente	Descripción
`+=`	*var+=expr*	*var=var+expr*	Aumenta *var* con *expr*
`–=`	*var–=expr*	*var=var–expr*	Disminuye *var* con *expr*
`*=`	*var*=expr*	*var=var*expr*	Multiplica *var* con *expr*
`/=`	*var/=expr*	*var=var/expr*	Divide *var* con *expr*

La mayoría de estos tipos predefinidos

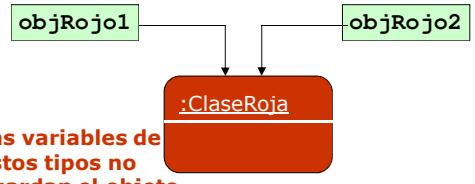
- Son tipos de valor.
- Por lo tanto, se comportan de forma diferente a los tipos de referencia, que son las clases que hemos visto hasta ahora.
- De ahí que nos interese saber la diferencia.

Tipos de datos en Visual Basic .NET

- **Tipos de referencia.** Son las clases que hemos visto hasta ahora. En las variables de estos tipos, se guarda una referencia al objeto.
- **Tipos de valor**. En las variables de estos tipos, se guarda un valor (no una referencia a objeto), por lo que se comportan de la misma manera que los tipos básicos de los lenguajes tradicionales.

Tipos de referencia

```
objRojo1 = New ClaseRoja()
objRojo2 = objRojo1
```



Las variables de estos tipos no guardan el objeto (valor) sino una referencia al objeto (valor)

Si hacemos algo a `objRojo1`, queda hecho para `objRojo2`.

Esto contrasta con los tipos por valor

```
entero1 = 5
entero2 = entero1
```

En las variables se guarda el mismo valor y no una referencia al valor. Cada variable tiene una copia diferente e independiente.

Si hacemos algo a `entero1`, `entero2` no se entera.

Tipos por valor y tipos por referencia

- Los tipos por valor incluyen:
  - Todos los tipos de datos primitivos, excepto **`Object`** y **`String`**.
  - Las enumeraciones, que aún no hemos visto.
- Los tipos de referencia incluyen:
  - **`String`** y **`Object`**.
  - Todos los arrays.
  - Los tipos que son clases, ya sea definida por el usuario o de las bibliotecas de .NET.

En resumen

- Todos los tipos son por referencia, excepto los numéricos, **`Boolean`**, **`Char`**, **`Date`** y enumeraciones.

4. Otros conceptos básicos de VB.NET

Paso por valor y paso por referencia

- Esto está relacionado con el tema anterior y no es específico de la programación orientada o objetos.
- A un método un parámetro se puede pasar:
- Por valor. En el parámetro se pasa una copia del objeto. El objeto original queda inalterado por lo que se hace dentro del método. Palabra clave **`ByVal`**.
- Por referencia. En el parámetro se pasa una referencia al objeto. Se puede pensar cómo si se pasara el objeto original. Todo lo que se haga con el parámetro se hace con el objeto original. Palabra clave **`ByRef`**

Ejemplo

```
Public Class Ejemplo
 Sub IncreValor(ByVal parametro As
  Integer)
   parametro+=1
 End Sub
 Sub IncreReferencia(ByRef parametro As
  Integer)
   parametro+=1
 End Sub
End Class
```

- Los dos métodos incrementan el parámetro que se les pasa. La única diferencia es que al primero se le pasa por valor y al segundo por referencia.

Ejemplo

```
Dim original As Integer
Dim ejemplo As Ejemplo
ejemplo = New Ejemplo()

original = 3
MessageBox.Show(original)

ejemplo.IncreValor(original)
MessageBox.Show(original)

ejemplo.IncreReferencia(original)
MessageBox.Show(original)
```

3

3

4

El paso por valor no modifica el original. Pero sí por referencia

```
Dim original As Integer
Dim ejemplo As Ejemplo
ejemplo = New Ejemplo()

original = 3
MessageBox.Show(original)

ejemplo.IncreValor(original)
MessageBox.Show(original)

ejemplo.IncreReferencia(original)
MessageBox.Show(original)
```

3

3

4

Esta diferencia sólo se aplica a los tipos por valor

- Estos pueden pasarse por valor o por referencia y hay una diferencia (como en el caso anterior de **`Integer`**).
- En cambio, con los tipos por referencia (la mayoría) no hay diferencia entre **`ByVal`** y **`ByRef`**.
- El parámetro es una variable y contiene una referencia por lo cual los cambios dentro del método se aplican al original. Siempre es por referencia.

Resumiendo

	Tipos de valor	Tipos de referencia
`ByVal`	Paso por valor	Paso por referencia
`ByRef`	Paso por referencia	Paso por referencia

4. Otros conceptos básicos de VB.NET

Estructuras de control

- Normalmente el orden de ejecución del programa es secuencial: las instrucciones se ejecutan de arriba abajo.
- Si necesitamos cambiar este orden de ejecución, utilizamos estructuras de control: condicionales, ciclos, etc.
- Ya conocemos estas estructuras de control. Sólo queremos ver brevemente cómo se escriben en Visual Basic .NET.

Mail: palasi@palasi.com Web: www.palasi.com

Condicionales: Estructura If

```
If condicion Then
   Instrucciones
Else
   Instrucciones
End If
```

- La rama **Else** (en rojo) es opcional

Condicionales: Estructura If .. ElseIf

```
If condicion Then
   Instrucciones
ElseIf condicion Then
   Instrucciones
ElseIf condicion Then
   Instrucciones
...
Else
   Instrucciones
End If
```

- Puede haber tantas ramas **ElseIf** como deseemos.
- La rama **Else** (en rojo) es opcional y se ejecuta si ninguna de las condiciones se cumple.

Condicionales: Estructura Select

```
Select expresion
Case listavalores
   Instrucciones
Case listavalores
   Instrucciones
...
Case Else
   Instrucciones
End Select
```

- Puede haber tantas ramas **Case** como deseemos.
- La rama **Case Else** (en rojo) es opcional.

Condicionales: Estructura Select

```
Select expresion
Case listavalores
   Instrucciones
Case listavalores
   Instrucciones
...
Case Else
   Instrucciones
End Select
```

- Se evalúa la expresión.
- Se mira si el valor de la expresión coincide con alguno de los valores de una rama **Case**.
- Si es así se ejecutan las instrucciones de esa rama.
- Si no, se ejecutan las instrucciones de **Case Else**.

Condicionales: Estructura Select

```
Select expresion
Case listavalores
   Instrucciones
Case listavalores
   Instrucciones
...
Case Else
   Instrucciones
End Select
```

- La expresión debe ser de tipo **Boolean**, **Byte**, **Char**, **Date**, **Double**, **Decimal**, **Integer**, **Long**, **Object**, **Short**, **Single** o **String**

Condicionales: Estructura Select

```
Select expresion
Case listavalores
   Instrucciones
Case listavalores
   Instrucciones
...
Case Else
   Instrucciones
End Select
```

- La lista de valores es una serie de expresiones separadas por comas.

Mail: palasi@palasi.com Web: www.palasi.com

Las expresiones que hay en la lista de valores

- Pueden ser:
- Una expresión.
- ***expresion1*** `To` ***expresion2*** (comprende todos los valores entre estos dos literales).
- `Is` `>` ***expresion*** (donde pone >, puede haber cualquier operación de comparación =, <>, <, <=, > o >=)

Ejemplo

```
Select monto
Case Is < 0
 MessageBox.Show("Monto negativo")
Case 0
 MessageBox.Show("Monto nulo")
Case 1,2
 MessageBox.Show("Monto uno o dos")
Case 3 To 10
 MessageBox.Show("Monto de 3 a 10")
Case Else
 MessageBox.Show("Mayor que 10")
End Select
```

Ciclos: Estructura `Do..While`

```
Do While condición
  Instrucciones
Loop
```

O bien

```
Do
  Instrucciones
Loop While condicion
```

- Se repiten las instrucciones mientras se cumple la condición.
- La condición puede ser evaluada antes o después de ejecutar las instrucciones.

Ciclos: Estructura `Do..Until`

```
Do Until condición
  Instrucciones
Loop
```

O bien

```
Do
  Instrucciones
Loop Until condicion
```

- Se repiten las instrucciones hasta que se cumple la condición.
- La condición puede ser evaluada antes o después de ejecutar las instrucciones.

Ciclos: Estructura `For..Next`

```
For contador = valor1 To valor2
  Instrucciones
Next contador
```

- Repite las instrucciones para todos los valores del contador desde valor1 a valor2.
- El valor del contador se incrementa en 1 cada vez. Si queremos otro incremento, se puede usar.

```
For contador = valor1 To valor2 Step inc
  Instrucciones
Next contador
```

Ciclos: Estructura `For Each ..Next`

- No lo veremos por ahora.
- Ejecuta las instrucciones del ciclo para cada elemento de una colección.

4. Otros conceptos básicos de VB.NET

Estructuras de datos

- Con las variables que hemos visto hasta ahora sólo podíamos guardar un valor u objeto para cada variable.

1 variable ⇔ 1 valor

- El valor debía ser **único**.
- Esto no es práctico cuando queremos tratar grandes conjuntos de datos de la misma forma.

Tratar muchos datos de la misma forma

- Por ejemplo, tenemos que obtener la planilla y a todos los empleados debemos calcular el sueldo.
- Con el esquema hasta ahora, deberíamos tener una variable por empleado, pero esto no es práctico. La programación se hace insufrible.
- Para ello, aparecen las estructuras de datos que son **variables que pueden contener un conjunto de datos**.

Estructuras de datos

- Variables que pueden contener un conjunto de datos.
- Las más básicas son los arreglos (también llamadas, arrays, matrices, tablas).
- Las veremos a continuación.

Concepto de arreglo

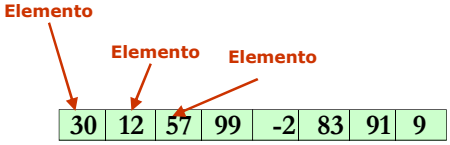
- Un arreglo es una variable que permite guardar una colección ordenada de datos del mismo tipo.
- Puntos importantes:
  - como es una variable, tiene un nombre
  - la colección es ordenada.
  - los datos deben ser todos del mismo tipo.

30	12	57	99	-2	83	91	9

Variable arreglo `tabla` (con enteros)

Un arreglo es similar a un conjunto de variables

- Cada dato se guarda en algo análogo a una variable que se llama elemento
- Como es parecido a una variable, podemos leer y escribir su valor.

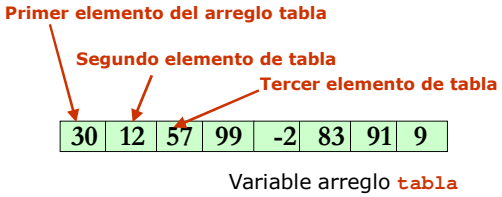


Variable arreglo `tabla`

Mail: palasi@palasi.com Web: www.palasi.com

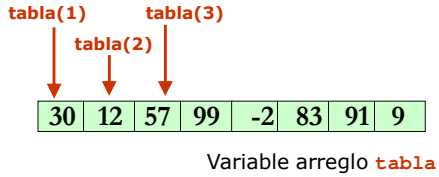
Escribiendo o leyendo un valor en un elemento de un arreglo

- Debemos decir qué elemento de todos queremos escribir o leer.
- Podríamos indicarlo con lenguaje natural, es decir, de la siguiente forma.



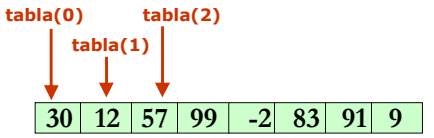
Pero esto es demasiado largo

- En vez de esto se utiliza la notación *nombreArreglo(numeroElemento)*
- Podemos denominar:



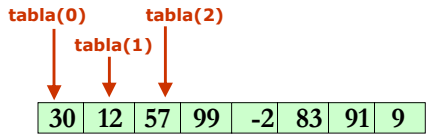
De hecho, no es exactamente así

- En VB.NET, los elementos del arreglo **comienzan a contarse desde cero**
- Así:



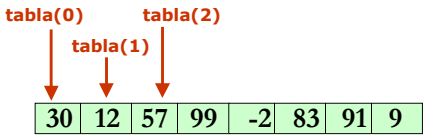
Leyendo un valor en un elemento de un arreglo

- Dos formas. Como una variable:
  nombreArreglo(numeroElemento)
- Como un objeto (preferible).
  nombreArreglo`.GetValue(`*numeroElemento*`)`



Escribiendo o leyendo un valor en un elemento de un arreglo

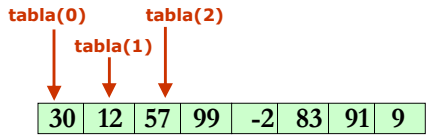
- Dos maneras. Como una variable:
  nombreArreglo(numeroElemento) = expresión
- En forma de objeto (preferible),
  nombreArreglo`.SetValue(`*expresion, numeroElemento*`)`



Escribiendo o leyendo un valor en un elemento de un arreglo

- Para leer (recuperar) el valor
  `tabla(1)` o `tabla.GetValue(1)`, por ejemplo, `MessageBox.Show(tabla(1))`
- Para escribir (asignar) el valor

```
tabla(1) = 30
tabla.SetValue(30,1)
```



Índice

- Es el número de elemento (o de posición) con el que accedemos a los datos.
- Aparece entre paréntesis. `tabla(5)`

índice

`tabla(0)` `tabla(2)` `tabla(1)`

0	1	2	3	4	5	6	7
30	12	57	99	-2	83	91	9

Variable arreglo `tabla`

La longitud de un arreglo

- Es el número de elementos que tiene.
- Hay dos tipos de arreglos en VB.NET respecto a la longitud:
- Arreglos estáticos. Tienen longitud fija. Una vez definida no es puede cambiar.
- Arreglos dinámicos. Tienen longitud variable. Se puede cambiar el número de elementos.

30	12	57	99	-2	83	91	9

Variable arreglo `tabla`

Tipos de arreglo

- Como hemos dicho, un arreglo tiene todos sus elementos de un mismo tipo.
- Así, hay arreglos de enteros, de **`doubles`**, de **`String`**, de objetos **`Empleado`**, de objetos de cualquier tipo.
- Lo que no existe es un arreglo que pueda mezclar diferentes tipos de elementos (así, enteros con **`String`**)

Definición de un arreglo estático

- Depende del tipo de sus elementos

```
Dim varArreglo(numeroElementos) As tipoElementos
```

- Así, por ejemplo,

```
Dim tabla(8) As Integer
Dim nombres(20) As String
Dim planilla(1000) As Empleado
```

Arreglo de mil objetos empleado

Definición de un arreglo estático inicializando el arreglo

- Se puede indicar los elementos del arreglo cuando se define

```
Dim varArreglo() As tipoElementos = {listaElementos}
```

- Como se ve, no hace falta indicar el número de elementos, porque el arreglo se ajusta automáticamente

```
Dim tabla() As Integer ={3,1,11,2}
```

Definición de un arreglo dinámico

- Es parecido al estático pero no se especifica el número de elementos

```
Dim varArreglo() As tipoElementos
```

- Cada vez que queramos cambiar la dimensión del arreglo se hace

```
ReDim varArreglo(numeroElementos)
```

Con éste se redimensiona pero se pierde el contenido anterior del arreglo.

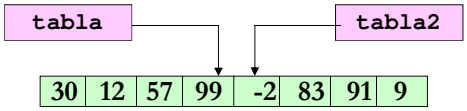
```
ReDim Preserve varArreglo(numeroElementos)
```

Con éste se redimensiona y se conserva el contenido anterior del arreglo.

Los arreglos son tipos de referencia

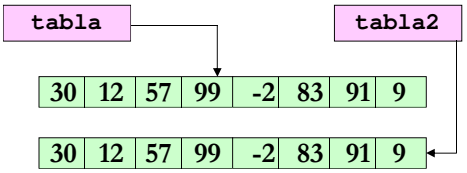
- Por ello, para copiarlos no basta con copiar sus variables referencia.

```
tabla2 = tabla
'Copia la referencia pero sigue
'siendo el mismo arreglo.
```



Copiando arreglos

```
tabla.CopyTo(tabla2,0)
```



En general

arreglo1`.CopyTo(`*arreglo2*`,`*índice*`)`

- ***arreglo1*** es el arreglo que se desea copiar.
- ***arreglo2*** es el arreglo donde se desea copiar.
- ***índice*** es la posición en la que se empezará a copiar.

Arreglos de varias dimensiones

- A veces, necesitamos arreglos de varias dimensiones que pueden accederse por varios índices.

Variable arreglo `coord`

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

Leyendo en arreglos de varias dimensiones

- Como ejemplo, dos dimensiones. Para leer el elemento de fila 3 y columna 2:

```
coord(3,2)
coord.GetValue(3,2)
```

Variable arreglo `coord`

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

Escribiendo en arreglos de varias dimensiones

- Para escribir un 0 en el elemento de fila 3 y columna 2:

```
coord(3,2) = 0
coord.SetValue(0,3,2)
```

Variable arreglo `coord`

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

La notación `SetValue` y `GetValue` sólo hasta arreglos de 3 dimensiones

- Para arreglos de más dimensiones sólo se utiliza la notación en forma de variable.
- Por ejemplo, supongamos que tenemos un arreglo de 4 dimensiones llamado **`arreglo`**.
- Podemos escribir estas instrucciones.

```
MessageBox.Show(arreglo(4,1,3,10))
arreglo(4,1,3,10) = 0
```

- Pero no podemos usar **`GetValue`** y **`SetValue`**.

Definición de un arreglo de varias dimensiones

- Arreglo estático:

```
Dim varArreglo(numElDim1,..., numElDimn) As tipoElementos
```

- Arreglo dinámico:

```
Dim varArreglo(,,,) As tipoElementos
```

(tantas comas como dimensiones menos una).

- Arreglo dinámico

```
ReDim [Preserve] varArreglo (numElDim1,..., numElDimn)
```

La longitud de un arreglo

- Es el número de elementos que tiene. En este caso, 8.

30	12	57	99	-2	83	91	9

- Si el arreglo es de varias dimensiones, cada dimensión tiene su longitud (en este caso 8 y 4):

30	12	57	99	-2	83	91	9
65	79	40	64	36	81	19	7
30	12	57	99	-2	83	91	9
30	12	57	99	-2	83	91	9
30	12	57	99	-2	83	91	9

Para conseguir la longitud

```
arreglo.GetLength(numDimension)
```

- La primera dimensión tiene el número 0, la segunda el número 1 y así sucesivamente.
- Por ejemplo, si el arreglo **`tabla`** tiene una única dimensión, su longitud se consigue con:

```
tabla.GetLength(0)
```

Pregunta

- Si

```
tabla.GetLength(0)
```

devuelve 8, la longitud del arreglo es de 8.

¿Cuáles serán los índices válidos del arreglo?

- De 1 a 8
- De 0 a 7
- De 0 a 8

Respuesta

- De 0 a 7.
- Los arreglos comienzan en 0 y, por lo tanto, su último índice es siempre la longitud-1.
- Esto hay que tenerlo en cuenta al programar. Sobre todo, cuando se hacen bucles sobre el arreglo.

La clase `System.Array`

- Clase predefinida que nos da una serie de métodos útiles para trabajar con arreglos.

Algunos métodos de `System.Array`

`Clear` (*arreglo*, *indice*, *longitud*)	Fija un rango de elementos a cero o a `Nothing`
`Sort` (*arreglo*)	Ordena los elementos en un arreglo de 1 dimensión.
`BinarySearch` (*arreglo*, *indice*, *longitud*, *valor*)	Busca un valor en un arreglo ordenado de 1 dimensión.
`IndexOf` (*arreglo*, *valor*)	Busca la primera ocurrencia de un valor en un arreglo
`LastIndexOf` (*arreglo*, *valor*)	Busca la última ocurrencia de un valor en un arreglo

Arreglos: unos objetos extraños

- Los objetos arreglo tienen **sintaxis diferente** de los restantes objetos VB.NET para algunas operaciones.
- Esta sintaxis es reminiscencia de otros lenguajes.
- Son objetos, pero extraños.
- Podemos verlos como objetos o como algo diferente, según nos convenga.

Ejercicio

- Creen un programa que modele una factura de consumidor final. Por simplicidad supondremos que todas las ventas son gravadas con 13%.
- De cada factura, nos interesa el nombre del cliente, la fecha, el total de ventas antes de IVA, el IVA y el total de ventas con IVA. También nos interesan las diferentes líneas de la factura. Cada línea tiene una cantidad, nombre de un producto, precio unitario y precio total.
- Se podrán añadir líneas a una factura, añadir nuevas facturas.
- Recuerden de aplicar los conceptos de orientación a objetos.

Ejercicio

- Crear un programa que simule un diccionario de traducción español-inglés. Un diccionario es un conjunto de entradas y cada entrada es un par de palabras en español y su traducción en inglés.
- Se pueden añadir entradas al diccionario, consultarlas.
- También se necesita que, dada una frase en español que está escrita en un arreglo (una palabra por posición) se nos devuelva su traducción en inglés. Pista: creen un objeto Frase.

4. Otros conceptos básicos de VB.NET

Mail: palasi@palasi.com Web: www.palasi.com

Veamos una clase que modela un cheque (muy simplificada)

```
Public Class Cheque
  Public Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Supongamos que tenemos un objeto cheque1 de esta clase

```
Public Class Cheque
  Public Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Da el mismo resultado poner

```
cheque1.Monto
cheque1.ConsigueMonto()
```

Supongamos que tenemos un objeto cheque1 de esta clase

```
Public Class Cheque
  Public Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Da el mismo resultado poner

```
cheque1.Monto = expresión
cheque1.FijaMonto(expresión)
```

Podríamos prohibir el acceso a atributos desde fuera de la clase

```
Public Class Cheque
  Private Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Ahora sólo se puede acceder al monto a través de métodos

```
cheque1.ConsigueMonto()
cheque1.FijaMonto(expresión)
```

Podríamos prohibir el acceso a atributos desde fuera de la clase

```
Public Class Cheque
  Private Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Ahora sólo se puede acceder al monto a través de métodos

```
cheque1.Monto
cheque1.Monto = expresión
```

¡¡PROHIBIDO!!

Se dice que la clase tiene el estado encapsulado

- **Una clase con el estado encapsulado es aquella que sus atributos son privados y son accesibles sólo a través de métodos.**
- La ventaja es que podemos implementar controles a la modificación de los datos.
- Por ejemplo, se puede comprobar que no se entre un monto negativo.
- De la otra manera, otra clase podría asignarnos un valor negativo y la clase no se podría proteger.

No podemos introducir montos no positivos

```
Public Class Cheque
  Private Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
   If NuevoMonto <=0 Then
    MessageBox.Show("Monto debe ser positivo")
   Else
     Monto = NuevoMonto
   End If
  End Sub
End Class
```

Esto no era posible en una clase con el estado no encapsulado

```
Public Class Cheque
  Public Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Como el atributo era público, podíamos poner un monto negativo sin ningún control. Esto era peligroso. Encapsulando el estado tenemos formas de evitarlo.

Otra ventaja

```
Public Class Cheque
  Private Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

Simplifica el mantenimiento. Si queremos que el monto se lea desde una base de datos en vez de un atributo, podemos hacer lo siguiente

Accediendo al monto desde una base de datos

```
Public Class Cheque
  Public Function ConsigueMonto() As Integer
     InstruccionesConsentenciasSQL
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     InstruccionesConsentenciasSQL
  End Sub
End Class
```

Como las otras clases accedían a esta por los métodos y las cabeceras de estas no han variado, no debemos cambiar nada del programa debido a los cambios de esta clase.

Se simplifica el mantenimiento.

Cuando un estado está encapsulado hay dos métodos para cada atributo

```
Public Class Cheque
  Private Monto As Integer
  Public Function ConsigueMonto() As Integer
     Return Monto
  End Function
  Public Sub FijaMonto(NuevoMonto As Integer)
     Monto = NuevoMonto
  End Sub
End Class
```

- Uno para consultarlo y otro para modificarlo. En este caso, los métodos son `ConsigueMonto` y `FijaMonto`.

¿Cuándo hay que encapsular el estado?

- En general, hay que encapsular (casi) siempre.
- Dejar acceso a los atributos de una clase sin ningún control es peligroso. Es mucho mejor que el acceso se haga a través de métodos que controlen anomalías.
- Es como si a una casa se le pone una puerta. Se controla el acceso. De otra forma, es peligroso.
- Además encapsular el estado simplifica el mantenimiento.

Sin embargo, el estado encapsulado tiene un pequeño inconveniente

- Es engorroso sintácticamente. Poner

```
cheque1.ConsigueMonto()
cheque1.FijaMonto(expresión)
```

- Es mucho más incómodo y confuso que sus equivalentes con estado no encapsulado:

```
cheque1.Monto
cheque1.Monto = expresión
```

- Afortunadamente, VB .NET nos provee de un mecanismo con el que podemos encapsular el estado sin esta inconveniencia sintáctica.

Este mecanismo se llama "Propiedades"

- Una propiedad es
  - un par de métodos que acceden a un atributo privado para conseguir encapsulación del estado.
  - Pero que nos permiten usar una sintaxis parecida a los atributos para usarlos.
- Dicho de forma más clara:
  - Un par de métodos que acceden a un atributo privado y que se usan como si fueran un atributo público.

Sintaxis para definir una propiedad

```
ámbito Property nombre() As tipo
    Get
        método para consultar atributo privado
    End Get
    Set(ByVal Value As tipo)
        método para modificar atributo privado
    End Set
End Property
```

- En la parte **Set** el valor nuevo que se pasa, se indica con un parámetro que, por convención se llama **Value**, aunque no tiene porqué ser así.

Sintaxis para definir una propiedad

```
ámbito Property nombre() As tipo
    Get
        método para consultar atributo privado
    End Get
    Set(ByVal Value As tipo)
        método para modificar atributo privado
    End Set
End Property
```

- El ámbito no debe ser **Private** (no tendría sentido). Pero puede ser todos los otros ámbitos.

Ejemplo

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property Monto() As Integer
     Get
       Return MontoPrivado
     End Get
     Set(ByVal Value As Integer)
       MontoPrivado=Value
     End Set
  End Property
End Class
```

¿Qué ventaja tiene esto?

- En principio sólo parece una forma de expresar los mismos métodos que encapsulaban el estado, pero con una sintaxis diferente.
- En realidad, la ventaja es que ahora se puede acceder a esta propiedad como si fuera un atributo, lo que es más intuitivo.

Supongamos que tenemos un objeto cheque1 que es de clase Cheque

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property Monto() As Integer
     Get
       Return MontoPrivado
     End Get
     Set(ByVal Value As Integer)
       MontoPrivado=Value
     End Set
  End Property
End Class
```

Podemos escribir

```
cheque1.Monto = 100
MessageBox.Show(cheque1.Monto)
```

Es decir, podemos acceder a una propiedad como si fuera un atributo

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property Monto() As Integer
     Get
       Return MontoPrivado
     End Get
     Set(ByVal Value As Integer)
       MontoPrivado=Value
     End Set
  End Property
End Class
```

Esto es más sencillo, tanto a nivel sintáctico como conceptual.

Podemos escribir

```
cheque1.Monto = 100
MessageBox.Show(cheque1.Monto)
```

Pero no es realmente un atributo sino un par de métodos

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property Monto() As Integer
     Get
       Return MontoPrivado
     End Get
     Set(ByVal Value As Integer)
       MontoPrivado=Value
     End Set
  End Property
End Class
```

Tenemos todo el control del estado encapsulado.

Podemos escribir

```
cheque1.Monto = 100
MessageBox.Show(cheque1.Monto)
```

Por ejemplo, ahora podemos escribir

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property Monto() As Integer
     Get
       Return MontoPrivado
     End Get
     Set(ByVal Value As Integer)
       If Value <=0 Then
          MessageBox.Show("Monto no es positivo")
       Else
          MontoPrivado = Value
       End If
     End Set
  End Property End Class
```

Con las propiedades tenemos estado encapsulado

- Y toda su ventaja de control sobre el acceso a los atributos.
- Sin la desventaja de la sintaxis incómoda.
- A la hora de programarla, pensamos en una propiedad como un par de métodos.
- A la hora de usarla, pensamos en la propiedad como un atributo.

En general

- Se recomienda no dejar ningún atributo público.
- Es decir, hacer los atributos privados y definir una propiedad para acceder a ellos.
- Así tenemos todas las ventajas que no da el estado encapsulado.

Mail: palasi@palasi.com Web: www.palasi.com

Propiedades de sólo lectura

- A veces necesitamos que un atributo privado pueda ser consultado, pero no pueda ser modificado (al menos, de forma directa).
- Para eso, podemos definir una propiedad de sólo lectura.
- Una propiedad de sólo lectura sólo tiene la parte **Get** y, por lo tanto, sólo puede leerse.

Sintaxis para definir una propiedad de sólo lectura

```
ámbito Property ReadOnly nombre() As tipo
    Get
        método para consultar atributo privado
    End Get
End Property
```

- **Esta propiedad sólo se podrá leer pero no asignar**

Ejemplo

```
Public Class Cheque
  Private MontoPrivado As Integer
  Public Property ReadOnly Monto() As Integer
     Get
       Return MontoPrivado
     End Get
  End Property
End Class
```

Esta propiedad se puede leer:

```
MessageBox.Show (cheque1.Monto)
```

OK

Pero no se puede escribir

```
cheque1.Monto = 100
```

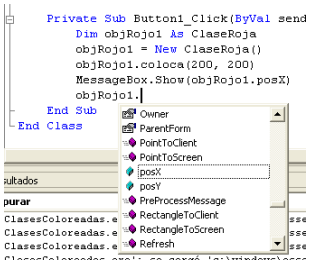
NO

El concepto de miembro

- Hasta ahora un miembro era un atributo o un método.
- Sin embargo, las propiedades no son más que métodos, así que es lógico incluirlos en el mismo concepto.
- A partir de ahora, los miembros de una clase serán:
  - **Sus atributos.**
  - **Sus métodos.**
  - **Sus propiedades.**

Cómo conocer las propiedades de un objeto

- Los atributos públicos aparecen con el icono
- Los métodos públicos aparecen con el icono
- Las propiedades aparecen con el icono



Ejercicio

- Rehacer el ejercicio anterior de las facturas encapsulando el estado de todos los objetos mediante propiedades.

4. Otros conceptos básicos de VB.NET

- 4.1. Variables locales
- 4.2. Tipos de datos en VB.NET
- 4.3. Paso de parámetros
- 4.4. Estructuras de control
- 4.5. Arreglos.
- 4.6. Propiedades.
- 4.7. Concepto de interfaz e implementación.

Interfaz e implementación

- Interfaz: Todo lo que puede accederse desde fuera del objeto.
- Implementación: todo aquello que NO puede accederse desde fuera del objeto.

Interfaz e implementación

- Interfaz: Atributos públicos, métodos públicos y propiedades (públicas).
- Implementación: Atributos privados, métodos privados y el cuerpo de los métodos.

Ocultación de la implementación

Sólo se que `mueveDerecha` mueve a la derecha, pero no sé cómo lo hace.

- Utilizamos los objetos sin saber cómo están implementados (sabemos QUÉ hacen, pero no el CÓMO)
- Esto simplifica la programación y el mantenimiento.
- Se puede mejorar la implementación sin cambiar la interfaz del objeto: ***refactoring***

Temario del curso

- 1. Introducción a .NET y Visual Basic .NET
- 2. Un primer programa en VB .NET
- 3. Introducción a la programación orientada a objetos.
- 4. Otros conceptos básicos de VB.NET.
- 5. Más sobre la programación orientada a objetos con VB.NET.
- 6. Conclusión. Los siguientes pasos.

5. Más sobre la programación orientada a objetos con VB.NET

- En el punto 3, habíamos visto los conceptos básicos de la programación orientada a objetos: clases, objetos, atributos y métodos.
- De hecho, hay más conceptos fundamentales, pero nos habíamos detenido un poco para asimilar estos conceptos básicos, viendo otras cosas.
- Ahora vamos a reanudar nuestro estudio de la programación orientada a objetos y cómo ésta se produce en VB.NET.

Mail: palasi@palasi.com Web: www.palasi.com

5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (**Shared**)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

palasi.com
IT and management

5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (**Shared**)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

El método MessageBox.Show

Se puede usar en muy diferentes situaciones. Pruébenlo.

```
MessageBox.Show("Hola a todos")
//Recibe un único parámetro
MessageBox.Show("Hola a todos","VB.NET")
//Recibe dos parámetros
MessageBox.Show("Hola a todos","VB.NET",
  MessageBoxButtons.OK,
  MessageBoxIcon.Question)
//Recibe cuatro parámetros.
```

El método MessageBox.Show

Parece que este método puede recibir muchos tipos de parámetros diferentes (en incluso ninguno).

```
MessageBox.Show("Hola a todos")
//Recibe un único parámetro
MessageBox.Show("Hola a todos","VB.NET")
//Recibe dos parámetros
MessageBox.Show("Hola a todos","VB.NET",
  MessageBoxButtons.OK,
  MessageBoxIcon.Question)
//Recibe cuatro parámetros.
```

palasi.com
IT and management

Esto es práctico

- Ya que, en caso contrario, tendríamos que tener un método para cada tipo de parámetro. Más difícil de recordar.

```
MessageBox.Show("Hola a todos")
MessageBox.ShowConTitulo("Hola a
  todos","VB.NET")
MessageBox.ShowConTituloBotonIcono(
  "Hola a todos","VB.NET",
  MessageBoxButtons.OK,
  MessageBoxIcon.Question)
```

palasi.com
IT and management

A esto se le llama sobrecarga ("overloading")

- Es cuando un mismo método puede recibir:
  - parámetros de un tipo diferente
  - un número diferente de parámetros.

```
MessageBox.Show("Hola a todos")
MessageBox.Show("Hola a
  todos","VB.NET")
MessageBox.Show("Hola a
  todos","VB.NET",
  MessageBoxButtons.OK,
  MessageBoxIcon.Question)
```

La sobrecarga es útil en multitud de situaciones

- Supongamos que implementamos un método que busca una cadena dentro de otra y especifica la posición en que se encuentra.

```
Function BuscaCadena(ByVal cadena1
  As String, ByVal cadena2 As
  String) As Integer
```

```
BuscaCadena("Consolar","sol")
```
devuelve 3
```
BuscaCadena("Consolar","t")
```
devuelve –1 (no se encuentra)

La sobrecarga es útil en multitud de situaciones

- Ahora queremos que busque a partir de una determinada posición.

```
Function BuscaCadenaPos(ByVal
  cadena1 As String, ByVal cadena2
  As String, ByVal posInicial As
  Integer) As Integer
```

```
BuscaCadenaPos("Consolar","sol",0)
```
devuelve 3
```
BuscaCadenaPos("Consolar", "sol", 4)
```
devuelve -1 (no hay a partir de posición 4)

La sobrecarga es útil en multitud de situaciones

- Sería útil que **buscaCadenaPos** y **buscaCadena** se escribiera de la misma forma. Más fácil de recordar.
- Si no se le pasa posición, se asume que es la posición 0. Así:

```
BuscaCadena("Consolar","sol")
```
devuelve 3.
```
BuscaCadena("Consolar", "sol", 4)
```
devuelve -1.

Otro ejemplo

- En **`System.Array`** teníamos un método **`Sort`** que ordenaba un arreglo unidimensional y que tenía la siguiente sintaxis:

`Sort` (***arreglo***)

- Además, podemos ordenar una parte del arreglo con esta sintaxis

`Sort` (***arreglo, inicio, longitud***)

- Esto es sobrecarga.

¿Cómo se declaran métodos con sobrecarga?

- Hasta ahora hemos utilizado algunos métodos sobrecargados ya definidos.
- Pero nos interesa saber cómo definir nuestros propios métodos sobrecargados.

Cómo se declaran métodos sobrecargados

- Los métodos sobrecargados, se declaran como varios métodos diferentes con el mismo nombre y diferente lista de parámetros.
- "Método sobrecargado" es un abuso de lenguaje. En realidad, son varios métodos con el mismo nombre.
- Para cada uno de los métodos individuales, se le coloca la palabra clave **`Overloads`** entre el ámbito y la palabra **`Function`** o **`Sub`**.

En el caso anterior

```
Public Class UtilesCadena
  Public Overloads Function BuscaCadena(ByVal
   cadena1 As String, ByVal cadena2 As String) As
   Integer
              Aquí el cuerpo de la función
  End Function

 Public Overloads Function BuscaCadena(ByVal
  cadena1 As String, ByVal cadena2 As String,
  ByVal posInicial As Integer) As Integer
           Aquí el cuerpo de la función
 End Function
End Class
```

¿Cómo funciona esto?

- Si ejecutamos
  `utiles1.BuscaCadena("hola","ho")`
  VB detecta que se han pasado dos parámetros cadena y, por tanto, ejecuta el primer método.

- Si ejecutamos
  `utiles1.BuscaCadena("hola", "ho",3)`
  VB detecta que se han pasado dos parámetros cadena y un entero y, por tanto, ejecuta el segundo método.

`Overloads` es opcional

- Podíamos haber prescindido de él. Así.

```
Public Class UtilesCadena
  Public Function BuscaCadena(ByVal
   cadena1 As String, ByVal cadena2 As String) As
   Integer
              Aquí el cuerpo de la función
  End Function

 Public Function BuscaCadena(ByVal cadena1 As
  String, ByVal cadena2 As String, ByVal
  posInicial As Integer) As Integer
           Aquí el cuerpo de la función
 End Function
End Class
```

Lo que no se puede hacer es `Overloads` en una y en otra no

- Esto da error de compilación.

```
Public Class UtilesCadena
  Public Overloads Function BuscaCadena(ByVal
   cadena1 As String, ByVal cadena2 As String) As
   Integer
              Aquí el cuerpo de la función
  End Function

 Public Function BuscaCadena(ByVal cadena1 As
  String, ByVal cadena2 As String, ByVal
  posInicial As Integer) As Integer
           Aquí el cuerpo de la función
 End Function
End Class
```

La sobrecarga no es sólo por el número de parámetros, sino por el tipo

```
Public Class EscriturasVarias
  Public Overloads Sub EscribeTipo(ByVal
   parametro As String)
     MessageBox.Show("El parámetro es " & _
      parametro & " y es de tipo String ")
  End Sub

  Public Overloads Sub EscribeTipo(ByVal
  parametro As Integer)
     MessageBox.Show("El parámetro es " & _
      parametro & " y es de tipo Integer ")
 End Function
End Class
```

¿Cómo funciona esto?

- Si ejecutamos
  `escrituras1.EscribeTipo("hola")`
  VB detecta que el parámetro pasado es de tipo **`String`** y, por tanto, ejecuta el primer método.

- Si ejecutamos
  `escrituras1.EscribeTipo(5)`
  VB detecta que el parámetro pasado es de tipo **`Integer`** y, por tanto, ejecuta el segundo método.

Resumen

- Los métodos sobrecargados tienen el mismo nombre pero diferentes numero y/o tipos de parámetros.
- A partir de los parámetros que le pasamos, VB.NET identifica qué método debe ejecutar.

Pregunta

- ¿Los siguientes métodos son sobrecargados?

```
Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Integer

Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Double
```

Respuesta

- ¿Los siguientes métodos son sobrecargados?

```
Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Integer

Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Double
```

Respuesta

```
Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Integer

Public Overloads Function
  Duplica(ByVal numero As Integer) As
  Double
```

- No lo están, pues a pesar de tener el mismo nombre tienen el mismo tipo de parámetros.
- Más importante, VB.NET no compilaría estos dos métodos si estuvieran en la misma clase

¿Por qué no compila?

```
Public Overloads Function
  Duplica(ByVal numero As Integer)
  As Integer

Public Overloads Function
  Duplica(ByVal numero As Integer)
  As Double
```

- Si hiciéramos

```
objeto.Duplica(6)
```

- ¿cómo sabe VB.NET cuál de los dos métodos usar?

Una regla de VB.NET

- Dos métodos de una misma clase no pueden tener al mismo tiempo:
  - El mismo nombre
  - El mismo tipo de parámetros.

Dicho de otra manera

- Los métodos de una misma clase deben tener:
  - O bien distinto nombre (métodos diferentes).
  - O bien distintos tipos de parámetros (métodos sobrecargados).

Ejercicio

- Programar una clase **Alumno**. De cada alumno, nos interesa su nombre, su apellido, su número de teléfono y su nota (que es un entero entre 0 y 10).
- Necesitamos dos métodos diferentes (que pueden ir sobrecargados). Uno para la inscripción y otro para poner nota.
- Para la inscripción, el único dato necesario es el apellido.
- La nota se puede introducir en número o en letra.

5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (**Shared**)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

Examinen la siguiente clase

```
Public Class LibretaBanco
  Private numero,saldo As Integer
  Public Sub Inicia(ByVal numeroLib As Integer,
   ByVal saldoIni As Integer)
     Me.numero = numeroLib
     Me.saldo = saldoIni
  End Sub
  Public Sub AñadeMovimiento
    (ByVal monto As Integer)
     Me.saldo = Me.saldo + monto
  End Sub
  Public Function RecuperaSaldo() As Integer
     Return Me.saldo
  End Sub
End Class
```

La clase *LibretaBanco*

- Consta de tres métodos:
- **Inicia** asigna el número de libreta y el saldo inicial.
- **AñadeMovimiento** añade un movimiento a la libreta (depósitos positivos, retiros negativos)
- **RecuperaSaldo** nos da el saldo de la libreta.
- El estado está encapsulado para prevenir que errores de programación amenacen la integridad de los atributos

Uso de la clase *LibretaBanco*

```
Dim libreta As LibretaBanco
libreta = New LibretaBanco()//Crea
libreta.inicia(134,500) //Inicializa
libreta.añadeMovimiento(200)
libreta.añadeMovimiento(-124)
...
MessageBox.Show("El saldo es" & _
  libreta.recuperaSaldo())
```

Problemas de la clase *LibretaBanco* (1)

```
Dim libreta As LibretaBanco
libreta = New LibretaBanco()//Crea
libreta.inicia(134,500) //Inicializa
libreta.añadeMovimiento(200)
libreta.añadeMovimiento(-124)
...
MessageBox.Show("El saldo es" & _
  libreta.recuperaSaldo())
```

- Para crear la libreta hay que hacer dos instrucciones: crearla e inicializarla, lo que resulta poco práctico.

Problemas de la clase *LibretaBanco* (2)

```
Dim libreta As LibretaBanco
libreta = New LibretaBanco()//Crea
libreta.inicia(134,500) //Inicializa
libreta.añadeMovimiento(200)
libreta.añadeMovimiento(-124)
libreta.inicia(122,0)
```

- En cualquier momento, se puede inicializar la libreta de nuevo, lo cual es peligroso, pues se pierde la información de los movimientos anteriores.

Causas de estos problemas

- Tenemos dos operaciones:
  - Una para crear la libreta (**New**).
  - Una para darle sus datos iniciales (**Inicia**).
- No está bien modelado. En un banco, el número de una libreta y su saldo inicial sólo se determinan al momento de la creación **y no se pueden cambiar.**
- En nuestro caso, sí se pueden cambiar y esto puede dar a errores de programación muy peligrosos.

La solución a este problema

- Sería que los datos iniciales de la libreta sólo se pudieran asignar en el momento de la creación.
- O sea, tener una única operación que fuera la suma de **New+inicia**
- Podría escribirse así:

```
libreta = New
  LibretaBanco(134,500)
```

Un poco de terminología

- A este método que ha surgido de la suma de **New+inicia** y que se ejecuta así:

```
libreta = New
  LibretaBanco(134,500)
```

se le llama **constructor**. 1ª definición:

- **Constructor** es un tipo especial de método que determina como se inicializa un objeto **cuando se crea**.

Cómo se ejecutan los constructores en VB .NET

- La llamada al constructor de **LibretaBanco** sería así:

```
libreta = New
  LibretaBanco(134,500)
```

- En general, la llamada es de este estilo:

New ***NombreClase****(listaparametros)*

Mail: palasi@palasi.com Web: www.palasi.com

Un constructor no es un método normal

- Los métodos normales se aplican sobre un objeto preexistente.
- Los constructores crean el objeto y después realizan el proceso que sea con este objeto recién creado.

- Es por eso que la llamada a un método normal tiene la sintaxis.

```
libreta.inicia(134,500)
```

- La llamada a un constructor tiene la sintaxis.

```
libreta = New LibretaBanco(134,500)
```

Observación

- Fíjense que ahora no podemos inicializar varias veces **LibretaBanco**, ya que el método para inicializar ya no se encuentra disponible en el objeto, sino sólo cuando este se crea.

- Esto aumenta la robustez de nuestro programa y nos previene de posibles errores de programación.

Hasta ahora hemos visto como se ejecutan los constructores

- Sin embargo, para programarlos necesitamos saber cómo se declaran los constructores.

- Esto lo veremos a continuación.

Como se declaran los constructores

Un método normal se declara así:

```
ámbito Sub nombre(listaparam)

End Sub
```

Un constructor se declara con el nombre **New**:

```
ámbito Sub New (listaparam)

End Sub
```

- Los elementos en negro son opcionales.

Ejemplo: constructor de LibretaBanco

```
Public Sub New(ByVal numeroLib As
  Integer,ByVal saldoIni As Integer)
   Me.numero = numeroLib
   Me.saldo = saldoIni
End Sub
```

Recuerden que este constructor no sólo asigna atributos sino que **crea el objeto.**

Este constructor se llamaría de la siguiente forma.

```
libreta1 = New Libreta(123,500)
```

Ejercicio

- Creen una clase **UsuarioBanco** con tres atributos: nombre, cédula y dirección. El nombre y la cédula sólo se podrán asignar cuando se cree el objeto y se deberán asignar en ese mismo momento. En cambio, la dirección deberá poder modificarse siempre. El estado deberá estar encapsulado usando una propiedad.

- Creen un formulario que manipule esa clase.

Mail: palasi@palasi.com Web: www.palasi.com

Pregunta

- En el formulario, intenten ejecutar esta línea de código.

```
usuario1 = New UsuarioBanco()
```

- ¿Qué pasa?

Da un error de compilación

No se ha especificado ningún argumento para el parámetro 'cedula' de 'Public Sub New(nombre As Integer, cedula As Integer)'.
No se ha especificado ningún argumento para el parámetro 'nombre' de 'Public Sub New(nombre As Integer, cedula As Integer)'.

- Dice que faltan parámetros.
- Es obvio que está intentando ejecutar el constructor que hemos definido, al cual se pasa el nombre y la cédula.
- ¿Qué pasó con la opción

```
New UsuarioBanco()
```

que hasta ahora estaba disponible para todas las clases?

Todas las clases tienen un constructor por defecto

- Es el constructor ***NombreClase*()** – sin parámetros. Este constructor se define por defecto cuando el programador no ha especificado constructores para una clase.
- Si el programador define un (o varios) constructores para una clase, el constructor sin parámetros deja de estar disponible automáticamente (aunque puede ser programado explícitamente).

Todas las clases tienen un constructor por defecto

- El constructor por defecto sólo crea el objeto y no hace nada más.
- No tiene parámetros y es el que hemos estado usando hasta ahora para definir nuevos objetos.

```
New NombreClase()
```

Otra definición de constructor

- Todo esto nos lleva a una nueva definición de constructor más adecuada:

Constructor es aquel método que crea un objeto de la clase que lo contiene.

- Por eso se llaman constructores. Crean (construyen) el objeto.

Los constructores nos permiten crear objetos a partir de clases

- En el ejemplo, los constructores permiten crear objetos **`Auto`** a partir de la clase **`Auto.`**

Ejercicio

- En la clase **`UsuarioBanco`**, definir un constructor sin parámetros, a parte del que está definido.
- Compilar y crear un objeto de esta clase. ¿Qué opciones tenemos ahora para crear un nuevo objeto de esta clase?

Solución: una opción sin parámetros y otra con ellos

```
Public Class UsuarioBanco
  Private nombre,cedula As String
  Private direccion As String
  Public Sub New()
  End Sub
  Public Sub New (ByVal nNombre As String,
   ByVal nCedula As String)
     Me.nombre = nNombre
     Me.cedula = nCedula
  End Sub
                  Aquí otros métodos
 End Class
```

Los constructores también pueden estar sobrecargados

```
Public Class UsuarioBanco
  Private nombre,cedula As String
  Private direccion As String
  Public Sub New()
  End Sub
  Public Sub New (ByVal nNombre As String,
   ByVal nCedula As String)
     Me.nombre = nNombre
     Me.cedula = nCedula
  End Sub
                  Aquí otros métodos
 End Class
```

Sin embargo, no necesitan la palabra clave `Overloads`

```
Public Class UsuarioBanco
  Private nombre,cedula As String
  Private direccion As String
  Public Sub New()
  End Sub
  Public Sub New (ByVal nNombre As String,
   ByVal nCedula As String)
     Me.nombre = nNombre
     Me.cedula = nCedula
  End Sub
                  Aquí otros métodos
 End Class
```

Los constructores también pueden estar sobrecargados

•Muy común. Por ejemplo:algunos constructores de la clase **`DateTime`** (hay más)

```
Public Sub New(ByVal ticks As Long)

Public Sub New(ByVal year As Integer, ByVal month
  As Integer, ByVal day As Integer)

Public Sub New(ByVal year As Integer, ByVal month
  As Integer, ByVal  day As Integer, ByVal hour
  As Integer, ByVal minute As Integer, ByVal
  second) As Integer

Public Sub New(ByVal year As Integer, ByVal month
  As Integer, ByVal  day As Integer, ByVal hour
  As Integer, ByVal minute As Integer, ByVal
  second As Integer, ByVal millisecond As
  Integer)
```

Los constructores también pueden estar sobrecargados

- Por ejemplo, se puede hacer:

```
fechahora1 = New
  DateTime(2003,1,1)
fechahora2 = New
  DateTime(2003,1,1,12,3,59)
```

Como llamar a un constructor desde dentro de una clase

- A veces, varios constructores comparten un mismo código. Por ello, un constructor puede llamar a otro.
- La llamada debe ser la primera línea del constructor y debe tener la forma:

```
Me.New(listaparametros)
```

Por ejemplo, inicialización de libretas de banco

Si no se especifica nada, se supone que el código de cuenta es 0 y que se abren con 100 dólares.

```
Public Class LibretaBanco
  Private numero,saldo As Integer
  Public Sub New()
     Me.numero = 0
     Me.saldo = 100
  End Sub
  Public Sub New (ByVal nNumero As Integer,
    ByVal nSaldo As Integer)
     Me.numero = nNumero
     Me.saldo = nSaldo
  End Sub
 End Class
```

Pero esto es código redundante

La redundancia de código disminuye la legibilidad y dificulta el mantenimiento.

```
Public Class LibretaBanco
  Private numero,saldo As Integer
  Public Sub New()
     Me.numero = 0
     Me.saldo = 100
  End Sub
  Public Sub New (ByVal nNumero As Integer,
    ByVal nSaldo As Integer)
     Me.numero = nNumero
     Me.saldo = nSaldo
  End Sub
 End Class
```

Una solución mejor

Si un constructor llama a otro, se acaba la red. de código y la clase es más fácil de mantener.

```
Public Class LibretaBanco
  Private numero,saldo As Integer
  Public Sub New()
     Me.New(0,100)
  End Sub
  Public Sub New (ByVal nNumero As Integer,
    ByVal nSaldo As Integer)
     Me.numero = nNumero
     Me.saldo = nSaldo
  End Sub
 End Class
```

Llama al otro constructor

Llamar a un constructor desde otro es una buena práctica

- Reduce la repetición de código.
- Mejora la legibilidad.
- Facilita el mantenimiento.

Llamar a un constructor desde otro es una buena práctica

Si queremos cambiar el nombre de los atributos, sólo deberíamos modificar el segundo constructor

```
Public Class LibretaBanco
  Private numeroLibreta,saldo As Integer
  Public Sub New()
     Me.New(0,100)
  End Sub
  Public Sub New (ByVal nNumero As Integer,
    ByVal nSaldo As Integer)
     Me.numeroLibreta = nNumero
     Me.saldo = nSaldo
  End Sub
 End Class
```

Llama al otro constructor

Ejercicios

- Crear una clase que implemente un cheque del Departamento Fiscal de una gran empresa. Se necesitan dos informaciones de cada cheque: el destinatario y el monto.
- Queremos realizar dos acciones: crear un cheque y recuperar el monto del cheque. Cuando creemos el cheque podemos introducir cualquiera de las dos informaciones, las dos o bien no introducir ninguna. Si no se introduce el destinatario, se supone que es el Ministerio de Hacienda. Si no se introduce el monto, se supone que son cien dólares.
- Utilizar al máximo posible la opción de llamar a un constructor desde otro. Nota: el estado de la clase no debe estar encapsulado.

5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (`Shared`)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (`Shared`)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

El problema de reinventar la rueda

- Muchas veces intentamos programar algo que ya está programado.
- Pérdida de tiempo y esfuerzo.
- Pérdida de calidad:
  - Lo que ya está programado ya ha sido depurado.
  - Lo que programamos es más probable que tenga errores.

Evitando reinventar la rueda

- Para evitar estos problemas, podemos utilizar clases ya programadas por otros programadores:
  - o bien **clases estándar** ya programadas que vienen con la plataforma .NET.
  - o bien **clases de otros programadores** de la misma empresa o incluso de nosotros mismo.
  - o bien **clases de terceras empresas** que compramos o descargamos.
- Estas clases se agrupan en **bibliotecas** para su mejor organización.

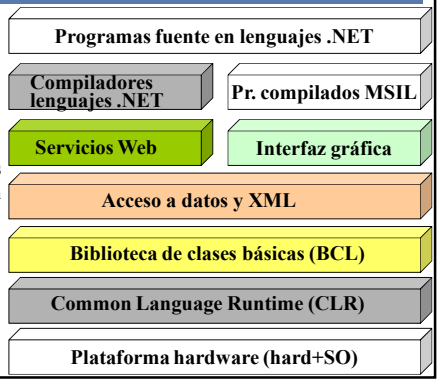
Las bibliotecas son conjuntos de clases listas para ser reutilizadas

Utilizar las clases de las bibliotecas:

- Ahorra tiempo y dinero.
- Aumenta la calidad de nuestro código.
  - Código con menos errores.
  - Código más legible.

Las bibliotecas en la plataforma .NET

Es todo lo que está en color. Son accedidas por los programas compilados. Contienen conjuntos de clases.



Pero estas no son las únicas bibliotecas disponibles

- Recordemos que también hay bibliotecas:
  - Desarrolladas por otros programadores de nuestra empresa o por nosotros mismos.
  - O bien desarrolladas por terceras personas y empresas.

Como acceder a las clases predefinidas de las bibliotecas

- Las bibliotecas se organizan mediante estructuras llamadas espacios de nombres (**namespaces**).
- El espacio de nombres es un conjunto de clases y otros espacios de nombres (subespacios) relacionados entre sí.
- De hecho, además de clases pueden tener otros elementos (interfaces, estructuras, etc.) pero esto no lo veremos por ahora.
- Cada espacio de nombres tiene un nombre. Así:

```
System
System.Windows.Forms
Com.Aurumsol.Idioma
```

Interpretando el nombre de un espacio de nombres

- Los puntos indican subespacios. Así:

```
System.Windows
```

Espacio `System`, subespacio `Windows`

```
Com.Aurumsol.Idioma
```

Esp. `Com`, subesp. `Aurumsol`, subesp. `Idioma`

- Como se ve, los espacios de nombres se organizan en una jerarquía parecida a la de subdirectorios. Son como "subdirectorios" de clases.

Mail: palasi@palasi.com Web: www.palasi.com

Los espacios de nombres son conjuntos de clases listas para ser utilizadas

- Por ejemplo, el espacio `Com.AurumSol.Idioma` tiene un conjunto de clases que podemos utilizar para traducir números a y desde diferentes idiomas.
- Entre estas clases están:

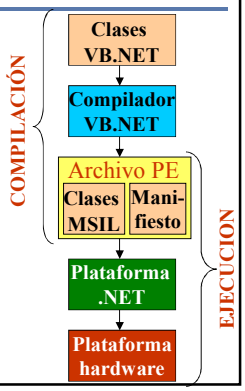
```
TraduccionCastellano
TraduccionCatalan
TraduccionFrances
TraduccionIngles
```

¿Cómo se guardan los espacios de nombres en disco?

- Las clases ocupan espacio en disco y los espacios de nombres, que son conjuntos de clases, también lo hacen.
- Por lo tanto, los espacios de nombre deberán guardarse en algún archivo.
- Concretamente, los espacios de nombre se guardan en archivos de biblioteca .DLL que pueden contener uno o varios espacios.
- Excepcionalmente, también puede pasar que un espacio de nombres esté definido en más de una .DLL.
- Las DLL son bibliotecas compiladas MSIL que contienen espacios de nombres con sus clases.

Importante: Las DLLs de las bibliotecas

- Son archivos PE, como los EXE que hemos generados.
- Es decir, no son DLLs tradicionales (escritas en binario) sino que están escritas en MSIL.
- Por lo tanto, son interpretadas por la plataforma .NET y la necesitan para ejecutarse



5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

Usando clases predefinidas

- Para usar una clase predefinida en nuestro programa, hay que seguir estos dos pasos:
- 1. Saber qué clase resuelve mi problema.
- 2. Saber qué espacio de nombres contiene esa clase.
- 3. Saber qué DLL contiene esa clase.
- 4. Agregar una referencia a la DLL en nuestro proyecto.
- 5. Opcionalmente, utilizar la palabra clave `Imports` si se quieren abreviar las expresiones.

Queremos traducir un número a castellano

- Vamos a aplicar el método que acabamos de ver:
- 1. Saber qué clase resuelve mi problema.
- 2. Saber qué espacio de nombres contiene esa clase.
- 3. Saber qué DLL contiene esa clase.
- 4. Agregar una referencia a la DLL en nuestro proyecto.
- 5. Opcionalmente, utilizar la palabra clave `Imports` si se quieren abreviar las expresiones.

Mail: palasi@palasi.com Web: www.palasi.com

1. Saber qué clase resuelve mi problema

- Mi problema es traducir un número a castellano.
- Hemos visto que hay una clase que se llama **`TraduccionCastellano`** que tiene un método **`TraduceNumero`**, que resuelve este problema.

2. Saber qué espacio de nombres contiene esa clase

- Acabamos de ver que el espacio de nombres es **`Com.AurumSol.Idioma`**
- Hemos visto que éste contiene las clases:

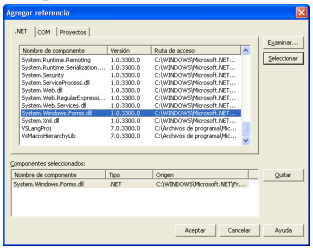
```
TraduccionCastellano
TraduccionCatalan
TraduccionFrances
TraduccionIngles
```

3. Saber qué DLL contiene el espacio de nombres

- En nuestro ejemplo, para acceder al espacio de nombres **`Com.AurumSol.Idioma`** hay que incluir la DLL **`Com.AurumSol.Idioma.dll`**
- Esta DLL se les proporciona con el curso, cópienla en un directorio de su disco duro.
- El nombre de la DLL no tiene porqué coincidir con el nombre del espacio de nombres, pero muchas veces es así por claridad.

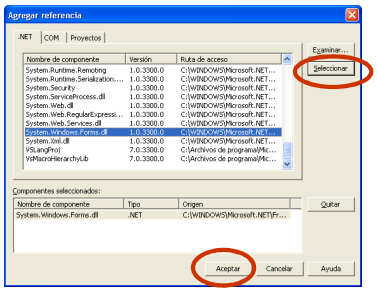
4. Agregar una referencia a la DLL en nuestro proyecto

- Se utiliza la opción **Proyecto|Agregar referencia**.
- Aparecen una lista de DLLs bibliotecas que contienen espacios de nombres.



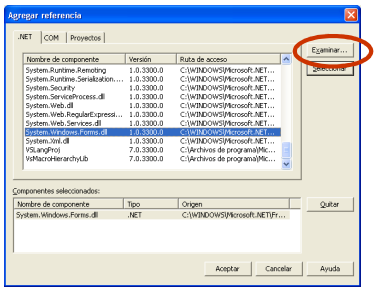
¿Cómo se hace eso?

- Se selecciona la DLL que se necesita, se hace clic en **Seleccionar** y después en **Aceptar.**



Pero no encontramos la DLL `Com.AurumSol.Idioma`

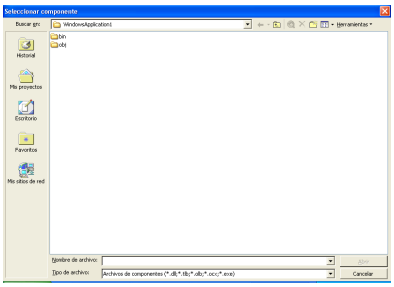
- Esto es porque es una DLL propia. Para encontrala hacemos clic en **Examinar**



Mail: palasi@palasi.com Web: www.palasi.com

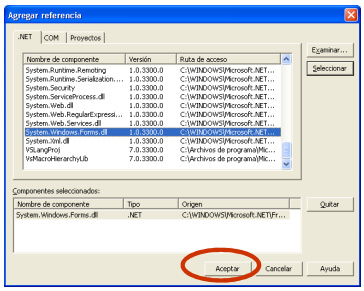
Aparecerá un cuadro de diálogo del tipo de "Abrir archivo"

- En él seleccionaremos la DLL **`Com.AurumSol.Idioma`** y haremos clic en **`Abrir`**.



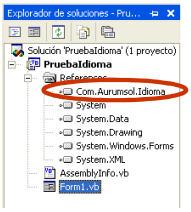
Una vez seleccionada la DLL

- Hacemos clic en **`Aceptar`** y ya hemos agregado la referencia de la DLL a nuestro proyecto.



Para ver que ha sido agregada correctamente

- En el **`Explorador de soluciones`**, expandimos el nodo **`References`**. Aquí están todas las referencias de nuestro programa y también debe estar **`Com.AurumSol.Idioma`**.



5. Opcionalmente, utilizar la palabra `Imports`

- Este paso es opcional y no lo veremos por ahora. Así que ya hemos acabado.
- Ya podemos usar en nuestro proyecto la clase **`TraduccionCastellano`**
- Sólo tenemos que tener en cuenta que la clase debe ir prefijada por el nombre del espacio para utilizarla. Así:

```
Com.AurumSol.Idioma.TraduccionCastellano
```

La clase `TraduccionCastellano`

- Tiene un método

```
Public Function
TraduceNumero(ByVal numero As
Integer) As String
```

- Que traduce un número del 0 al 10 a una cadena en lenguaje castellano.

Ejercicio

- Usando la clase anterior, crear un nuevo proyecto con un formulario que contenga un cuadro de texto y un botón.
- En el cuadro de texto se introducirá un número del 0 al 10. Cuando se haga clic en el botón, la traducción al castellano de ese número aparecerá en un **`MessageBox`**.

Una posible solución

```
Public Class Form1
    Inherits System.Windows.Forms.Form
Código generado por el Diseñador de Windows Forms
Private Sub Button1_Click...
    Dim objTrad As _
       Com.AurumSol.Idioma.TraduccionCastellano
    Dim traduccion As String
    objTrad = New _
       Com.AurumSol.Idioma.TraduccionCastellano()
    traduccion = _
      objTrad.TraduceNumero(Me.TextBox1.Text)
    MessageBox.Show(traduccion)
End Sub
```

Vemos que es incómodo

```
Public Class Form1
    Inherits System.Windows.Forms.Form
Código generado por el Diseñador de Windows Forms
Private Sub Button1_Click...
    Dim objTrad As _
       Com.AurumSol.Idioma.TraduccionCastellano
    Dim traduccion As String
    objTrad = New _
       Com.AurumSol.Idioma.TraduccionCastellano()
    traduccion = _
      objTrad.TraduceNumero(Me.TextBox1.Text)
    MessageBox.Show(traduccion)
End Sub
```

Escribir todo el nombre del espacio delante de la clase. Es poco legible e incómodo de teclear.

Para evitarlo, tenemos la palabra clave `Imports`

- La palabra clave **`Imports`** nos permite usar las clases de un espacio de nombres sin tener que prefijarlas con el nombre del espacio.
- Debe ser la primera instrucción del texto:

`Imports` ***espacioDeNombres***

- Después de **`Imports`**, todas las clases del espacio se pueden usar sin prefijarlos.

Usando `Imports`

```
Imports Com.AurumSol.Idioma
Public Class Form1
    Inherits System.Windows.Forms.Form
Código generado por el Diseñador de Windows Forms
Private Sub Button1_Click...
    Dim objTrad As TraduccionCastellano
    Dim traduccion As String
    objTrad = New TraduccionCastellano()
    traduccion = _
      objTrad.TraduceNumero(Me.TextBox1.Text)
    MessageBox.Show(traduccion)
End Sub
```

Debido a la instrucción **`Imports`**, ahora podemos usar **`TraduccionCastellano`** sin tener que prefijarla.

Como vemos, este era el último paso de nuestro método

- Vamos a aplicar el método que acabamos de ver:
- 1. Saber qué clase resuelve mi problema.
- 2. Saber qué espacio de nombres contiene esa clase.
- 3. Saber qué DLL contiene esa clase.
- 4. Agregar una referencia a la DLL en nuestro proyecto.
- 5. Opcionalmente, utilizar la palabra clave **`Imports`** si se quieren abreviar las expresiones.

Ejercicio

- Reformar el programa anterior para que, cada vez que se haga clic en el botón, se traduzca simultáneamente el número al castellano, catalán, inglés y francés.
- Utilicen **`Imports`** para no tener que prefijar los nombres de las clases.

5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

Usando bibliotecas de la plataforma .NET a nuestro proyecto

- Hasta ahora nos hemos dedicado a usar clases de terceras personas (como **`TraduccionCastellano`**).
- Pero sabemos que la plataforma .NET tiene muchas bibliotecas con clases predefinidas, listas para ser usadas.
- ¿Cómo las usamos?

Se usan como las clases de terceros

- Por ello, en la explicación hasta ahora, no hemos hecho distinciones.
- Sin embargo, hay una pequeña diferencia.
- Hay bibliotecas que ya están importadas por defecto en los proyectos de Visual Basic.
- Veamos un ejemplo.

El espacio de nombres `System.Windows.Forms`

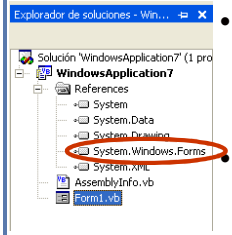
- Contiene clases que sirven para diseñar formularios.
- Entre estas clases están:

```
MessageBox
TextBox
Button
```

Teóricamente, tendríamos que escribir

-
```
System.Windows.Forms.MessageBox
.Show("Hola a todos")
```
- Sin embargo, sólo escribimos
```
MessageBox.Show("Hola a todos")
```
- Esto es porque el espacio **`System.Windows.Forms`** está importado por defecto en todos los proyectos de VB .NET definidos como “Aplicación para Windows”.

Creen una nueva solución y miren en el Explorador de soluciones



- En **References**, verán todos los espacios de nombres importados por defecto.
- No sólo sus referencias están agregadas al proyecto, sino que hay un **`Import`** implícito.
- Por ello podemos usar sus clases sin prefijarlas.

Mail: palasi@palasi.com Web: www.palasi.com

5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

Del método hay dos puntos que son "mecánicos"

- Vamos a aplicar el método que acabamos de ver:
- 1. Saber qué clase resuelve mi problema.
- 2. Saber qué espacio de nombres contiene esa clase.
- 3. Saber qué DLL contiene esa clase.
- 4. Agregar una referencia a la DLL en nuestro proyecto.
- 5. Opcionalmente, utilizar la palabra clave **`Imports`** si se quieren abreviar las expresiones.

Sin embargo, necesitamos información para los otros tres

- Vamos a aplicar el método que acabamos de ver:
- 1. Saber qué clase resuelve mi problema.
- 2. Saber qué espacio de nombres contiene esa clase.
- 3. Saber qué DLL contiene esa clase.
- 4. Agregar una referencia a la DLL en nuestro proyecto.
- 5. Opcionalmente, utilizar la palabra clave **`Imports`** si se quieren abreviar las expresiones.

Para estos, necesitamos documentación sobre las bibliotecas.

¿Dónde encontramos documentación sobre las bibliotecas?

- Si son bibliotecas de terceras partes, el vendedor debe proporcionarnos documentación suficiente sobre los espacios de nombres, las clases que contienen y las DLLs en las que están guardados.
- Si son bibliotecas de nuestra empresa, también debemos tener esa documentación.
- ¿Qué pasa si son bibliotecas de la plataforma .NET?

Encontrando documentación sobre las bibliotecas de la plataforma .NET

- Microsoft tiene una documentación completa en el URL.

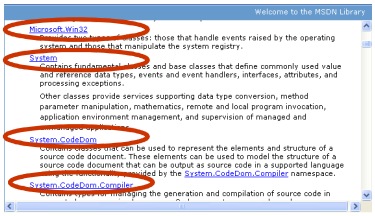
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/cpref_start.asp`

- Esto corresponde a

MSDN Home > MSDN Library > .NET Development > .NET Framework SDK > .NET Framework > Reference

en el árbol de la izquierda de MSDN.

En esta página

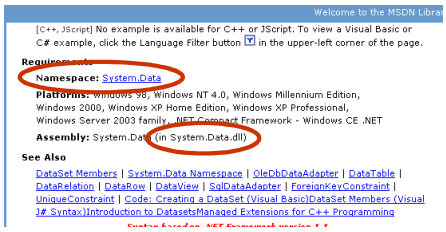


- Hay hiperenlaces a cada uno de los espacios de nombres. Siguiéndolos se llega a las diferentes clases.
- Cada espacio de nombres incluye clases de un cierto tema y allí buscaremos las clases que resuelven ese tema.

Mail: palasi@palasi.com Web: www.palasi.com

Al final de la página de cada clase

- Aparece el espacio de nombres al que pertenece y la DLL en que se encuentra.



Otros recursos

- Hay un programa llamado WinCV que se encuentra en **<*Directorio de Visual Studio*> `\FrameworkSDK\Bin\WinCV.exe`**
- En el programa se puede escribir una palabra y aparecen todas las clases que tienen esa palabra.
- Esto nos puede servir para buscar la clase indicada.

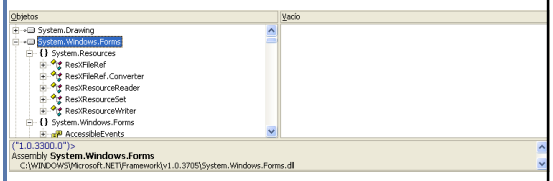
Por ejemplo, poniendo la palabra "File"

- Aparecen todas las clases de la plataforma que tienen esa palabra.
- También aparecen sus atributos, propiedades y métodos (pero en sintaxis C#)

Otros recursos

- Para ver todas las clases y espacios de nombres, agregados en nuestro programa podemos mostrar el Examinador de objetos oprimiendo F2.

El examinador de objetos



- Aparecen las DLLs ("assemblys"), espacios de nombres y clases. Cada clase si es seleccionada, revela sus miembros.

Un libro útil

- "VB.NET Core Classes in a Nutshell", Budi Kurniawan, Ted Neward, Editorial O'Reilly.

Mail: palasi@palasi.com Web: www.palasi.com

palasi.com
IT and management

5. Espacios de nombres

- 5.1. Conceptos de bibliotecas y espacios de nombres.
- 5.2. Usar clases predefinidas.
- 5.3. Usar bibliotecas de la plataforma .NET
- 5.4. Obtener documentación sobre bibliotecas.
- 5.5. Crear nuestras propias bibliotecas.

Crear nuestras propias bibliotecas de clases

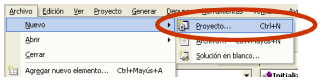
- Hasta ahora, hemos hablado de usar bibliotecas de clases preexistentes.
- Sin embargo, no hemos creado bibliotecas de clases propias.
- Esto es fácil de crear siguiendo el método que explicamos a continuación.

Método para crear nuestras propias clases

- 1. Crear un proyecto "Biblioteca de clases".
- 2. Programar las clases que deseemos
  - Crear una clase nueva
  - Escribir su código.
  - Asignarle un espacio de nombre.
- 3. Compilar la biblioteca.
- 4. Distribuir la biblioteca.

1. Crear un proyecto "Biblioteca de clases"

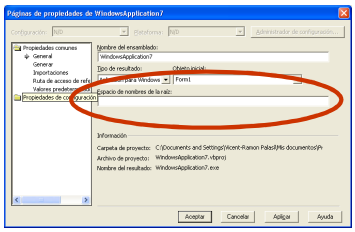
- Se selecciona la opción **Archivo|Nuevo|Proyecto**



- En la ventana que aparece se elige "Biblioteca de clases", se escribe el nombre de la DLL y se hace clic en "Aceptar".

1. Crear un proyecto "Biblioteca de clases"

- En el **Explorador de soluciones**, se hace clic derecho en el proyecto y se elige **Propiedades**
- El cuadro de texto llamado **Espacio de nombres de la raíz** se deja en blanco.



2. Programar las clases que deseamos

- 2.1. Creamos la clase con **Agregar nuevo elemento | Clase** (ya lo vimos)
- 2.2 Escribimos su código.

```
Public Class ClasePrueba
    Private atributo1 As Integer
    Private atributo2 As String
    Public Sub Metodo1()

    End Sub
    Public Function Metodo2() As String

    End Function
End Class
```

- 2.3. Comenzamos el código con

```
Namespace nombreEspacioNombres
```

Lo acabamos con

```
End Namespace
```


Mail: palasi@palasi.com Web: www.palasi.com

2. Programar las clases que deseamos

```
Namespace Com.Aurumsol.Prueba
    Public Class ClasePrueba
        Private atributo1 As Integer
        Private atributo2 As String
        Public Sub Metodo1()
        End Sub
        Public Function Metodo2() As String
        End Function
    End Class
End Namespace
```

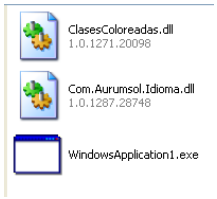
- Con la palabra clave **`Namespace`** indicamos que espacio de nombres queremos incluir esa clase.

3. Compilar la biblioteca

- Basta con hacer **Generar|Generar solución**.
- En el subdirectorio **`bin`**, aparecerá una DLL con la biblioteca compilada.

4. Distribuir la biblioteca

- Cada vez que usamos nuestra biblioteca desde otro proyecto, en el directorio **`bin`** se genera una copia de esta biblioteca.
- Hay que copiar todas las DLLs y EXEs que hay en el directorio **`bin`**. Es decir, se debe copiar las bibliotecas junto con el programa. Si no, no funciona.



5. Más sobre la programación orientada a objetos con VB.NET

- 5.1. Sobrecarga.
- 5.2. Constructores.
- 5.3. Ámbito de clase (**`Shared`**)
- 5.4. Namespaces.
- 5.5. Herencia.
- 5.6. Modificadores de acceso.

La clase `Auto`

- Modela la descripción de un auto. Permite crear objetos **`Auto`**.
- De un auto me interesa la clase de su motor y la marca de sus ruedas.

La clase `Auto` (1)

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Public Property Motor() As String
     Get
       Return pMotor
     End Get
     Set(ByVal Value As String)
       pMotor=Value
     End Set
  End Property
```


Mail: palasi@palasi.com Web: www.palasi.com

La clase Auto (2)

```
Public Property MarcaRuedas() As String
     Get
       Return pMarcaRuedas
     End Get
     Set(ByVal Value As String)
       pMarcaRuedas=Value
     End Set
  End Property
End Class
```

La clase Auto abreviada

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Public Property Motor() As String
    …
  End Property
  Public Property MarcaRuedas() As String
    …
  End Property
End Class
```

Ahora queremos modelar el concepto de un auto importado

- De un auto importado me interesa además de su motor y marca de ruedas, el año de importación y el código de la aduana.
- ¿Cómo podemos modelar esto?

Primera solución: Crear una nueva clase

```
Public Class AutoImportado
  Private pMotor,pMarcaRuedas As String
  Private pAno,pCodigoAduana As Integer
  Public Property Motor() As String
    …
  Public Property MarcaRuedas() As String
    …
  Public Property Ano() As Integer
    …
  Public Property CodigoAduana() As Integer
    …
End Class
```

Esta clase tiene muchos elementos comunes con Auto

```
Public Class AutoImportado
  Private pMotor,pMarcaRuedas As String
  Private pAno,pCodigoAduana As Integer
  Public Property Motor() As String
    …
  Public Property MarcaRuedas() As String
    …
  Public Property Ano() As Integer
    …
  Public Property CodigoAduana() As Integer
    …
End Class
```

Las clases Auto y AutoImportado tienen mucho código en común

- La repetición de código dificulta el mantenimiento.
- Cuando se quiere cambiar algo, siempre hay que recordar que debe hacerse en dos sitios diferentes.
- A la larga, hay un descuido (con clases más complicadas) y el código se vuelve inconsistente.

Mail: palasi@palasi.com Web: www.palasi.com

Ejemplo de problema que causa la repetición de código

- Ahora queremos que el motor y la marca de las ruedas de cada auto no puedan fijarse en cualquier circunstancia sino sólo cuando se crea el auto.
- Para ello haremos que las propiedades **`Motor`** y **`MarcaRuedas`** sean de sólo lectura y crearemos un constructor que reciba estas dos informaciones como parámetros.

Esto debería realizarse por duplicado

- Se debería cambiar el código para las clases **`Auto`** y **`AutoImportado`**.
- Podemos equivocarnos y dejar una versión inconsistente con la otra (por ejemplo, número de constructores).
- Imagínense qué difícil se vuelve si tenemos cinco clases más: **`AutoDeLujo`**, **`AutoPúblico`**, **`AutoTaxi`**, **`AutoExtranjero`** y **`AutoRobado`**.

Conclusión

- Esta no es una buena solución: hace del mantenimiento un infierno.

Segunda solución: Mezclar las dos clases

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Private pAno,pCodigoAduana As Integer
  Private pEsImportado As Boolean
  Public Property Motor() As String ...
  Public Property MarcaRuedas() As String
  Public Property Ano() As Integer ...
  Public Property CodigoAduana() As Integer
  Public Property EsImportado() As Boolean
End Class
```

Segunda solución: Mezclar las dos clases

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Private pAno,pCodigoAduana As Integer
  Private pEsImportado As Boolean
  Public Property Motor() As String ...
  Public Property MarcaRuedas() As String
  Public Property Ano() As Integer ...
  Public Property CodigoAduana() As Integer
  Public Property EsImportado() As Boolean
End Class
```

De **EsImportado** depende que utilice las propiedades enteras o no

Segunda solución: Mezclar las dos clases

- Esta segunda solución es inadecuada por dos razones.
- Primera: las propiedades **`Ano`** y **`CodigoAduana`** están disponibles para todos los autos y es tarea del programador cuidarse de que un auto no importado no los use.

Segunda solución: Mezclar las dos clases

- Esta segunda solución es inadecuada por dos razones.
- Segunda: cuando implementamos un nuevo método siempre tenemos que ir distinguiendo los diferentes casos y resulta "código espagueti", excesivamente enredado.

Esta segunda solución produce código enredado

- Imaginemos un método **`ImprimeTodosLosDatos`**.

```
Public Sub ImprimeTodosLosDatos()
  MessageBox.Show(Me.Motor)
  MessageBox.Show(Me.MarcaRuedas)
  If Me.EsImportado Then
   MessageBox.Show("Auto importado");
   MessageBox.Show(Me.Ano);
   MessageBox.Show(Me.CodigoAduana);
  Else
   MessageBox.Show("Auto no importado")
  End If
End Sub
```

Esta segunda solución produce código enredado

- Código enredado para dos opciones.

```
Public Sub ImprimeTodosLosDatos()
  MessageBox.Show(Me.Motor)
  MessageBox.Show(Me.MarcaRuedas)
  If Me.EsImportado Then
   MessageBox.Show("Auto importado");
   MessageBox.Show(Me.Ano);
   MessageBox.Show(Me.CodigoAduana);
  Else
   MessageBox.Show("Auto no importado")
  End If
End Sub
```

Esta segunda solución produce código enredado

- El código sería difícil de mantener si se tiene docenas de tipos de autos y muchos atributos en cada auto.

```
Public Sub ImprimeTodosLosDatos()
  MessageBox.Show(Me.Motor)
  MessageBox.Show(Me.MarcaRuedas)
  If Me.EsImportado Then
   MessageBox.Show("Auto importado");
   MessageBox.Show(Me.Ano);
   MessageBox.Show(Me.CodigoAduana);
  Else
   MessageBox.Show("Auto no importado")
  End If
End Sub
```

Esta segunda solución produce código enredado

- Además cada nuevo tipo significa reprogramar el método (añadiendo **`If`**s) por lo cual podemos introducir errores de programación.

```
Public Sub ImprimeTodosLosDatos()
  MessageBox.Show(Me.Motor)
  MessageBox.Show(Me.MarcaRuedas)
  If Me.EsImportado Then
   MessageBox.Show("Auto importado");
   MessageBox.Show(Me.Ano);
   MessageBox.Show(Me.CodigoAduana);
  Else
   MessageBox.Show("Auto no importado")
  End If
End Sub
```

Debe haber una mejor solución

- En los lenguajes no orientados a objetos debíamos conformarnos con la última solución y resignarnos a un mantenimiento difícil.
- Pero en los lenguajes orientados a objetos podemos adoptar un enfoque distinto gracias al poderoso mecanismo llamado **herencia.**

Herencia

- Con la ayuda de la palabra reservada **`Inherits`** podemos definir **`AutoImportado`** como una clase específica de **`Auto`**.
- Comparte todos los miembros de **`Auto`** pero, además tiene miembros especiales: los dos miembros del año de importación y el código de la aduana.

Supongamos que tenemos la clase `Auto` como antes

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Public Property Motor() As String
    ...
  End Property
  Public Property MarcaRuedas() As String
    ...
  End Property
End Class
```

Con la herencia escribimos la siguiente clase

- Mucho más corta. Sólo incluimos los atributos y métodos específicos de **`AutoImportado`**.
- Los otros atributos y métodos están disponibles pues decimos que heredamos (**`Inherits`**) la clase **`Auto`**. Por lo tanto, todo lo de **`Auto`** está disponible aquí

```
Public Class AutoImportado
  Inherits Auto
  Private pAno,pCodigoAduana As Integer
  Public Property Ano() As Integer …
  Public Property CodigoAduana() As Integer
End Class
```

Con la herencia escribimos la siguiente clase

- Es como si hubiéramos copiado los atributos y métodos de la clase **`Auto`** dentro de la clase **`AutoImportado`**
- Pero sin las molestias y problemas de hacerlo realmente (las cuales hemos visto).

```
Public Class AutoImportado
  Inherits Auto
  Private pAno,pCodigoAduana As Integer
  Public Property Ano() As Integer …
  Public Property CodigoAduana() As Integer
End Class
```

Ahora cambios en la implementación son fáciles

- Si quiero añadir un constructor para que la marca y el modelo no cambien, lo hago en la clase **`Auto`**, ya que no hay repetición de código.

```
Public Class AutoImportado
  Inherits Auto
  Private pAno,pCodigoAduana As Integer
  Public Property Ano() As Integer …
  Public Property CodigoAduana() As Integer
End Class
```

Ahora cambios en la implementación son fáciles

- Si quiero añadir un nuevo tipo de auto no tengo que tocar las dos clases para reprogramarlas y me evito introducir errores de programación en clases ya depuradas.

```
Public Class AutoRobado
  Inherits Auto
  ...

End Class
```


Mail: palasi@palasi.com Web: www.palasi.com

Además el código es más legible

- Pues es más sencillo. No incluye **If**s innecesarios ni repetición de código.

```
Public Class AutoImportado
  Inherits Auto
  Private pAno,pCodigoAduana As Integer
  Public Property Ano() As Integer …
  Public Property CodigoAduana() As Integer
End Class
```

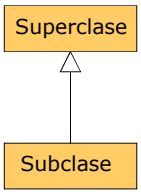
Ventajas de la herencia

- Código más legible y claro.
- Código más fácil de mantener.
- Código más flexible.

Un poco de terminología

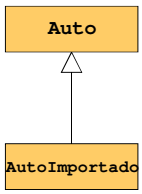
- La clase que tiene la palabra **Inherits** se le llama subclase.
- La clase a la cual hace referencia con **Inherits** se le llama superclase.
- En nuestro caso, la superclase es **Auto** y la subclase se llama **AutoImportado**

Un poco de notación



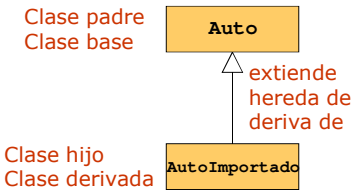
- Se representa la herencia conectando la subclase con la superclase mediante una línea con un triángulo que toca la superclase.

En nuestro caso



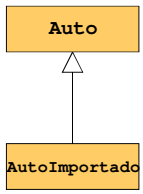
- **Auto** es la superclase y **AutoImportado** es la subclase.

Más terminología



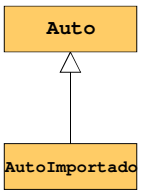
- Se dice que **AutoImportado** hereda de **Auto** o bien que **AutoImportado** deriva de **Auto**.

Es fácil ver que *AutoImportado* es una clase especial de *Auto*



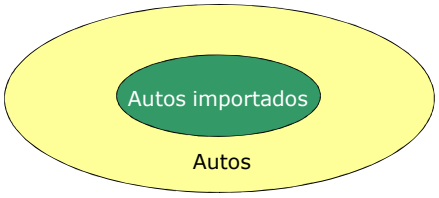
- Esto es general: siempre la subclase debe ser un tipo especial de la superclase.

Dicho de otra manera: *AutoImportado* es un *Auto*



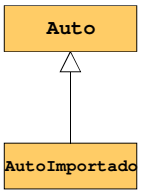
- Por ello, a la relación de herencia se le llama relación "ES UN" ("is a").

El conjunto de los autos importados es un subconjunto del de los autos



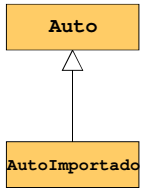
- Se dice que la clase **AutoImportado** es una subclase de **Auto**.

Especialización



- Es el proceso por el cual derivamos una subclase de una superclase.
- Reconocemos el caso especial y lo programamos.
- Es lo que hemos hecho.

Generalización



- Es el proceso contrario: se obtiene una superclase de la subclase generalizando las condiciones.

La clase Auto

```
Public Class Auto
  Private pMotor,pMarcaRuedas As String
  Public Property Motor() As String
    ...
  End Property
  Public Property MarcaRuedas() As String
    ...
  End Property
End Class
```

- Examinando la clase **Auto** me doy cuenta de que hay aspectos que podrían aplicarse a cualquier vehículo, como la marca de las ruedas.

Mail: palasi@palasi.com Web: www.palasi.com

Implemento una clase Vehiculo para estos aspectos en común

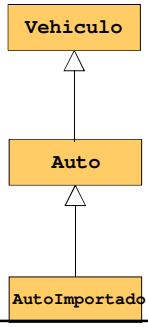
```
Public Class Vehiculo
  Private pMarcaRuedas As String
  Public Property MarcaRuedas() As String
    ...
End Class
```

Ahora Auto es una subclase de Vehiculo.
Hemos generalizado.

```
Public Class Auto
  Inherits Vehiculo
  Private pMotor As String
  Public Property Motor() As String
    ...
End Class
```

Generalización

- Hemos generalizado **Auto** para obtener **Vehiculo.**



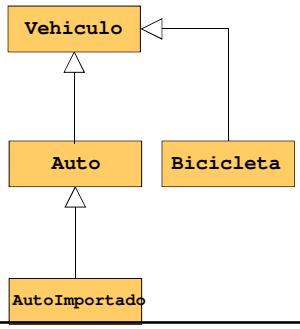
Ahora podría especializar Vehiculo para obtener Bicicleta

```
Public Class Bicicleta
  Inherits Vehiculo
  Private pManillar As String
  Public Property Manillar() As String
  ...
End Class
```

- Como se ve, siempre podemos estar especializando y generalizando con la herencia.

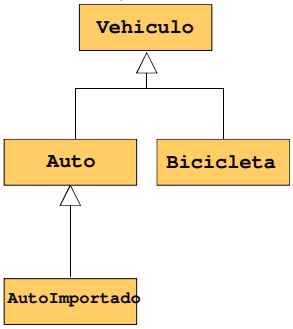
Se obtiene una jerarquía

- Se obtiene un árbol de subclases



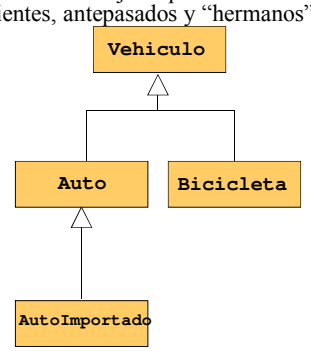
Podemos representarlo así

- No es 1 única línea, son 2 líneas



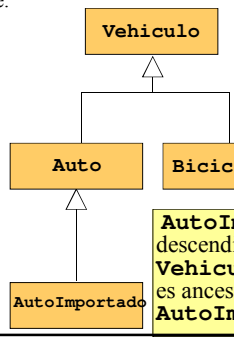
El árbol de clases se le llama una jerarquía

- Cada clase es un nodo de la jerarquía. Podemos definir descendientes, antepasados y "hermanos".



El árbol de clases se le llama una jerarquía

Auto y **Bicicleta** son "hermanos", pues son subclases del mismo padre.



AutoImportado es descendiente de **Vehiculo. Vehiculo** es ancestro o antepasado de **AutoImportado**

Ejercicio

- Crear una jerarquía de clases que contenga las clases Persona, Empleado y Gerente. Se supone que nos interesan los nombres de todos ellos y sus fechas de nacimiento (a partir de la cual calcularemos su edad por un método o propiedad). De los empleados nos interesa su sueldo y su comisión (que es 5%) del sueldo. De un gerente, nos interesa cuál es su jefe y el Departamento al que pertenece. De cada Departamento, nos interesa su nombre y su fecha de fundación.

Motivos de derivar subclases

- Una clase se deriva (especializa) por dos motivos diferentes:
- 1. Extendemos la clase añadiendo miembros (atributos y métodos) como se acaba de ver.
- 2.Especializamos la forma en que se ejecutan los métodos. Esto es un mecanismo conocido como **sobreescritura ("overriding") de métodos.**

Sobreescritura de métodos

- Un método se sobreescribe cuando una subclase da una implementación (cuerpo del método) diferente de la que da la superclase.
- Lo veremos con las clases **Vehiculo** y **Auto**. Suponemos que definimos un método **imprimeAtributos** en **Vehiculo.**

Introduciendo el método ImprimeAtributos

```
Public Class Vehiculo
  Private pMarcaRuedas As String
  Public Property MarcaRuedas() As String
  Public Sub ImprimeAtributos()
     MessageBox.Show (Me.MarcaRuedas)
  End Sub
End Class
```

Introduciendo el método ImprimeAtributos

```
Public Class Vehiculo
  Private pMarcaRuedas As String
  Public Property MarcaRuedas() As String
  Public Sub ImprimeAtributos()
     MessageBox.Show (Me.MarcaRuedas)
  End Sub
End Class
```

La clase **Auto** heredará este método **imprimeAtributos**, de forma que si tenemos un objeto **auto1** de clase **Auto**, cuando pongamos

```
auto1.ImprimeAtributos()
```

imprimirá la marca de las ruedas.

Pero esto no es lógico

- Lo lógico es que el método **imprimeAtributos** del objeto **Auto** imprima todos los atributos de la clase **Auto** y, en especial, el motor.
- Pero como he heredado la implementación de la clase **Vehiculo** esto no lo hago.
- Solución: **sobreescribir** (reemplazar) la implementación heredada del método por una definida en la subclase.

Sobreescribiendo el método imprimeAtributos

```
Public Class Vehiculo
  Private pMarcaRuedas As String
  Public Property MarcaRuedas() As String
  Public Overridable Sub ImprimeAtributos()
     MessageBox.Show (Me.pMarcaRuedas)
  End Sub
End Class
```

Primero, indicamos en la superclase que el método se puede sobreescribir con la palabra clave **Overridable** (delante de **Sub** o **Function** y detrás de **Overloads**, si existe).

Esto es necesario pues, por defecto, los métodos no se pueden sobreescribir.

Sobreescribiendo el método imprimeAtributos

```
Public Class Auto
  Inherits Vehiculo
  Private pMotor As String
  Public Property Motor() As String  ...
  Public Overrides Sub ImprimeAtributos()
     MessageBox.Show(pMarcaRuedas)
     MessageBox.Show(pMotor)
  End Sub
End Class
```

Se rehace la implementación del método, sin cambiar su signatura: ámbito, nombre, tipos de parámetros y resultado.

Sobreescribiendo el método imprimeAtributos

```
Public Class Auto
  Inherits Vehiculo
  Private pMotor As String
  Public Property Motor() As String  ...
  Public Overrides Sub ImprimeAtributos()
     MessageBox.Show(pMarcaRuedas)
     MessageBox.Show(pMotor)
  End Sub
End Class
```

La palabra **Overrides** (delante de **Sub** o **Function** y detrás de **Overloads**, si existe) indica que el método ha sido reescrito.

Ahora el método se adapta a todas las circunstancias

- Si tengo un objeto **Auto auto1** y un objecto **Vehiculo vehi1**.

```
auto1.ImprimeAtributos()
```

imprimirá los atributos de un auto, es decir, su motor y marca de ruedas.

```
vehi1.ImprimeAtributos()
```

imprimirá los atributos de un vehículo, es decir, su marca de ruedas y nada más.

Las propiedades también se pueden sobreescribir

En la propiedad que puede sobreescribirse

```
Public Overridable Property Nombre()...
...
End Property
```

En la propiedad que sobreescribe la anterior

```
Public Overrides Property Nombre() ...
...
End Property
```

Ejercicio

Queremos modelar figuras geométricas (sin dibujarlas en pantalla) solo modelar sus datos.

Concretamente, de cada figura geométrica, nos interesa saber su color y tener métodos para obtener su área y su perímetro.

Las figuras que nos interesan son el cuadrado y el círculo. El área de un cuadrado es l^2 y su perímetro es $4l$ (donde l es la longitud del lado).

El área de un círculo es πr^2 y su perímetro $2\pi r$ (donde r es el radio).

Solución

- La solución se le adjunta en proyectos que se les proporciona.
- Una solución está sin propiedades.
- Otra con propiedades.

Aún podemos mejorar un poco más el método

```
Public Class Auto
  Inherits Vehiculo
  Private pMotor As String
  Public Property Motor() As String  ...
  Public Overrides Sub ImprimeAtributos()
     MessageBox.Show(pMarcaRuedas)
     MessageBox.Show(pMotor)
  End Sub
End Class
```

`MessageBox.Show(pMarcaRuedas)` está repetido en `Vehiculo` y en `Auto`.

Ya sabemos que la repetición de código se debe evitar

- Hace del mantenimiento una pesadilla.
- ¿Cómo podríamos hacerlo aquí?
- La forma más correcta sería llamar al método de la superclase **`Vehiculo`** desde la subclase **`Auto`**.

Esto se hace de la siguiente forma

```
Public Class Auto
  Inherits Vehiculo
  Private pMotor As String
  Public Property Motor() As String  ...
  Public Overrides Sub ImprimeAtributos()
     MyBase.ImprimeAtributos()
     MessageBox.Show(pMotor)
  End Sub
End Class
```

`MyBase.ImprimeAtributos()` ejecuta el método **`ImprimeAtributos`** de la clase padre.

En general

- **Métodos:**

`MyBase.` *`metodo(parámetros)`*

Ejecuta la versión del método ***metodo*** que se definió en la clase padre (superclase)
Es útil para evitar repeticiones de código.

- Si lo que queremos es ejecutar un constructor de la superclase

`MyBase.New(`*`parámetros`*`)`

Sólo en la primera línea del constructor.

Mail: palasi@palasi.com Web: www.palasi.com

Importante

- **`MyBase`** se utiliza cuando la versión del método que tiene la superclase es diferente de la que tiene la subclase.
- Si son la misma versión, se utiliza **`Me`**, no **`MyBase`**.

Prohibiciones cuando sobreescribimos

- No se debe cambiar la semántica (la intención) de un método cuando se sobreescribe.
- No se puede "sobreescribir" un atributo. Para hacer esto se usa un procedimiento llamado "shadowing", pero no es recomendable y, por tanto, no lo veremos.
- No se puede cambiar la signatura de un método sobreescribiéndolo (su ámbito, nombre y tipos de datos o resultado). Se puede hacer con shadowing, pero no es recomendable.

Ventajas de la herencia

- Las clases derivadas son mucho más concisas que lo que serían sin herencia.
- Podemos reutilizar y extender código que ya ha sido depurado sin modificarlo.
- Además, se puede derivar una subclase de una clase incluso si no hay el código fuente de la segunda.
- Además, la clasificación es la manera en que los humanos organizamos la información.

Ejercicio

Queremos modelar figuras geométricas (sin dibujarlas en pantalla) solo modelar sus datos.

Concretamente, de cada figura geométrica, nos interesa saber su color y tener métodos para obtener su área y su perímetro.

Las figuras que nos interesan son el cuadrado, el triángulo y el círculo. El área de un cuadrado es l^2 y su perímetro es $4l$ (donde l es la longitud del lado).

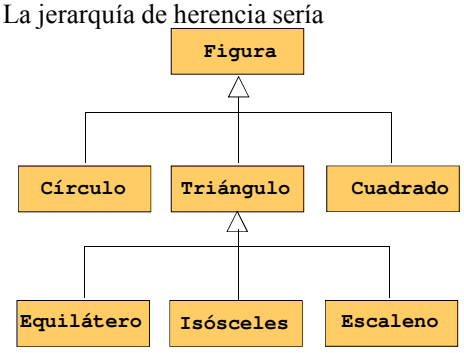
El área de un círculo es πr^2 y su perímetro $2\pi r$ (donde r es el radio).

Ejercicio

El área de un triángulo es *(base x altura)/2* El perímetro del triángulo depende de si el triángulo es equilátero (todos los lados son iguales y miden *l1*), isósceles (tiene dos lados iguales cuya longitud expresaremos por *l1* y otro diferente que se llama *l2*) o escaleno (los tres lados *l1, l2, l3*) son desiguales. En el primer caso el perímetro es *l1 x 3* en el segundo *2 x l1+l2*, en el tercero *l1+l2+l3*. **Utilizar constructores**. Todos los estados de todas las clases deben estar encapsulados. Nota: provisionalmente, se considerará que los métodos de área y perímetro para **`Figura`** devuelven cero. Se deberá sobrescribirlos para todas las clases.

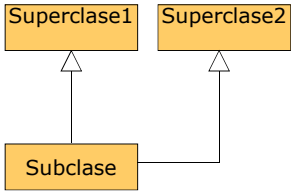
Solución de los ejercicios anteriores

- La jerarquía de herencia sería



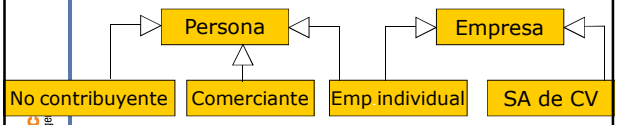
En VB .NET, no existe herencia múltiple

- Toda subclase tiene sólo UNA superclase.
- No hay clases con más de un padre. Lo siguiente está prohibido



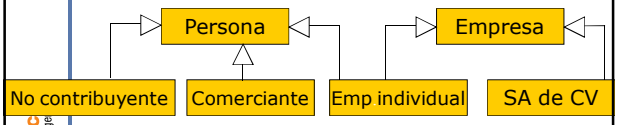
¡¡NO!!

Es decir, se prohíbe la herencia múltiple



- "Emp. Individual" tiene herencia múltiple:
  - hereda todos los datos y métodos de empresa.
  - hereda todos los datos y métodos de persona física.
- Es una persona y es una empresa
- A veces, esto puede producir conflictos.
- La mayoría de lenguajes orientados a objetos no lo permiten. VB.NET tampoco.

Conflictos en herencia múltiple



- Supongamos que:
  - "Persona" define un método llamado "imprimir estado financiero".
  - "Empresa" define un método con el mismo nombre pero que es diferente (adecuado a una empresa)
- ¿Cuál es el método que hereda "Emp. Individual"?
- Es por eso que VB.NET prohíbe la herencia múltiple.

5. Más sobre la programación orientada a objetos con VB.NET

Entendiendo los modificadores de acceso

- Ahora que ya entendemos el concepto de herencia, podemos entender los modificadores de acceso, que antes hemos visto sin explicarlos.

Modificadores de acceso (ámbito)

- `Public`. El acceso al miembro es público. Se puede utilizar el atributo en cualquier circunstancia.
- `Protected`. El miembro es accesible desde la clase que lo define y las clases que derivan de ella.
- `Friend`. El miembro es accesible dentro del mismo proyecto.
- `Protected Friend`. El miembro es accesible dentro del mismo proyecto o bien desde la clase que lo define y las clases que derivan de ella.
- `Private`. El miembro sólo puede ser usado dentro de la clase que lo define.

Mail: palasi@palasi.com Web: www.palasi.com

Los modificadores de acceso se aplican

- A miembros (atributos, métodos y propiedades), como ya hemos visto.
- A clases.
  - Una clase cuya declaración comienza por **`Public Class`** es accesible en toda circunstancia.
  - Una clase **`Friend Class`** o simplemente **`Class`**, es accesible sólo dentro del proyecto.

Modificadores de acceso (miembros)

Resumiendo: tenemos el siguiente cuadro

Acceso desde	Misma clase	Clases mismo proy.	Sub clases	Clases otro proyecto
`Public`	OK	OK	OK	OK
`Protected Friend`	OK	OK	OK	
`Protected`	OK		OK	
`Friend`	OK	OK		
`Private`	OK			

Modificadores de acceso (miembros)

Resumiendo: tenemos el siguiente cuadro

Acceso desde	Misma clase	Clases mismo proy.	Sub clases	Clases otro proyecto
`Public`	OK	OK	OK	OK
`Protected Friend`	OK	OK	OK	
`Protected`	OK		OK	
`Friend`	OK	OK		
`Private`	OK			

más público

más privado

¿Y si no ponemos modificador de acceso?

- Si no ponemos modificadores de acceso, el acceso por defecto es
- Para clases, **`Friend`**.
- Para atributos, **`Private`**.
- Para métodos y propiedades, **`Public`**.
- Sin embargo, es mucho mejor poner siempre el modificador de acceso: más claro, legible y fácil de mantener.

Una opinión personal (1)

- No es recomendable usar **atributos** que no sean **`Private.`**
- Con un atributo **`Private`**, las comprobaciones de error al acceder quedan incluidas en la clase en que se define el atributo.
- Esto es sencillo, elegante, fácil de mantener y reduce errores.

Una opinión personal (2)

- Con **`Friend`**, todas las clases del proyecto que acceden a un atributo, deben realizar estas comprobaciones de error.
- Con **`Protected`**, todas las subclases deben realizar estas comprobaciones de error.
- Con **`Protected Friend`**, todas las subclases y las clases del proyecto.
- Repetición de código, difícil de mantener, fácil de introducir errores cuando nos olvidamos de las comprobaciones.

Mail: palasi@palasi.com Web: www.palasi.com

Cuando cambio la implementación de un atributo

- Si es **`Friend`**, deben cambiar todas las clases del mismo proyecto que lo acceden.
- Si es **`Protected`**, deben cambiar todas las subclases que lo acceden.
- Esto es bastante crítico: alguien puede haber subclasificado mis clases sin que yo lo sepa.

En resumen

- Utilizar atributos no **`Private`** elimina las ventajas de la ocultación de la implementación.
- Se debe evitar lo máximo posible.

Temario del curso

- 1. Introducción a .NET y Visual Basic .NET
- 2. Un primer programa en VB .NET
- 3. Introducción a la programación orientada a objetos.
- 4. Otros conceptos básicos de VB.NET.
- 5. Más sobre la programación orientada a objetos con VB.NET.
- 6. Conclusión. Los siguientes pasos.

Conclusión

- Este curso ha sido una introducción a los conceptos básicos del lenguaje Visual Basic .NET.
- Dada la complejidad del lenguaje, no se ha podido tratar ni siquiera el lenguaje en toda su totalidad.
- Aún menos, las bibliotecas que lo acompañan y que resultan imprescindibles para programar aplicaciones empresariales.
- VB .NET tiene una curva de aprendizaje más lenta que los lenguajes tradicionales tipo VB6.

Aprendizaje de Visual Basic .NET

- Por comodidad, lo expresaremos en forma de cursos.
- Hay siete cursos, de los cuales 2 son del lenguaje y el resto de las bibliotecas.

Los dos cursos sobre el lenguaje

- **1. Introducción a VB.NET (o "Lenguaje VB .NET parte 1").** Es el curso que hemos dado. Trata los conceptos básicos del lenguaje y de la programación O-O.
- **2. "Lenguaje VB.NET parte 2".** Unas 40 horas. Introduce conceptos más avanzados del lenguaje y de O-O: tratamiento de errores, interfaces, polimorfismo, clases **`MustInherit`**, **`NotInheritable`** y **`Sealed`**, ensamblados, delegados, sombreado, estructuras, constantes, etc.

Mail: palasi@palasi.com Web: www.palasi.com

Los cuatro cursos sobre las bibliotecas

- **3. Clases de la plataforma .NET.** Explica las clases básicas de la plataforma .NET: colecciones, interoperabilidad con código no gestionado, etc.
- **4. Windows Forms en VB.NET.** Trata sobre la interfaz gráfica de escritorio que en este curso apenas hemos tocado. También se estudian los eventos.
- **5. ADO.NET.** Indica como acceder a las bases de datos desde Visual Basic.NET
- **6. ASP.NET.** Indica cómo programar aplicaciones Web desde VB .NET
- **7. Web Services y .NET Remoting.** Indica cómo programar aplicaciones distribuidas.

Conclusión

- Visual Basic .NET es un lenguaje totalmente orientado a objetos y mucho más potente que los lenguajes tradicionales como VB 6.
- El precio que se paga por ello es que VB .NET es más costoso de aprender que estos lenguajes tradicionales.

Acerca de esta presentación

Aurum Solutions

Consultoría internacional en desarrollo de soluciones de software.

Tel: 275-4254
E-mail: info@aurumsol.com