Bloque 2. Visual Basic .NET
UT 3. Confección de interfaces de usuario y creación de componentes visuales

TEMA 8
La programación orientada a objetos.

1. La programación orientada a objetos

Como hemos ido viendo en muchas ocasiones todo .NET Framework está basado en clases que generan objetos. Visual Basic .NET basa su funcionamiento en las clases contenidas en .NET Framework. Debido a esta dependencia en las clases del .NET Framework y sobre todo a la forma "hereditaria" de usarlas, Visual Basic .NET ofrece esta característica sin ningún tipo de restricciones.

La POO es una evolución de la programación por procedimientos llamada también estructurada. Ésta se basaba en funciones, procedimientos y el código que controlaba el flujo de las llamadas a estos.

De la programación estructurada al enfoque orientado a objetos

En la programación estructurada, (con nuestros procedimientos y funciones) el crecimiento de una aplicación hace que el mantenimiento de esta aplicación sea una laboriosa tarea. Fundamentalmente debido a la gran cantidad de procedimientos y funciones que están interrelacionadas entre sí con multitudes de llamadas. Si necesitamos hacer una pequeña modificación nos puede suponer realizar un repaso completo para comprobar a qué funciones puede afectar este cambio. El código se encuentra disperso a lo largo del programa por lo que el mantenimiento se hará progresivamente más difícil.

La organización de una aplicación en POO se realiza mediante estructuras de código que contienen un conjunto de procedimientos e información que ejecutan una serie de procesos destinados a resolver un grupo de tareas. Una aplicación orientada a objetos tendrá tantas estructuras de código como aspectos del programa tengamos que resolver.

Un procedimiento situado dentro de una estructura no podrá llamar sin ser llamado por otro de una estructura distinta sino es bajo una serie de reglas. Lo mismo pasa con los datos, estarán separados del exterior excepto los que designemos. Este concepto de estructura es lo que llamaremos "**Objeto**".

Estos objetos tienen una información precisa y un comportamiento detallado realizado por procedimientos que hacen que puedan clasificarse según el tipo de tarea que realizan.

Veamos ahora los conceptos básicos de la POO.

2. Elementos básicos de la POO

2.1. Las clases

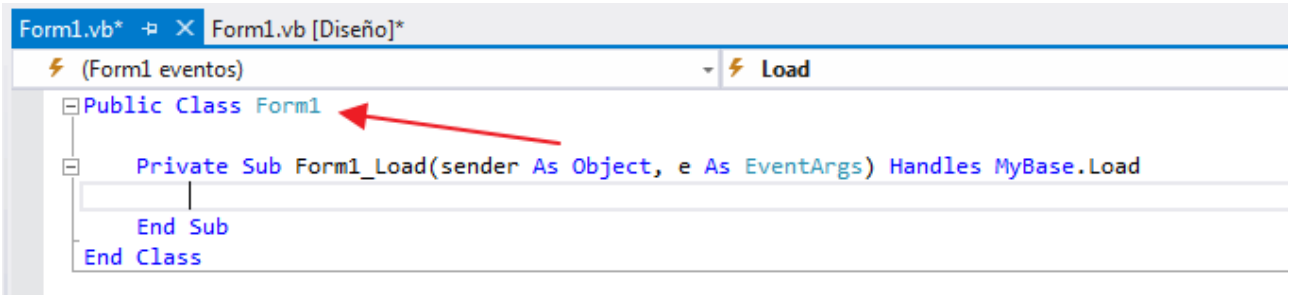
Ya hemos hablado en varias ocasiones de las clases y hemos comentado que todo lo que tiene .NET Framework son clases. Una clase no es ni más ni menos que código. Aunque dicho de esta forma, cualquier programa sería una clase.

Cuando definimos una clase, realmente estamos definiendo dos cosas diferentes: los datos que dicha clase puede manipular o contener y la forma de acceder a esos datos.

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

Veamos de otra forma: una clase es el conjunto de especificaciones o normas que definen cómo se va a crear un objeto, es decir, las instrucciones para crear ese objeto. Una clase es la representación abstracta de algo y el objeto es la representación concreta de lo que una clase define.

Recuerda que incluso los formularios son una clase y así nos lo indica el editor con "Public Class Form1":



Como vemos, el formulario es una clase pública que será accesible por todo el ensamblado (nuestro programa) --> "Public Class Form1". Veamos este concepto de otra forma: una clase es el conjunto de especificaciones o normas que definen cómo se va a crear un objeto, es decir, las instrucciones para crear ese objeto. Una clase es la representación abstracta de algo y el objeto es la representación concreta de lo que una clase define.

2.2. Los Objetos

Por un lado tenemos una clase que es la que define un "algo" con lo que podemos trabajar. Pero para que ese "algo" sea "algo tangible" y podamos trabajar con él tendremos que crearlo. Aquí es cuando entran en juego los objetos, ya que un objeto es una clase que tiene información real.

Digamos que la clase es la "plantilla" a partir de la cual podemos crear un objeto en la memoria. Por ejemplo, podemos tener varios objetos del tipo Cliente, uno por cada cliente que tengamos en nuestra cartera de clientes, pero la clase sólo será una. Dicho de otra forma: podemos tener varias instancias en memoria de una clase. Una instancia es un objeto (los datos) creado a partir de una clase (la plantilla o el código).

En nuestros formularios: tenemos 10 botones que han sido creados a partir de la clase "Botón". La clase es la teoría y el objeto la práctica. El manual de instrucciones para crear el objeto. Y con el ejemplo del coche... podemos crear diferentes coches a partir de la clase "Coche" y cada uno puede tener sus propias propiedades: color, ... pero funcionan todos igual

2.3. Los miembros de una clase

Las clases contienen datos, esos datos suelen estar contenidos en variables. A esas variables cuando pertenecen a una clase, se les llama: campos o propiedades.

Por ejemplo, el nombre de un cliente sería una propiedad de la clase Cliente. Ese nombre lo almacenaremos en una variable de tipo String. De dicha variable podemos decir que es el "campo" de la clase que representa al nombre del cliente.

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

Por tanto, los **miembros** de una clase son las propiedades (los datos) y los **métodos** las acciones a realizar con esos datos.

Como te he comentado antes, el código que internamente usemos para almacenar esos datos o para, por ejemplo, mostrarlos, es algo que no debe preocuparnos mucho, simplemente sabemos que podemos almacenar esa información (en las propiedades de la clase) y que tenemos formas de acceder a ella, (mediante los métodos de dicha clase), eso es "abstracción" o encapsulación.

El proceso por el cual se obtiene un objeto a partir de las especificaciones de una clase se conoce como **instanciación**. Como hemos dicho el molde (la clase) puede crear los objetos con propiedades diferentes: varios botones en el formulario pero con distintos tamaños o colores (propiedades).

3. Los tres pilares de la Programación Orientada a Objetos

Todos los lenguajes basados en objetos deben cumplir estos tres requisitos:

1. **Herencia**
2. **Encapsulación**
3. **Polimorfismo**

Nota: Algunos autores añaden un cuarto requisito: **la abstracción**, pero este último está estrechamente ligado con la encapsulación, ya que, de hecho, es prácticamente lo mismo, aunque ahora lo veremos.

3.1. Herencia

Según la propia documentación de Visual Studio .NET, esta es la característica más importante de la POO.:

Una relación de herencia es una relación en la que un tipo (el tipo derivado) se deriva de otro (el tipo base), de tal forma que el espacio de declaración del tipo derivado contiene implícitamente todos los miembros de tipo no constructor del tipo base.

Un poco más claro.

La herencia es la capacidad de una clase de obtener la interfaz y comportamiento de una clase existente.

Y otra vez:

La herencia es la cualidad de crear clases que estén basadas en otras clases. La nueva clase heredará todas las propiedades y métodos de la clase de la que está derivada, además de poder modificar el comportamiento de los procedimientos que ha heredado, así como añadir otros nuevos.

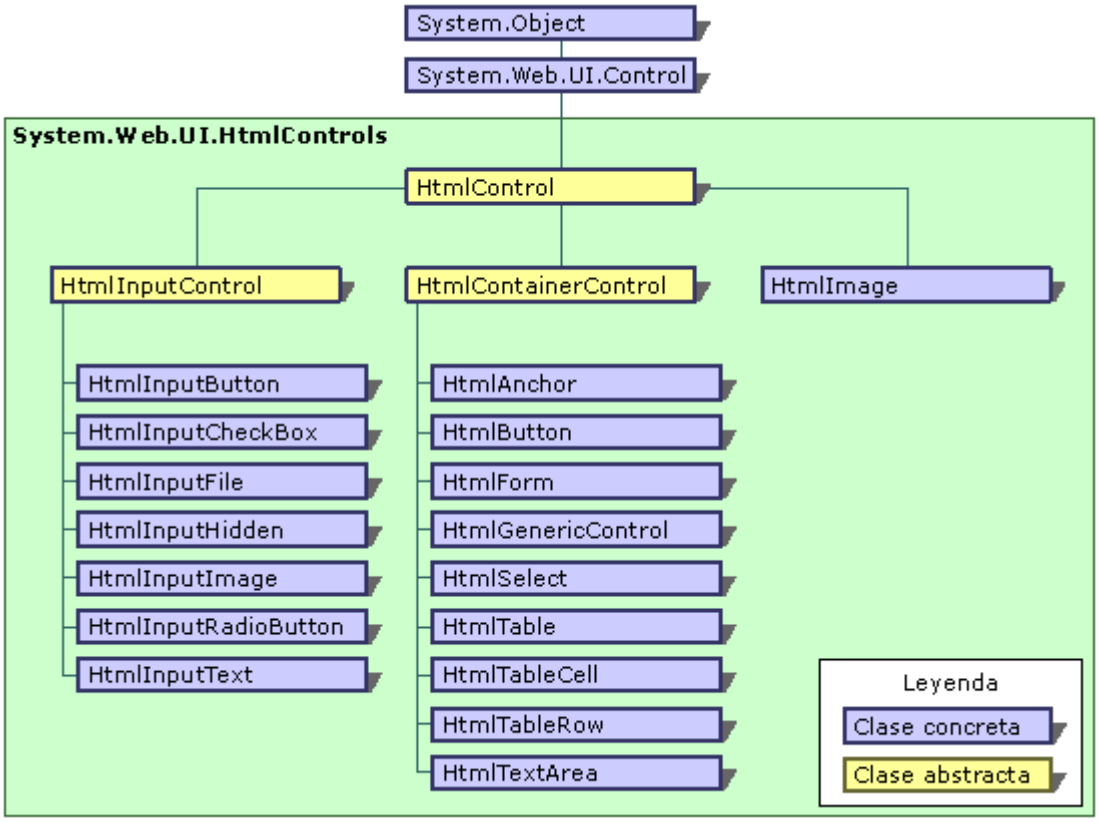
Es decir, tengo una clase que es, por ejemplo, un coche, la herencia me dice que puedo crear un objeto coche a partir de este y que va a "heredar" todas sus características y además podré personalizarlo a mi gusto. Resumiendo: Gracias a la herencia podemos ampliar cualquier clase existente, además de aprovecharnos de todo lo que esa clase haga... De momento la clase es ese "coche" que os he comentado, enseguida veremos la definición de clase.

Veamos otro ejemplo *clásico*: Supongamos que tenemos una clase Gato que está derivada de la clase Animal. El Gato hereda de Animal todas las características comunes a los animales, además de añadirle algunas características particulares a su condición felina. Podemos decir que un Gato es un Animal, lo mismo que un Perro es un Animal, ambos están derivados (han heredado) de la

clase Animal, pero cada uno de ellos es diferente, aunque en el fondo los dos son animales. Esto es herencia: usar una clase base (Animal) y poder ampliarla sin perder nada de lo heredado, pudiendo ampliar la clase de la que se ha derivado (o heredado).

En ocasiones podrás ver los términos de otra forma, por ejemplo otra definición es que partiendo de una clase llamada clase base, padre o superclase (para decir jerárquicamente que es más importante que la otra) podemos crear una nueva clase llamada derivada, hija o subclase. Así de una clase podemos derivar una subclase y otra dentro de esta... creando una jerarquía de clases.

Por ejemplo una jerarquía de clases sería como este gráfico:



Tenemos unos elementos de más nivel que otros. Para referencias a los menores lógicamente tendré que indicar quién es su padre jerárquicamente.

Hay dos tipos de herencia:

- **Simple**. Cuando creamos una clase a partir de una sola clase base.
- **Múltiple**. Cuando creamos una clase a partir de varias clases base. Realmente complejo y no permitido por .NET.

Repitiendo el ejemplo del coche, es una clase que tiene como todas las clases sus propiedades, métodos y eventos, por ejemplo.

- Propiedades: color del coche, modelo, tapicería, potencia,... es decir, sus características

- Métodos: Acelerar, frenar, ...
- Eventos: está claro que está esperando muchos eventos, que se toque el claxon, la radio, una puerta... Al dispararse un evento ejecutará una serie de acciones o un método.

Teniendo definida una clase, podemos crear un "cochedeportivo" a partir de ella. Va a heredar todas sus características pero puedo modificar algunas cosas y añadirles otras por ejemplo un turbo o ABS.

3.2. Encapsulación

Como en el caso anterior, veamos qué nos dice Visual Studio.NET sobre esta cualidad de la programación orientada a objetos:

La encapsulación es la capacidad de contener y controlar el acceso a un grupo de elementos asociados. Las clases proporcionan una de las formas más comunes de encapsular elementos.

La encapsulación es la capacidad de separar la implementación de la interfaz de una clase del código que hace posible esa implementación. Esto realmente sería una especie de abstracción, ya que no nos importa cómo esté codificado el funcionamiento de una clase, lo único que nos debe interesar es cómo funciona.

Digamos que una clase es un elemento de Windows que como ya hemos visto tiene unas propiedades (color, fuente), métodos y eventos (clic). Cuando decimos la implementación de la interfaz de una clase, nos referimos a los miembros de esa clase: métodos, propiedades, eventos, etc. Es decir, lo que la clase es capaz de hacer.

Cuando usamos las clases, éstas tienen una serie de características (los datos que manipula) así como una serie de comportamientos (las acciones a realizar con esos datos). La encapsulación es esa capacidad de la clase de ocultarnos sus interioridades para que sólo veamos lo que tenemos que ver, sin tener que preocuparnos de cómo está codificada para que haga lo que hace. Simplemente nos debe importar que lo hace.

Si tomamos el ejemplo de la clase Gato, sabemos que araña, come, se mueve, etc., pero el cómo lo hace no es algo que deba preocuparnos.

Al final el concepto es más sencillo de lo que parece. Sólo es la capacidad de separar por un lado lo que nos muestra que hace ese objeto con el código interno que lo realiza. Así para nosotros un objeto por muy complejo que sea sólo veremos sus propiedades, métodos y eventos desde el propio IDE. Aplica esto a la clase botón. Sé que tiene esas propiedades, métodos y eventos pero no veo nada del código que realiza esas acciones, es decir, está **encapsulado.** Dicho finalmente de forma técnica: **Establece la separación entre la interfaz del objeto y su implementación.**

Esto aporta dos ventajas:

- Seguridad. Se evitan los accesos no deseados, es decir, si está bien encapsulada no podremos modificar directamente variables ni ejecutar métodos que sean de uso interno.
- Simplicidad. Un programador no debe conocer las tripas de cómo está hecho sino simplemente se limitará a utilizarlo.

Siguiendo con el ejemplo del coche, cuando lo ponemos en marcha simplemente giramos la llave sin saber qué hace dentro del capó. Lo mismo aplicado al código, tenemos una clase que dibuja un gráfico, éste tiene un método que realiza el dibujo, en nuestro programa llamaremos a éste método y ya está. No sabemos cómo se las arregla para dibujar líneas, escribir textos, rellenar superficies,...

3.3. Polimorfismo

El polimorfismo se refiere a la posibilidad de definir múltiples clases con funcionalidad diferente, pero con métodos o propiedades denominados de forma idéntica, que pueden utilizarse de manera intercambiable mediante código cliente en tiempo de ejecución.

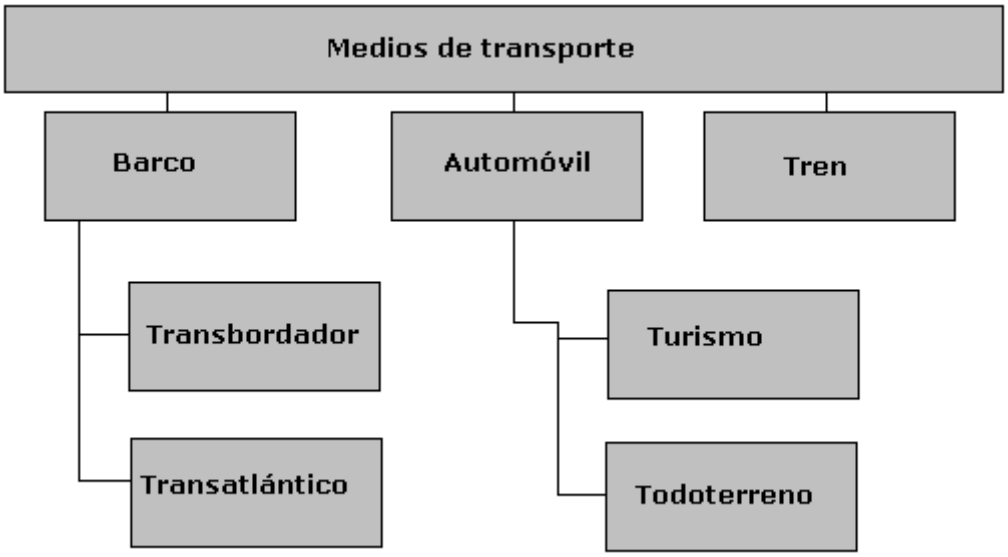
Significa que puede tener múltiples clases que se pueden utilizar de forma intercambiable, si bien cada clase implementa las mismas propiedades o los mismos métodos de maneras diferentes. El polimorfismo es importante en la programación orientada a objetos puesto que permite usar elementos que tienen el mismo nombre, independientemente del tipo de objeto que se esté utilizando en ese momento. Por ejemplo, dada una clase base "Coche", el polimorfismo permite al programador definir diferentes métodos "Iniciar_motor" para cualquier número de clases derivadas. El método "Iniciar_motor" de una clase derivada denominada "Coche_Diesel" puede ser totalmente diferente del método con el mismo nombre en la clase base. Otros procedimientos o métodos pueden usar el método "Iniciar_motor" de las clases derivadas de la misma forma, con independencia del tipo de objeto "coche" que se esté utilizando en ese momento.

El Polimorfismo nos permite usar miembros de distintas clases de forma genérica sin tener que preocuparnos de si pertenece a una clase o a otra. Siguiendo con el ejemplo de los animales, si el Gato y el Perro pueden morder podríamos tener un "animal" que muerda sin importarnos que sea el Gato o el Perro, simplemente podríamos usar el método Morder ya que ambos animales tienen esa característica "animal mordedor".

Dicho de otra forma, el polimorfismo determina que el mismo nombre de método realizará diferentes acciones según el objeto sobre el que se ha aplicado. Aplicado a la programación simplemente significa que en dos objetos: ventana y archivo al utilizar un mismo método se aplicará un resultado diferente. Es decir, si aplicamos el método "Abrir" obviamente es el mismo nombre pero realiza cosas diferentes dependiendo del objeto que lo llame: en un caso se abrirá una ventana y en el otro un fichero.

4. Jerarquía de clases

Es una de las formas típicas de ver las superclases y subclases: en forma de árbol de clases o jerarquía:



La nomenclatura, como hemos visto en otras ocasiones, sería jerárquica: transportes.barco.transbordador para hacer referencia al nombre completo del objeto. En ocasiones si el nombre del objeto es único VB.NET es capaz de encontrar su sitio correcto, es decir, si en el código hacemos referencia a "turismo", como sólo existe un objeto con ese nombre en nuestra jerarquía de objetos no hay problema en trabajar con él sin escribir el nombre completo. En cambio, si el nombre existe en varios sitios tendríamos que poner el nombre completo para romper esa ambigüedad.

4.1. Relaciones entre objetos

Los objetos se comunican entre sí mediante una serie de relaciones que vamos a comentar ahora:

4.1.1. Herencia

Ya vimos antes que era la característica más importante de la POO y que permite crear una clase derivada de otra principal. Esta nueva clase tiene todos los métodos y propiedades de la original y además otros nuevos que podemos añadir. En el código comprobaremos la relación de herencia "es un": "un todoterreno es un automóvil" es cierto pero "un todoterreno es un barco" devolvería falso.

4.1.2. Pertenencia

Los objetos pueden estar formados a su vez por otros objetos. Un objeto coche puede estar formado por un objeto "motor" o un objeto "dirección". En este caso hay una relación de

pertenencia puesto que existe un conjunto de objetos que pertenecen a otro objeto o se unen para formar otro objeto.

En el código lo representaremos como "tiene un". Un "coche tiene un motor" sería cierto pero "una bicicleta tiene un motor" sería falso.

4.1.3. Utilización

En ocasiones un objeto utiliza a otro para alguna operación sin implicar ninguna pertenencia entre ellos. Por ejemplo un objeto Ventana puede utilizar un objeto Gráfico para mostrar el dibujo pero no es necesario que el objeto Grafico sea propiedad del objeto Ventana. La notación en el código es "usa un": el objeto Grafico usa una ventana sería cierto.

4.1.4. Reutilización

Finalmente si el objeto ha sido bien diseñado puede ser reutilizado por otra aplicación de forma directa o creando una clase derivada de él. Este es uno de los objetivos principales de la POO: aprovechar el código escrito.

5. Caso práctico

5.1. Crear o definir una clase

Al igual que existen instrucciones para declarar o definir una variable o cualquier otro elemento de un programa de Visual Basic, existen instrucciones que nos permiten crear o definir una clase.

Para crear una clase debemos usar la instrucción **Class** seguida del nombre que tendrá dicha clase, por ejemplo:

```
Class Cliente
```

A continuación escribiremos el código que necesitemos para implementar las propiedades y métodos de esa clase, pero para que Visual Basic sepa que ya hemos terminado de definir la clase, usaremos una instrucción de cierre:

```
End Class
```

Por tanto, todo lo que esté entre `Class <nombre>` y `End Class` será la definición de dicha clase.

Definir los miembros de una clase

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

```
Class Cliente

  Public Nombre As String

  Sub Mostrar()
        Console.WriteLine("El nombre del cliente: {0}", Nombre)
  End Sub

End Class
```

En este caso, la línea ***Public Nombre As String***, estaría definiendo una propiedad o "campo" público de la clase Cliente.

Por otro lado, el procedimiento Mostrar sería un método de dicha clase, en esta caso, nos permitiría mostrar la información contenida en la clase Cliente.

Esta es la forma más simple de definir una clase. Por tanto, podemos comprobar que es sencillo definir una clase con sus miembros.

Ahora vamos a crear objetos con esta clase:

Crear un objeto a partir de una clase

Las clases definen las características y la forma de acceder a los datos que contendrá, pero sólo eso: los define. Para que podamos asignar información a una clase y poder usar los métodos de la misma, tenemos que crear un objeto basado en esa clase, o lo que es lo mismo: tenemos que crear una nueva instancia en la memoria de dicha clase. Para ello, haremos lo siguiente:

Definimos una variable capaz de contener un objeto del tipo de la clase, esto lo haremos como con cualquier variable:

```
Dim cli As Cliente
```

Pero, a diferencia de las variables basadas en los tipos vistos hasta ahora, para poder crear un objeto basado en una clase, necesitamos algo más de código que nos permita "crear" ese objeto en la memoria, ya que con el código usado en la línea anterior, simplemente estaríamos definiendo una variable que es capaz de contener un objeto de ese tipo, pero aún no existe ningún objeto en la memoria, para ello tendremos que usar el siguiente código:

```
cli = New Cliente()
```

Con esto le estamos diciendo a Visual Basic: crea un **nuevo** objeto en la memoria del tipo Cliente.

Estos dos pasos los podemos simplificar de la siguiente forma:

```
Dim cli As New Cliente()
```

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

Una vez definido y creado nos queda saber cómo acceder al objeto:

Acceder a los miembros de una clase

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

```
cli.Nombre = "Guillermo"
```

Es decir, de la misma forma que haríamos con cualquier otra variable, pero indicando el objeto al que pertenece dicha variable. Como podrás comprobar, esto ya lo hemos estado usando anteriormente tanto en la clase Console como en las otras clases que hemos usado en temas anteriores, incluyendo los arrays.

Y para acceder al método Mostrar:

```
cli.Mostrar()
```

Igual que antes, pero en este caso estamos llamando a un método de la clase que internamente en un procedimiento que nos muestra un dato.

5.2. Ejemplo de cómo usar la herencia

Para poder usar la herencia en nuestras clases, debemos indicar a Visual Basic que esa es nuestra intención. Para ello disponemos de la instrucción **Inherits**, la cual se usa seguida del nombre de la clase de la que queremos heredar. Veamos un ejemplo.

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

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

Veamos el código de estas dos clases. Para ello crea un nuevo proyecto del tipo consola y escribe estas líneas al principio o al final del fichero que el IDE añade de forma predeterminada.

```
Class Cliente
  Public Nombre As String
  Sub Mostrar()
          Console.WriteLine("El nombre del cliente: {0}", Nombre)
  End Sub
End Class

Class ClienteMoroso
  Inherits Cliente
  Public Deuda As Decimal
End Class
```

Para que la clase ClienteMoroso herede la clase Cliente, hemos usado **Inherits Cliente**. Con esta línea, (la cual debería ser la primera línea de código después de la declaración de la clase), le estamos indicando a VB que nuestra intención es poder tener todas las características que la clase Cliente tiene. Haciendo esto, añadiremos a la clase ClienteMoroso la propiedad Nombre y el método Mostrar, aunque también tendremos la nueva propiedad que hemos añadido: Deuda.

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

```
Module Module1

Sub Main()
  Dim cli As New Cliente()
  Dim cliM As New ClienteMoroso()
  '
       cli.Nombre = "Guille"
       cliM.Nombre = "Pepe"
       cliM.Deuda = 2000
  '
       Console.WriteLine("Usando Mostrar de la clase Cliente")
       cli.Mostrar()
  '
       Console.WriteLine("Usando Mostrar de la clase ClienteMoroso")
       cliM.Mostrar()
  '
       Console.WriteLine("La deuda del moroso es: {0}", cliM.Deuda)
  '
      Console.ReadLine()
End Sub
End Module
```

Lo que hemos hecho es crear un objeto basado en la clase Cliente y otro basado en ClienteMoroso. Le asignamos el nombre a ambos objetos y a la variable **cliM** (la del ClienteMoroso) le asignamos un valor a la propiedad Deuda.

Fíjate que en la clase ClienteMoroso no hemos definido ninguna propiedad llamada Nombre, pero esto es lo que nos permite hacer la herencia: heredar las propiedades y métodos de la clase base.

Por tanto, podemos usar esa propiedad como si la hubiésemos definido en esa clase. Lo mismo ocurre con los métodos, el método Mostrar no está definido en la clase ClienteMoroso, pero sí que lo está en la clase Cliente y como resulta que ClienteMoroso hereda todos los miembros de la clase Cliente, también hereda ese método.

La salida de este programa sería la siguiente:

```
Usando Mostrar de la clase Cliente
El nombre del cliente: Guille
Usando Mostrar de la clase ClienteMoroso
El nombre del cliente: Pepe
La deuda del moroso es: 2000
```

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

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

```
Sub Main()
Dim cli As Cliente
Dim cliM As New ClienteMoroso()
        '
        cliM.Nombre = "Pepe"
        cliM.Deuda = 2000
        cli = cliM
        '
        Console.WriteLine("Usando Mostrar de la clase Cliente")
        cli.Mostrar()
        '
        Console.WriteLine("Usando Mostrar de la clase ClienteMoroso")
        cliM.Mostrar()
        '
        Console.WriteLine("La deuda del moroso es: {0}", cliM.Deuda)
        '
        Console.ReadLine()
End Sub
```

En este caso, la variable **cli** simplemente se ha declarado como del tipo Cliente, pero no se ha creado un nuevo objeto, simplemente hemos asignado a esa variable el contenido de la variable **cliM**.

Con esto lo que hacemos es asignar a esa variable el contenido de la clase ClienteMoroso, pero como comprenderás, la clase Cliente "no entiende" nada de las nuevas propiedades implementadas en la clase derivada, por tanto, sólo se podrá acceder a la parte que es común a esas dos clases: la parte heredada de la clase Cliente.

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

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

```
Console.WriteLine()
cli.Nombre = "Juan"
Console.WriteLine("Después de asignar un nuevo nombre a cli.Nombre")
cli.Mostrar()
cliM.Mostrar()
```

La salida de este nuevo código, sería la siguiente:

```
Usando Mostrar de la clase Cliente
El nombre del cliente: Pepe

Usando Mostrar de la clase ClienteMoroso
El nombre del cliente: Pepe
La deuda del moroso es: 2000

Después de asignar un nuevo nombre a cli.Nombre
El nombre del cliente: Juan
El nombre del cliente: Juan
```

Como puedes comprobar, al cambiar en una de las variables el contenido de la propiedad Nombre, ese cambio afecta a las dos variables, pero eso no es porque haya nada mágico ni ningún fallo, es por lo que te comenté antes: sólo existe un objeto en la memoria y las dos variables acceden al mismo objeto, por tanto, cualquier cambio efectuado en ellas, se reflejará en ambas variables por la sencilla razón de que sólo hay un objeto en memoria.

A este tipo de variables se las llama variables por referencia, ya que hacen referencia o apuntan a un objeto que está en la memoria. A las variables que antes hemos estado viendo se las llama variables por valor, ya que cada una de esas variables tiene asociado un valor que es independiente de los demás. (Esto ya lo vimos al ver las variables en los procedimientos)

5.3. Sobrecargar el constructor de las clases

Una de las utilidades más prácticas de la sobrecarga de procedimientos es sobrecargar el constructor de una clase. Ahora veremos esto del constructor de la clase...

El constructor de una clase es un procedimiento de tipo Sub llamado New, dicho procedimiento se ejecuta cada vez que creamos un nuevo objeto basado en una clase. Si al declarar una clase no escribimos el "constructor", será el compilador de Visual Basic .NET el que se encargará de escribir uno genérico.

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

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

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

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

Esta clase nos permite crear nuevos objetos del tipo **Cliente** de dos formas. Por ejemplo, si tenemos una variable llamada **cli**, declarada de esta forma:

```
Dim cli As Cliente
```

podemos crear nuevas instancias sin indicar ningún parámetro:

```
cli = New Cliente()
```

o indicando un parámetro, el cual se asignará a la propiedad Nombre de la clase:

```
cli = New Cliente("Guillermo")
```

5.4. Los campos y las propiedades.

Recordamos que los campos son variables usadas a nivel de una clase: los campos representan los datos de la clase. Recuerda que la razón de ser de una clase es poder manipular cierta información: los campos y propiedades representan los datos manipulados por la clase y los métodos manipulan (o permiten manipular) esos datos.

Los campos representan la información que "internamente" la clase manipulará. Dependiendo de que esos campos sean accesibles sólo dentro de la clase o también puedan ser accedidos desde cualquier instancia, (creada en la memoria), de la clase, estarán declarados de una forma o de otra. Para simplificar, después entraremos en detalle, podemos declarar cualquier *miembro de una clase* de dos formas, según el nivel de visibilidad o ámbito que queramos que tenga.

Si lo declaramos con el modificador de acceso **Private**, ese miembro sólo será accesible desde "dentro" de la clase, es decir, en cualquier sitio de la clase podremos usar ese miembro, pero no será accesible desde "fuera" de la clase, por ejemplo, en una nueva instancia creada. Por otro lado, si declaramos un miembro de la clase como **Public**, ese miembro será accesible tanto desde

dentro de la clase como desde fuera de la misma. Un miembro público de una clase siempre será accesible.

Nota: Cuando decimos *miembro de una clase*, me refiero a cualquier campo, propiedad, enumeración, método o evento.

Además del modificador Private y Public hay otros, que seguramente no veremos.

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

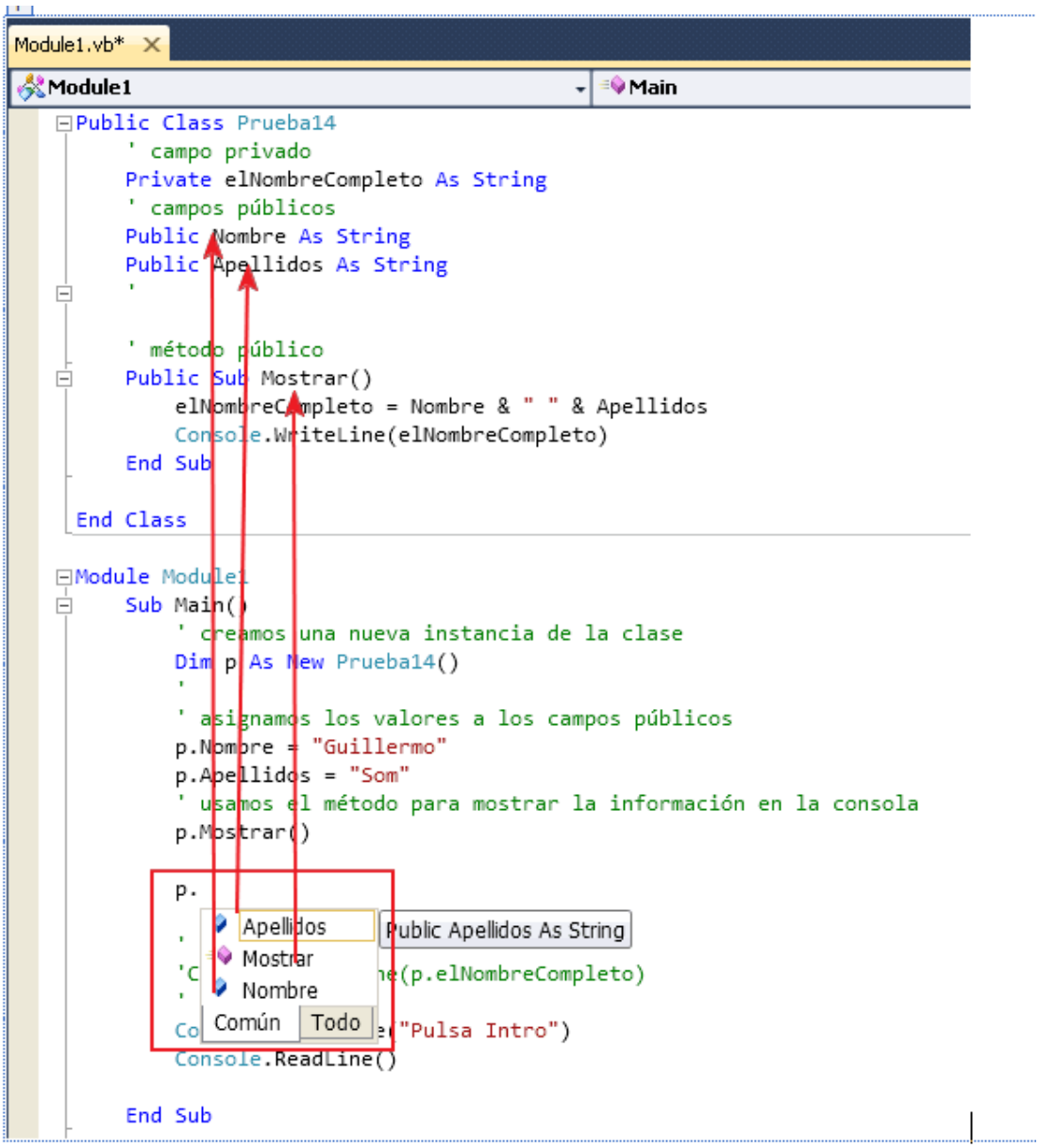
Veamos un ejemplo. En el siguiente código vamos a declarar una clase que tendrá tres miembros públicos y uno privado. De estos tres miembros públicos, dos de ellos serán campos y el tercero será un método que nos permitirá mostrar por la consola el contenido de esos campos. El campo privado simplemente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

```
Public Class Prueba14
  ' campo privado
  Private elNombreCompleto As String
  ' campos públicos
  Public Nombre As String
  Public Apellidos As String
  '
  ' método público
  Public Sub Mostrar()
        elNombreCompleto = Nombre & " " & Apellidos
        Console.WriteLine(elNombreCompleto)
  End Sub
End Class

Module Module1
  Sub Main()
    ' creamos una nueva instancia de la clase
    Dim p As New Prueba14()
    '
    ' asignamos los valores a los campos públicos
    p.Nombre = "Guillermo"
    p.Apellidos = "Som"
    'usamos el método para mostrar la información en la consola
    p.Mostrar()
    '
    'esto dará error
    'Console.WriteLine(p.elNombreCompleto)
    '
    Console.WriteLine("Pulsa Intro")
    Console.ReadLine()
End Sub
End Module
```

En la clase **Prueba14** (que está declarada como Public) tenemos declarado **Nombre** y **Apellidos** con el modificador Public, por tanto, podemos acceder a estos dos campos desde una nueva instancia de la clase, así como desde cualquier sitio de la clase. Lo mismo es aplicable al método **Mostrar**, al ser público se puede usar desde la variable declarada en el procedimiento Main.

Veamos este ejemplo en el IDE:



Simplemente hemos escrito "p." para que el "intellisense" me muestre las posibilidades del objeto "p" que es del tipo "prueba14". Como ves tiene dos propiedades: Nombre y Apellidos y un método "Mostrar". No aparece "elnombrecompleto" ya que es de tipo "Private" y, por tanto, interno para la clase y no accesible desde el exterior.

Por tanto, el campo **elNombreCompleto** está declarado como Private y, como hemos comprobado, sólo será accesible desde la propia clase, (esto se demuestra, porque se usa en el método Mostrar), y no desde fuera de ella, es decir, no podremos usar ese campo desde la

instancia creada en Main por la sencilla razón de que es "privada" y, por tanto, no visible ni accesible desde fuera de la propia clase. Si quitas el comentario que muestra por la consola el contenido del campo privado, comprobarás que dará un error en tiempo de compilación, ya que **elNombreCompleto** no es accesible por estar declarado como privado.

En Visual Basic .NET **se pueden considerar a los campos públicos como si fuesen propiedades de una clase.**

Pero, realmente, las propiedades son otra cosa diferente, al menos así deberíamos planteárnoslo. Además de declarar una propiedad usando la declaración de un campo público, también podemos usar la instrucción **Property** para que quede más claro cuál es nuestra intención.

Como veremos a continuación, el uso de Property nos dará algo más de trabajo, pero también nos proporcionará mayor control sobre lo que se asigne a esa propiedad. Por ejemplo, imagínate que no se debería permitir que se asigne una cadena vacía al Nombre o al Apellido. En el ejemplo anterior, esto no sería posible de "controlar", ya que al ser "variables" públicas no tenemos ningún control sobre cómo ni cuándo se asigna el valor, por tanto, salvo que creemos unos métodos específicos para asignar y recuperar los valores de esas dos "propiedades", no tendríamos control sobre la asignación de una cadena vacía.

Vamos a ver cómo se podría "controlar" esto que acabo de comentar. Para simplificar, sólo vamos a comprobar si al nombre se le asigna una cadena vacía. Para ello, vamos a crear un procedimiento que se encargue de asignar el valor al nombre y otro que se encargue de recuperar el contenido de esa "propiedad".

Estos métodos se llamarán **AsignaNombre** y **RecuperaNombre**. El primero nos permitirá asignar un nuevo nombre y el segundo nos permitirá recuperar el contenido de esa "propiedad".

```
Public Class Prueba14_2
  ' variable (campo) privado para guardar el nombre
  Private elNombre As String
  '
  ' campo público para los apellidos
  Public Apellidos As String
  '
  ' método que permite asignar el nuevo nombre
  Public Sub AsignarNombre(ByVal nuevoNombre As String)
         ' comprobamos si no es una cadena vacía
         If nuevoNombre <> "" Then
            elNombre = nuevoNombre
         End If
  End Sub
  ' método que recupera el nombre asignado
  Public Function RecuperarNombre() As String
         Return elNombre
  End Function
```

```
    '
    ' método público para mostrar el nombre completo
    Public Sub Mostrar()
          Console.WriteLine(elNombre & " " & Apellidos)
    End Sub
End Class

Module Module1
    Sub Main()
        ' creamos una nueva instancia de la clase
        Dim p As New Prueba14_2()
            '
        ' asignamos los valores a los campos públicos
        p.AsignarNombre("Guillermo")
        p.Apellidos = "Som"
        'usamos el método para mostrar la información en la consola
        p.Mostrar()
        '
        Console.WriteLine("El nombre es: " & p.RecuperarNombre)
        '
        Console.WriteLine()
        Console.WriteLine("Pulsa Intro")
        Console.ReadLine()
    End Sub
End Module
```

Declaramos una variable privada (un campo privado) llamado **elNombre**. Este campo nos permitirá almacenar el nombre que se asigne usando el método **AsignarNombre**, el cual recibirá un parámetro, antes de asignar el contenido indicado en ese parámetro, comprobamos si es una cadena vacía, de ser así, no hacemos nada, ya que sólo se asigna el nuevo valor si el contenido del parámetro es distinto de una cadena vacía, con esto conseguimos lo que nos proponíamos: no asignar al nombre una cadena vacía.

Por otro lado, si queremos acceder al contenido del "nombre", tendremos que usar el método **RecuperarNombre**. Esta función simplemente devolverá el contenido del campo privado **elNombre**.

En cuanto al campo **Apellidos**, (el cual actuará como una propiedad, es decir, representa a un dato de la clase), éste permanece sin cambios. Por tanto, no se hace ninguna comprobación y podemos asignarle lo que queramos, incluso una cadena vacía.

Esto está muy bien, pero no es "intuitivo", lo lógico sería poder usar la propiedad **Nombre** tal como la usamos en el primer ejemplo, pero de forma que no permita asignarle una cadena vacía. Esto lo podemos conseguir si declaramos **Nombre** como un procedimiento de tipo **Property**. La forma de usarlo sería como en el segundo ejemplo, pero en lugar de usar **AsignarNombre** o **RecuperarNombre**, usaremos **Nombre**, que es más intuitivo, aunque, como comprobarás, "internamente" es como si usáramos los dos procedimientos que acabamos de ver.

5.5. ¿Cómo declarar una propiedad como un procedimiento Property?

Para estos casos, el compilador de Visual Basic .NET pone a nuestra disposición una instrucción, que al igual que Sub o Function, nos permiten declarar un procedimiento que tiene un trato especial, este es el caso de **Property**.

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

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

```
Dim s As String = p.Nombre
```

O como esto otro:

```
Console.WriteLine(p.Nombre)
```

En estos dos casos, lo que queremos es recuperar el contenido de la propiedad. Pero si lo que queremos es asignar un nuevo valor, esa propiedad normalmente estará a la izquierda de una asignación. Sería el caso de hacer esto: **p.Nombre = "Guillermo"**, en este caso estamos asignando un nuevo valor a la propiedad Nombre.

Pero... ¿cómo se declara un procedimiento Property? Veamos.

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

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

Es decir, declaramos un procedimiento del tipo **Property**, que tiene dos bloques internos:

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

Como puedes comprobar, el bloque Set recibe un parámetro llamado **Value** que es del mismo tipo que la propiedad, en este caso de tipo String. Value representa el valor que queremos asignar a la propiedad y representará lo que esté a la derecha del signo igual de la asignación. Por ejemplo, si tenemos esto: **p.Nombre = "Guillermo"**, "Guillermo" será lo que Value contenga.

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

Nota: En el caso de Visual Basic .NET, el parámetro indicado en el bloque Set se puede llamar como queramos. En C#, siempre se llamará **value**, además de que no se indica el parámetro en el bloque **set**. Por esa razón, se recomienda dejar el nombre Value, que es el que el VB .NET utiliza automáticamente cuando declaramos un procedimiento de tipo Property.

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

Lógicamente el tipo de datos del campo privado debe ser del mismo tipo que el de la propiedad.

La ventaja de usar propiedades declaradas como Property en lugar de usar variables (o campos) públicos es que podemos hacer comprobaciones u otras cosas dentro de cada bloque Get o Set, tal como hemos hecho en el ejemplo de la propiedad Nombre para que no se asigne una cadena vacía al Nombre.

De todas formas, no es recomendable hacer mucho "trabajo" dentro de una propiedad, sólo lo justo y necesario. Si nuestra intención es que dentro de una propiedad se ejecute un código que pueda consumir mucho tiempo o recursos, deberíamos plantearnos crear un método, ya que las propiedades deberían asignar o devolver los valores de forma rápida.

Debido a que en Visual Basic .NET los campos públicos son tratados como propiedades, no habría demasiada diferencia en crear una propiedad declarando una variable pública o usando un procedimiento Property, pero deberíamos acostumbrarnos a crear procedimientos del tipo

Property si nuestra intención es crear una propiedad, además de que el uso de procedimientos Property nos da más juego que simplemente declarando una variable pública.

5.6. Propiedades de sólo lectura.

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

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

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

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

5.7. Propiedades de sólo escritura.

De la misma forma que podemos definir una propiedad de sólo lectura, también podemos crear una propiedad de sólo escritura, es decir, una propiedad que sólo aceptará que se asignen nuevos valores, pero que no permitan obtener el valor que tienen. La verdad es que este tipo de propiedades no son muy habituales, pero podemos hacerlo. Veamos cómo tendríamos que declarar una propiedad de sólo escritura.

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

Es decir, usamos el modificador `WriteOnly` al declarar la propiedad y sólo debemos especificar el bloque `Set`. Si declaramos un procedimiento `WriteOnly Property` no podemos indicar el bloque `Get`, ya que eso dará error.

Cuando declaramos una propiedad de sólo lectura no podemos declarar otra propiedad con el mismo nombre que sólo sea de escritura. Si nuestra intención es crear una propiedad de lectura/escritura, simplemente declaramos la propiedad sin indicar ni `ReadOnly` ni `WriteOnly`.

5.8. Campos de sólo lectura.

Lo mismo que existen propiedades de sólo lectura, podemos crear campos de los que sólo podamos leer el valor que contiene y no asignar ninguno nuevo. Aunque en principio podrías pensar que con una constante se podría conseguir lo mismo que con un campo de solo lectura. Como verás puede haber una pequeña diferencia de la que nos podemos aprovechar.

Para aclarar esto último, recordad que una constante es como una variable que tiene un valor fijo, es decir, una vez que se ha asignado, no se puede cambiar. Los campos de sólo lectura a diferencia de las constantes, se pueden cambiar de valor, pero sólo en la definición, lo cual no se diferenciaría de la forma de declarar una constante, o dentro del constructor de la clase.

Esto último es algo que no se puede hacer con una constante, ya que **las constantes siempre tienen el mismo valor**, el cual se asigna al declararla.

Veamos un par de ejemplos. En el siguiente código vamos a declarar una constante y también un campo (o variable) de sólo lectura.

```
Public Const PI As Double = 3.14159
Public ReadOnly LongitudNombre As Integer = 50
```

En este código tenemos declarada una constante llamada **PI** que tiene un valor fijo. Las constantes siempre deben declararse con el valor que contendrán.

Por otro lado, tenemos un campo de sólo lectura llamado **LongitudNombre**, que es del tipo Integer y tiene un valor de 50. Como podrás comprobar, en este caso no hay diferencia entre una constante y un campo de sólo lectura. La verdad es que si lo dejamos así, no habría ninguna diferencia entre una declaración y otra.

Bueno, sí hay diferencia. Cuando declaramos una constante pública, ésta estará accesible en las nuevas instancias de la clase además de ser accesible "globalmente", es decir, no tendremos que crear una nueva instancia de la clase para poder acceder a la constante. Por tanto, podríamos decir que las constantes declaradas en una clase son "variables" compartidas por todas las instancias de la clase. Es como si declarásemos la constante usando **Shared** o como si estuviese declarada en una clase de tipo Module. Como ya vimos anteriormente, la diferencia entre una clase de tipo Module y una de tipo Class es que en la primera, todos los miembros están compartidos (Shared), mientras que en la segunda, salvo que se indique explícitamente, cada miembro pertenecerá a la instancia de la clase, es decir, de cada objeto creado con New. De esto hablaremos más en profundidad en otra ocasión.

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

```
Public Shared ReadOnly LongitudNombre As Integer = 50
```

En este caso, no habría diferencia con una constante. Pero, esta no sería la forma habitual de declarar un campo de sólo lectura. Lo habitual es declararlo sin un valor inicial, aunque haciéndolo así nos aseguramos que tenga un valor predeterminado, en caso de que no se asigne ninguno nuevo.

Como comentamos hace unas líneas, la forma de asignar el valor que tendrá un campo de sólo lectura, sería asignándolo en el constructor de la clase. Por tanto, podríamos tener un constructor (Sub New) que reciba como parámetro el valor que tendrá ese campo de sólo lectura.

En el siguiente código vamos a declarar una clase que tendrá un campo de sólo lectura, el cual se asigna al crear una nueva instancia de la clase.

```
Public Class Prueba14_6
  Public ReadOnly LongitudNombre As Integer = 50
  '
  Public Sub New()
  '
  End Sub
  Public Sub New(ByVal nuevaLongitud As Integer)
         LongitudNombre = nuevaLongitud
  End Sub
  '

  ' Este será el punto de entrada del ejecutable
  Public Shared Sub Main()
   '
   ' si creamos la clase sin indicar la nueva longitud...
   Dim p As New Prueba14_6()
   ' el valor será el predeterminado: 50
   Console.WriteLine("p.LongitudNombre = {0}", p.LongitudNombre)
   '
   ' si creamos la clase indicando la nueva longitud...
   Dim p1 As New Prueba14_6(25)
   ' el valor será el indicado al crear la instancia
   Console.WriteLine("p1.LongitudNombre = {0}", p1.LongitudNombre)
   '
   Console.WriteLine()
   Console.WriteLine("Pulsa Intro")
   Console.ReadLine()
  End Sub
End Class
```

Esta clase tiene definidos dos constructores: uno sin parámetros y otro que recibe un valor de tipo Integer, ese valor será el que se use para el campo de sólo lectura. En el Sub Main, el cual está declarado como Shared para que se pueda usar como punto de entrada del ejecutable, declaramos dos objetos del tipo de la clase, el primero se instancia usando New sin ningún parámetro, mientras que en el segundo se crea la nueva instancia indicando un valor en el

constructor. Ese valor será el que se utilice para darle valor al campo de sólo lectura, cosa que se demuestra en la salida del programa:

```
p.LongitudNombre = 50
p1.LongitudNombre = 25
```

En los comentarios está aclarado porqué el objeto **p** toma el valor 50 y porqué usando **p1** el valor es 25.

Así que, una vez que hemos asignado el valor al campo de sólo lectura, ya no podemos modificar dicho valor, salvo que esa modificación se haga en el constructor. Por tanto, sólo podemos asignar un nuevo valor a un campo de sólo lectura en el constructor de la clase. Dicho esto, si nuestras mentes fuesen tan retorcidas como para hacer esto, sería factible y no produciría ningún error:

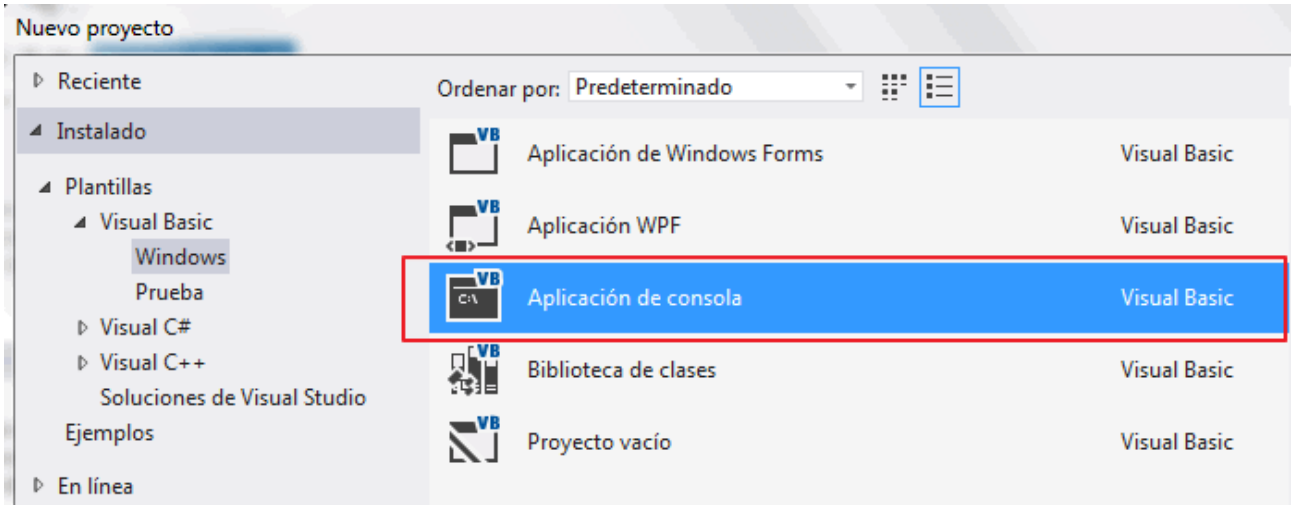
```
Public Sub New(ByVal nuevaLongitud As Integer)
    LongitudNombre = nuevaLongitud
    LongitudNombre = LongitudNombre + 10
End Sub
```

Por la sencilla razón de que "dentro" del constructor podemos asignar un nuevo valor al campo de sólo lectura.

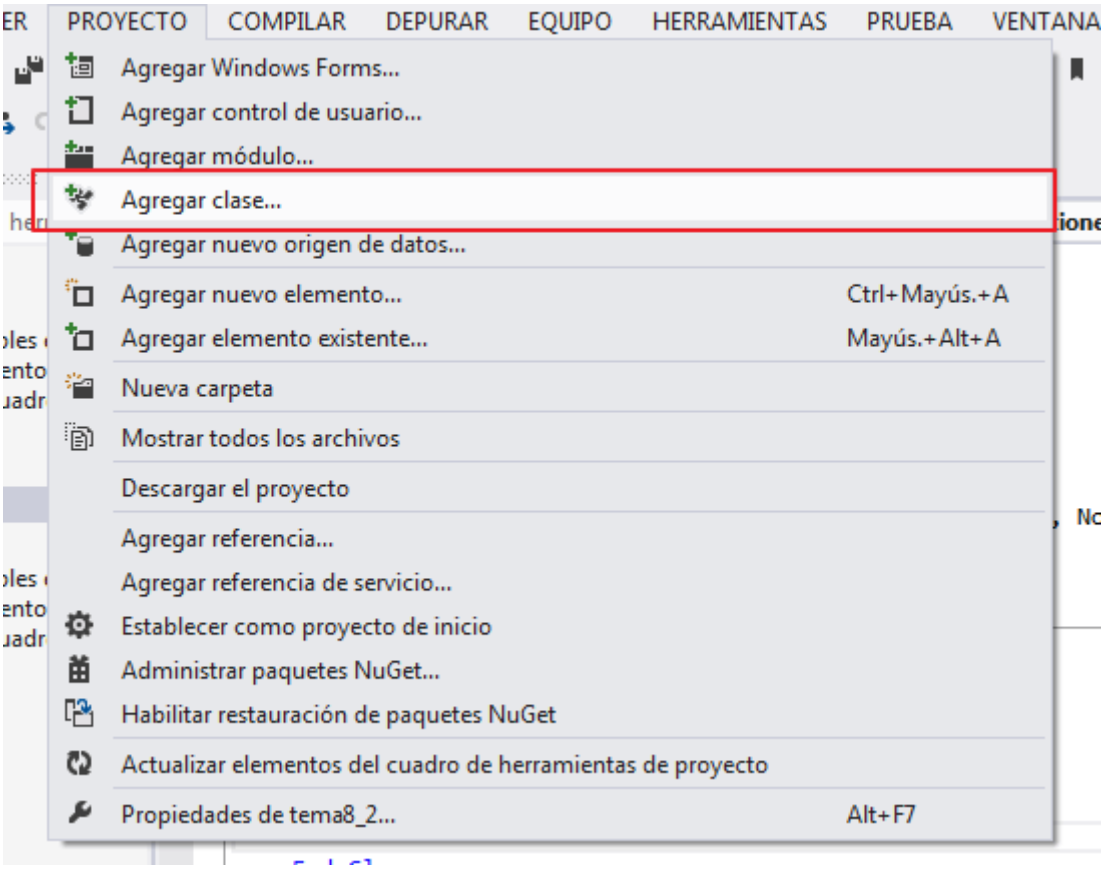
6. Ejemplo

Vamos a replantear un diseño, encauzándolo bajo una perspectiva orientada a objetos, que nos permita un uso más sencillo del código y un mantenimiento también más fácil. Para lo cual desarrollaremos una clase que contenga todas las operaciones a realizar por un empleado; en definitiva, crearemos la clase Empleado, cuyo proceso describiremos a continuación.

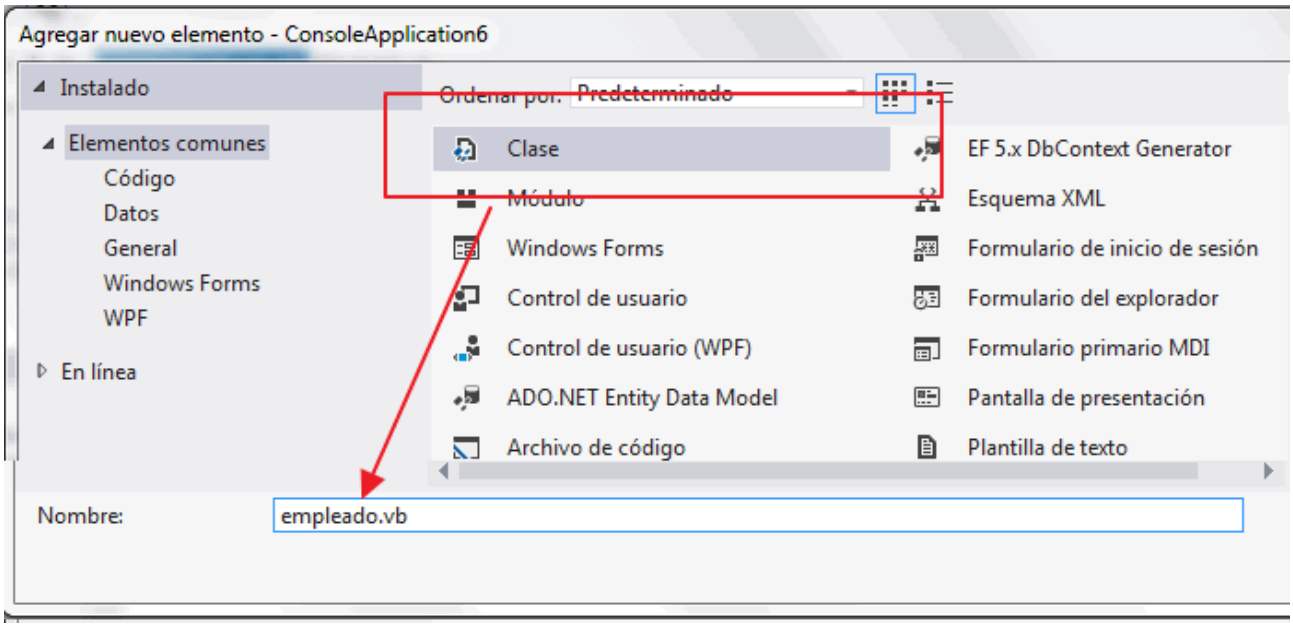
Podemos escribir una clase en VB.NET utilizando diferentes tipos de aplicación, en este caso emplearemos una aplicación de consola. Iniciaremos en primer lugar VS.NET, creando un proyecto de tipo consola.



A continuación seleccionaremos el menú Proyecto y luego Agregar clase:



Nos mostrará la ventana para agregar nuevos elementos al proyecto. El nombre por defecto asignado por el IDE para la clase será Class1; cambiaremos dicho nombre por Empleado, y pulsaremos Abrir.

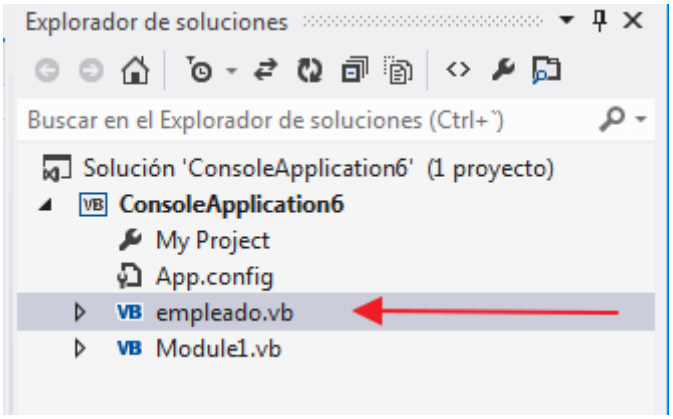


Se creará de esta forma un nuevo fichero de código (Empleado.vb), y nos mostrará el editor de código con su contenido. En el nuevo fichero del proyecto veremos ya en el código las palabras clave Class...End Class.

```
Public Class Empleado

End Class
```

Y en el explorador de soluciones tendremos por supuesto la clase con el nombre que le hemos indicado:



6.1. Organización de clases en uno o varios ficheros de código

Como hemos comentado antes una clase es uno de los tipos de contenedor lógico disponible dentro del entorno de .NET Framework. Su organización puede realizarse al igual que los módulos (Module) de código estándar del entorno, es decir, cada vez que añadimos una clase al proyecto utilizando el IDE, se crea por defecto, un fichero de código por clase.

Sin embargo, también podemos incluir varias clases dentro del mismo fichero de código, o mezclar clases con módulos y otro tipo de contenedores lógicos, cambiando si es necesario el nombre del fichero. Este ejemplo muestra dos clases creadas dentro del mismo fichero.

```
' Fichero MisClases.VB
' ====================
Public Class Empleado
' código de la clase
' ......
' ......
End Class

Public Class Factura
' código de la clase
' ......
' ......
End Class
```

6.2. Código de clase y código cliente

Antes de comenzar a realizar pruebas con las clases que vayamos escribiendo, debemos explicar dos conceptos referentes a la escritura de código orientado a objeto, ya que en función del lugar desde el que sea manipulado, debemos distinguir entre código de clase y código cliente.

- **Código de clase**. Se trata del código que escribimos para crear nuestra clase, y que se encuentra entre las palabras clave `Class...End Class`.
- **Código cliente**. Se trata del código que hace uso de la clase mediante la acción de crear o instanciar objetos a partir de la misma. Aquí se englobaría todo el código que se encuentra fuera de la clase.

6.3. Reglas de ámbito generales para clases

Las normas de ámbito para variables y procedimientos escritos dentro de una clase son las mismas que las ya vistas, pero debemos tener en cuenta que en el caso actual, el ámbito a nivel de módulo debe ser equiparado con el ámbito a nivel de clase. Por ejemplo, cuando declaremos una variable dentro de las palabras clave `Class...End Class`, pero fuera de todo procedimiento de la clase, dicha variable tendrá un ámbito a nivel de clase, siendo accesible por los procedimientos de la clase o por todo el código, según la hayamos definido privada o pública.

6.4. Instanciación de objetos

En este momento, nuestra clase Empleado cuenta con el código mínimo para poder ser utilizada, para lo que debemos instanciar objetos a partir de la misma. Recuerda que el proceso de instanciación consiste en crear un objeto a partir de las especificaciones de la clase. El modo más común de trabajar con una instancia de una clase, o lo que es lo mismo, con un objeto, pasa por asignar dicho objeto a una variable.

Instanciaremos un objeto en el código utilizando la sintaxis de declaración de variables junto a la palabra clave `New`, empleando como tipo de dato el nombre de la clase. Todo este código lo podemos situar en un módulo dentro del proyecto, bien en un fichero de código aparte o en el mismo fichero en donde estamos escribiendo la clase. El ejemplo siguiente muestra las formas disponibles de instanciar un objeto y asignarlo a una variable.

```
Module General
    Sub Main()
      ' declarar primero la variable
     ' y después instanciar el objeto
      Dim empleado1 As Empleado
      empleado1 = New Empleado()
```

```
        ' declaración e instanciación simultánea
        Dim empleado2 As New Empleado()
        ' declaración y posterior instanciación en
        ' la misma línea de código
        Dim empleado3 As Empleado = New Empleado()
    End Sub
End Module
```

Si bien es cierto que ya es posible crear objetos a partir de nuestra clase, no lo es menos el hecho de que no podemos hacer grandes cosas con ellos, puesto que la clase se encuentra vacía de código. Debemos añadir propiedades y métodos para conseguir que los objetos actúen en nuestra aplicación.

6.5. Miembros de la clase

Los elementos de una clase que contienen sus datos y definen su comportamiento, es decir, las propiedades y métodos, reciben además el nombre de miembros de la clase, término que también utilizaremos a partir de ahora.

6.6. Definir la información de la clase

Existen dos formas de almacenar los datos o información en una clase: a través de campos de clase y de propiedades. Desde la perspectiva del programador que hace uso de una clase para crear objetos, la diferencia entre un campo o una propiedad resulta imperceptible; sin embargo, desde el punto de vista del programador de la clase existen claras diferencias, concernientes fundamentalmente, a preservar la encapsulación del código de la clase.

El uso de campos o propiedades para una clase es una cuestión de diseño, no pudiendo afirmar categóricamente que un tipo de almacenamiento de datos sea mejor que otro.

6.7. Creación de campos para la clase

Un campo de una clase no es otra cosa que una variable, generalmente con ámbito público, accesible desde el exterior de la clase. Este ejemplo muestra la creación de un campo para la clase Empleado.

```
Public Class Empleado
    ' declaramos un campo en la clase
    ' para guardar el identificador
    ' del empleado
  Public identificador As Integer
End Class
```

Para manipular un campo desde código cliente, debemos instanciar un objeto, a continuación de la variable que lo contiene situar un punto (.), y finalmente el nombre del campo a manipular. Este modo de operación es común para todos los miembros de clases, tanto creadas por el programador, como pertenecientes a la propia plataforma .NET Framework.

```
Module General
 Sub Main()
    Dim empleado1 As Empleado
    ' instanciar el objeto
    empleado1 = New Empleado()
    ' asignar un valor al campo del objeto
    empleado.identificador = 75
    ' mostrar el valor de un campo del objeto
    Console.WriteLine("El valor del campo es: {0}", _
    empleado1.identificador)
    Console.ReadLine()
 End Sub
End Module
```

Habrás comprobado cómo al escribir el nombre del objeto y el punto, aparece una lista con los miembros de la clase accesibles desde el código cliente. De momento sólo disponemos del campo y el método GetType( ), que devuelve un objeto de la clase Type, conteniendo información sobre el tipo del objeto. Esta lista irá aumentando progresivamente según añadimos más propiedades y métodos a la clase, constituyendo una inestimable ayuda para el programador, que le evita el tener que recordar los nombres de todos los elementos de la clase, o consultar continuamente su documentación.

6.8. Creación de propiedades para la clase

Una propiedad en la clase se define, por norma general, mediante dos elementos: una variable de propiedad y un procedimiento de propiedad. La variable de propiedad, tal y como su nombre indica, es una variable con ámbito privado a nivel de la clase, que se encarga de guardar el valor de la propiedad. Por su parte el procedimiento de propiedad o Property, es el encargado de actuar de puente entre el código cliente y la variable de propiedad, realizando las operaciones de acceso y asignación de valores a dicha variable.

Por lo tanto, para crear una propiedad en nuestra clase, declararemos en primer lugar una variable Private, y, en segundo lugar, un procedimiento de tipo Property, que consta de dos bloques: Get, para devolver el valor de la variable de propiedad; y Set, para asignárselo. La sintaxis a emplear se muestra es este ejemplo:

```
Public Class Empleado
    ' declarar una variable de propiedad para la propiedad Nombre
    Private msNombre As String
    ' declarar el procedimiento Property para la propiedad Nombre
    Public Property Nombre() As String
     ' bloque Get para devolver el valor de la propiedad
     Get
         Return msNombre
     End Get
     ' bloque Set para asignar valor a la propiedad
     Set(ByVal Value As String)
            msNombre = Value
     End Set
   End Property
End Class
```

Cuando declaramos un procedimiento Property, debemos, al igual que en una función, tipificarlo (decirle el tipo de datos), ya que una de sus labores consiste en la devolución de un valor.

Para devolver el valor, en el bloque Get podemos utilizar la palabra clave Return, seguida del valor de retorno, o bien la sintaxis clásica de asignar el valor al nombre de la función. Nuestra recomendación es el uso de Return por las ventajas explicadas en el tema del lenguaje.

En cuanto a la asignación de valor, el bloque Set utiliza un parámetro con el nombre Value, que contiene el valor para asignar a la propiedad. Habrás visto que al declarar un procedimiento de este tipo, el IDE de VS.NET crea automáticamente los correspondientes bloques Get y Set, ahorrándonos este trabajo.

A la hora de manipular una propiedad desde el código cliente, y como ya habíamos apuntado anteriormente, la diferencia no será notoria, como muestra el ejemplo siguiente. La única forma de hacer más patente el uso del procedimiento Property, consiste en ejecutar el programa utilizando el depurador; de esta manera comprobaremos como el flujo de la ejecución salta a los bloques Get y Set al manejar la variable del objeto.

```
Sub Main()
    Dim empleado1 As New Empleado()
   ' asignar valor a una propiedad
    empleado1.Nombre = "Guillermo"
    ' mostrar el valor de una propiedad del objeto
    Console.WriteLine("El valor de la propiedad Nombre es: {0}", _
    empleado1.Nombre)
    Console.ReadLine()
End Sub
```

Dado que los procedimientos Property no son otra cosa que rutinas de código, también se les denomina métodos de acceso y asignación en el argot POO.

6.9. Ventajas en el uso de propiedades

Comprobada la facilidad de los campos de clase, seguramente os estaréis preguntando en estos momentos por qué deben utilizar propiedades, si en definitiva, su finalidad es la misma que los campos: guardar un valor en el objeto.

Existen varias y poderosas razones, por las cuales nos debemos decantar en muchas ocasiones, hacia el uso de propiedades. En los siguientes apartados haremos una descripción de ellas.

6.10. Encapsulación a través de propiedades

Una de las características de la POO, la encapsulación, establece que el código de una clase debe permanecer, siempre que sea posible, protegido de modificaciones no controladas del exterior (código cliente). Nuestra clase debe actuar como una especie de caja negra, que expone un interfaz para su uso, pero que no debe permitir el acceso a la implementación de dicho interfaz.

Supongamos que en nuestra clase Empleado necesitamos crear un elemento para guardar el sueldo pagado, pero el importe del sueldo deberá estar entre un rango de valores en función de la categoría del empleado. Si la categoría es 1, el sueldo estará entre 1 y 200, mientras que si la categoría es 2, el sueldo podrá llegar hasta 300. Si abordamos este problema utilizando campos de clase, puede ocurrir lo que mostramos en este ejemplo:

```
Module General
Sub Main()
  Dim loEmpleado As Empleado
  loEmpleado = New Empleado()
   loEmpleado.psNombre = "Juan"
   loEmpleado.piCategoria = 1
  'atención, el sueldo para este empleado debería estar entre 1 a 200,
  ' debido a su categoría
   loEmpleado.mdbSueldo = 250
End Sub
End Module

Public Class Empleado
    Private msNombre As String
    Public miCategoria As Integer
    Public mdbSueldo As Double
End Class
```

¿Qué está sucediendo aquí?. Hemos creado un objeto empleado al que le hemos dado categoría 1, sin embargo, le estamos asignando un sueldo que no corresponde a su categoría, pero se nos permite hacerlo sin ningún problema, ya que no existe un medio de control que nos lo impida.

Afrontando el problema mediante el uso de propiedades, contamos con la ventaja de escribir código de validación en los correspondientes procedimientos Property; con ello encapsulamos el código de la clase, manteniéndolo a salvo de asignaciones incoherentes. Veamos esta solución:

```
Module General
   Sub Main()
      Dim loEmpleado As Empleado
      loEmpleado = New Empleado()
      loEmpleado.psNombre = "Pedro"
      loEmpleado.Categoria = 1
      loEmpleado.Sueldo = 250
      Console.WriteLine("Asignación incorrecta")
      Console.WriteLine("Empleado {0} - Categoria {1} - Sueldo {2}", _
      loEmpleado.psNombre, loEmpleado.Categoria, loEmpleado.Sueldo)
      loEmpleado.Sueldo = 175
      Console.WriteLine("Asignación correcta")
      Console.WriteLine("Empleado {0} - Categoria {1} - Sueldo {2}", _
      loEmpleado.psNombre, loEmpleado.Categoria, loEmpleado.Sueldo)
      Console.ReadLine()
   End Sub
End Module

Public Class Empleado
   Public psNombre As String
   ' variables de propiedad
   Private miCategoria As Integer
   Private mdbSueldo As Double
   ' procedimientos de propiedad
   Public Property Categoria() As Integer
     Get
        Return miCategoria
     End Get
     Set(ByVal Value As Integer)
        miCategoria = Value
     End Set
   End Property
   Public Property Sueldo() As Double
      Get
         Return mdbSueldo
      End Get
      ' cuando asignamos el valor a esta propiedad,
      ' ejecutamos código de validación en el bloque Set
      Set(ByVal Value As Double)
      ' si la categoría del empleado es 1...
         If miCategoria = 1 Then
            ' ...pero el sueldo supera 200
            If Value > 200 Then
               ' mostrar un mensaje y asignar un cero
               Console.WriteLine("La categoría no corresponde con el sueldo")
               mdbSueldo = 0
            Else
               ' si todo va bien, asignar el sueldo
                mdbSueldo = Value
            End If
         End If
      End Set
```

```
        End Property
    End Class
```

6.11. Propiedades de sólo lectura o sólo escritura

Se nos plantea ahora un nuevo caso para nuestra clase Empleado: debemos guardar el valor del código de cuenta bancaria del empleado en el objeto, pero sin permitir que dicha información sea accesible desde el código cliente.

Igualmente y en función de los primeros dígitos de la cuenta bancaria, necesitamos mostrar el nombre de la entidad, pero sin permitir al código cliente su modificación, ya que esta va a ser siempre una operación que debe calcular el código de la clase.

Utilizando campos de clase no es posible resolver esta situación, ya que al ser de ámbito público, permiten tanto la escritura como lectura de sus valores. Pero si empleamos propiedades, estas nos permiten la creación de las denominadas propiedades de sólo lectura o sólo escritura, en las que utilizando las palabras clave ReadOnly y WriteOnly, conseguimos que a una determinada propiedad, sólo podamos asignarle o recuperar su valor.

Debido a esto, en una propiedad ReadOnly no podremos escribir el bloque Set, ya que no tendría sentido, puesto que no se va a utilizar. Lo mismo podemos aplicar para una propiedad WriteOnly, sólo que en esta, el bloque que no podremos codificar será Get. Igualmente obtendremos un error del compilador, si en el código cliente intentamos asignar un valor a una propiedad ReadOnly, u obtener un valor de una propiedad WriteOnly.

Veamos a continuación, un ejemplo de cómo resolver el problema comentado al comienzo de este apartado.

```
Module General
  Sub Main()
    Dim loEmpleado As Empleado
    loEmpleado = New Empleado()
    loEmpleado.psNombre = "Pedro"
    ' a esta propiedad sólo podemos asignarle valor, si intentamos
    obtenerlo, se producirá un error
    loEmpleado.CuentaBancaria = "2222-56-7779995555"
    ' en esta línea, la propiedad EntidadBancaria sólo nos 'permite
    obtener valor,
    ' si intentamos asignarlo se producirá un error
    Console.WriteLine("La entidad del empleado {0} es {1}", _
    loEmpleado.psNombre, loEmpleado.EntidadBancaria)
    Console.ReadLine()
  End Sub
End Module

Public Class Empleado
    ' campo de clase
    Public psNombre As String
    ' variables de propiedad
    Private msCtaBancaria As String
```

```
Private msEntidad As String
' variables diversas
Private msCodigoEntidad As String
' esta propiedad sólo permite asignar valores, por lo que no
dispone de bloque Get
Public WriteOnly Property CuentaBancaria() As String
  Set(ByVal Value As String)
        Select Case Left(Value, 4)
         Case "1111"
                msEntidad = "Banco Universal"
         Case "2222"
                msEntidad = "Banco General"
         Case "3333"
                msEntidad = "Caja Metropolitana"
         Case Else
                msEntidad = "entidad sin catalogar"
        End Select
  End Set
End Property
' esta propiedad sólo permite obtener valores, por lo que no
dispone de bloque Set
Public ReadOnly Property EntidadBancaria() As String
        Get
            Return msEntidad
        End Get
End Property
End Class
```

6.12. Propiedades virtuales

Otra de las ventajas del uso de propiedades reside en la posibilidad de definir propiedades virtuales; es decir, una propiedad que no tenga una correspondencia directa con una variable de propiedad. Así, podremos crear un procedimiento Property que no esté obligatoriamente asociado con una variable.

Siguiendo con la clase Empleado, en esta ocasión creamos una propiedad para almacenar la fecha en la que el empleado ha sido incorporado a la empresa; esto no entraña ninguna novedad. Sin embargo, seguidamente necesitamos disponer de una propiedad que nos permita mostrar el nombre del mes en el que se ha dado de alta al empleado.

Podemos resolver esta cuestión creando una variable de propiedad, guardando en ella una cadena con el nombre del mes; pero si disponemos de la fecha de alta, que ya contiene el mes, nos ahorraremos ese trabajo extra creando una propiedad, en este caso de sólo lectura, en la que extraigamos el nombre del mes de la fecha de alta y lo devolvamos como resultado. Veamos cómo hacerlo.

```
Module General
```

```
    Sub Main()
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()
        loEmpleado.psNombre = "Antonio"
        loEmpleado.FechaAlta = "12/6/2018"
        ' mostramos el mes de alta, que corresponde a una propiedad virtual del objeto
        Console.WriteLine("El empleado {0} se ha dado de alta en el mes de {1}", _
        loEmpleado.psNombre, loEmpleado.MesAlta)
        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    ' campo de clase
    Public psNombre As String
    ' variables de propiedad
    Private mdtFechaAlta As Date
    ' propiedad para manejar la fecha de alta del empleado
    Public Property FechaAlta() As Date
        Get
            Return mdtFechaAlta
        End Get
        Set(ByVal Value As Date)
            mdtFechaAlta = Value
        End Set
    End Property
    ' propiedad virtual, en ella devolvemos el nombre del mes en el que se ha dado
    ' de alta al empleado, utilizando la variable de otra propiedad
    Public ReadOnly Property MesAlta() As String
        Get
            Return Format(mdtFechaAlta, "MMMM")
        End Get
    End Property
End Class
```

6.13. Nombres de propiedad más naturales

Cuando desde código cliente trabajamos con objetos, estos ofrecen habitualmente nombres de propiedades claros y sin notaciones.

En el caso de la clase Empleado tenemos un inconveniente a este respecto con el campo de clase correspondiente al nombre del empleado, ya que en él utilizamos convenciones de notación para facilitar el mantenimiento del código, pero por otra parte, estamos contribuyendo a dificultar la legibilidad de los miembros de la clase desde el código cliente.

Es cierto que podemos obviar las convenciones de notación en el código, pero esto, como ya comentamos en el tema sobre el lenguaje, puede hacer que la lectura del programa sea más complicada.

Como hemos comprobado también en los pasados ejemplos, si utilizamos propiedades, podemos mantener nuestras normas de notación en cuanto a las variables de la clase, sea cual sea su tipo, y ofrecer al código cliente, nombres más naturales a través de los procedimientos Property.

Por lo tanto, si en lugar de utilizar un campo de clase para el nombre del empleado, la convertimos en una propiedad, habremos ganado en claridad de cara al programador usuario de nuestra clase. Vamos a verlo.

```
Module General
    Sub Main()
        Dim loEmpleado As New Empleado()
        ' al utilizar un objeto desde código cliente siempre es más sencillo manipular la
        ' propiedad Nombre, que msNombre, en cuanto a claridad del código se refiere
        loEmpleado.Nombre = "Juan"
    End Sub
End Module

Public Class Empleado
    ' antes usábamos un campo de clase...
    'Public psNombre As String <---
    ' ...pero lo convertimos en una variable de propiedad...
    Private msNombre As String
    ' ...creando su procedimiento de propiedad correspondiente
    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property
End Class
```

6.14. Propiedades predeterminadas

Una **propiedad predeterminada o por defecto**, es aquella que nos permite su manipulación omitiendo el nombre. Para establecer una propiedad como predeterminada en una clase, la variable de propiedad asociada deberá ser un array, pudiendo crear sólo una propiedad predeterminada por clase.

Al declarar una propiedad por defecto, deberemos utilizar al comienzo de la sentencia de declaración, la palabra clave **`Default`**. Para asignar y obtener valores de este tipo de propiedad, tendremos que utilizar el índice del array que internamente gestiona la propiedad. Pongamos como ejemplo el hecho de que el trabajo desempeñado por el empleado le supone realizar viajes a diferentes ciudades; para llevar un control de los viajes realizados, crearemos una nueva propiedad, que además será predeterminada. Atento a este ejemplo:

```
Module General
   Sub Main()
      Dim loEmpleado As New Empleado()
      Dim liContador As Integer
      ' primero manejamos la propiedad predeterminada igual que una normal
      loEmpleado.Viajes(0) = "Valencia"
      ' aquí manipulamos la propiedad predeterminada sin indicar su nombre
      loEmpleado(1) = "Toledo"
      For liContador = 0 To 1
         Console.WriteLine("Visita: {0} - Ciudad: {1}", _
         liContador, loEmpleado(liContador))
      Next
      Console.ReadLine()
   End Sub
End Module

Public Class Empleado
   ' este es el array asociado a la propiedad predeterminada
   Private msViajes() As String
   ' declaración de la propiedad predeterminada
   Default Public Property Viajes(ByVal Indice As Integer) As String
      Get
         ' para devolver un valor, empleamos el número de índice pasado como parámetro
        Return msViajes(Indice)
      End Get
      Set(ByVal Value As String)
        ' para asignar un valor a la propiedad, comprobamos primero si está vacío
         ' comprobar si el array está vacío, al ser el array también un objeto,
        ' utilizamos el operador Is
       If msViajes Is Nothing Then
                ReDim msViajes(0)
       Else
           ' si el array ya contenía valores, añadir un nuevo elemento
           ReDim Preserve msViajes(UBound(msViajes) + 1)
       End If
         ' asignar el valor al array
       msViajes(Indice) = Value
      End Set
   End Property
End Class
```

El uso de propiedades predeterminadas proporciona una cierta comodidad a la hora de escribir el código, sin embargo, si nos acostumbramos a especificar en todo momento las propiedades en el código, ganaremos en legibilidad.

Ejercicios

Ejercicio 1

Crea una clase que se llame coche y que tenga las propiedades de marca (cadena), modelo (cadena), color (cadena), velocidad (entero) y tamaño del coche (real). Crea también los constructores de la misma. Además añade un método que muestre todas las propiedades de un coche, otro que permita acelerar el coche y otro que permita frenar.

Crea un programa de consola y crea un objeto de esa clase en el que pruebes la misma.

Ejercicio 2

Crea una clase "descapotable" que se herede de la clase anterior. Esta clase tendrá además una propiedad para almacenar el color de la capota.

Realiza varias llamadas a los métodos de las dos clases.

Ejercicio 3

Deberás crear una **aplicación de consola** que contenga lo siguiente:

- La **clase Alumno** con las siguientes propiedades:

- Nombre
- Edad. (número entero entre 0 y 99)
- Nota. (Número decimal entre 0 y 10 con dos cifras decimales)

Un alumno se podrá crear de las siguientes formas:

- Sin ningún parámetro
- Con los parámetros nombre y edad. En este caso se inicializará la nota a 0.

También deberá contener los siguientes métodos:

- Subir nota. Deberá solicitar en cuántos puntos (con decimales) se quiere subir la nota. La nota máxima será 10.
- Bajar nota. Deberá solicitar en cuántos puntos (con decimales) se quiere bajar la nota. La nota mínima será 0.

- Un **programa** que sirva para probar todas las propiedades, constructores y métodos de la clase creada.