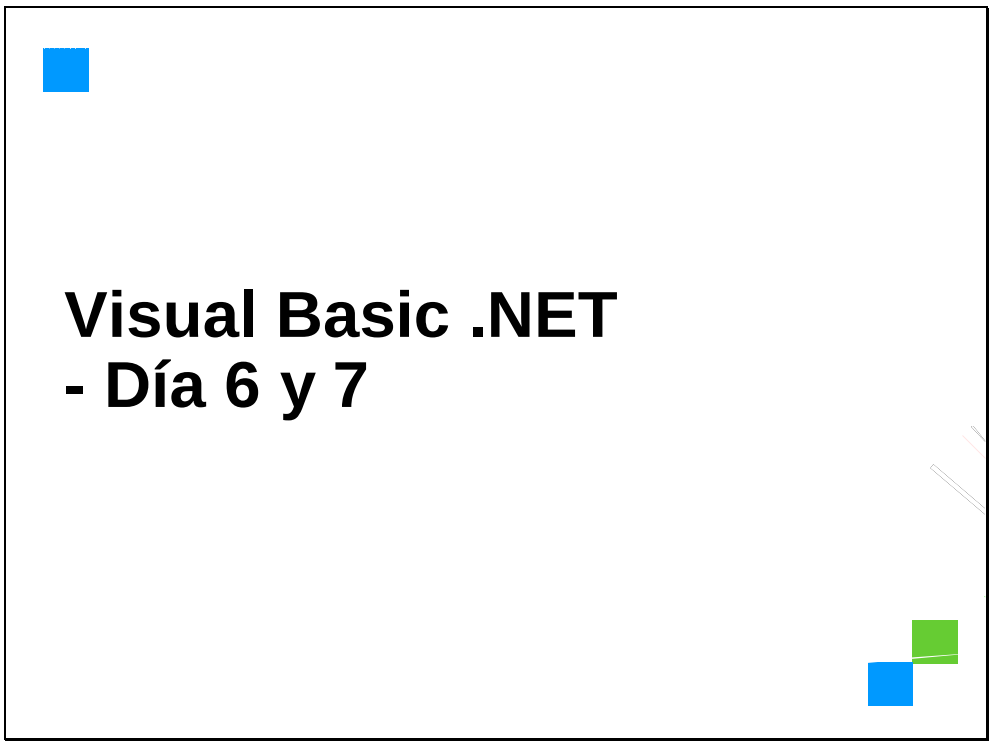
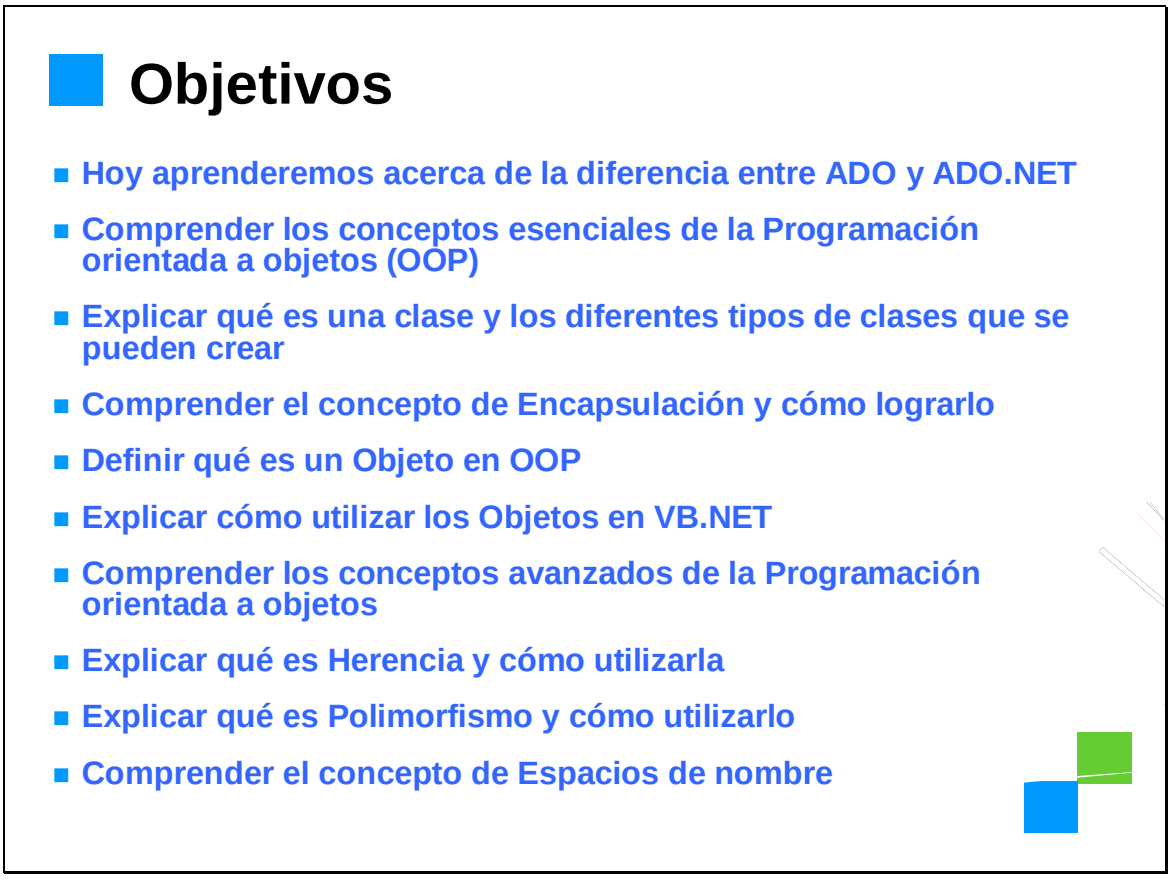
**Diapositiva 1**




**Diapositiva 2**



**Diapositiva 3**

## Programación orientada a objetos

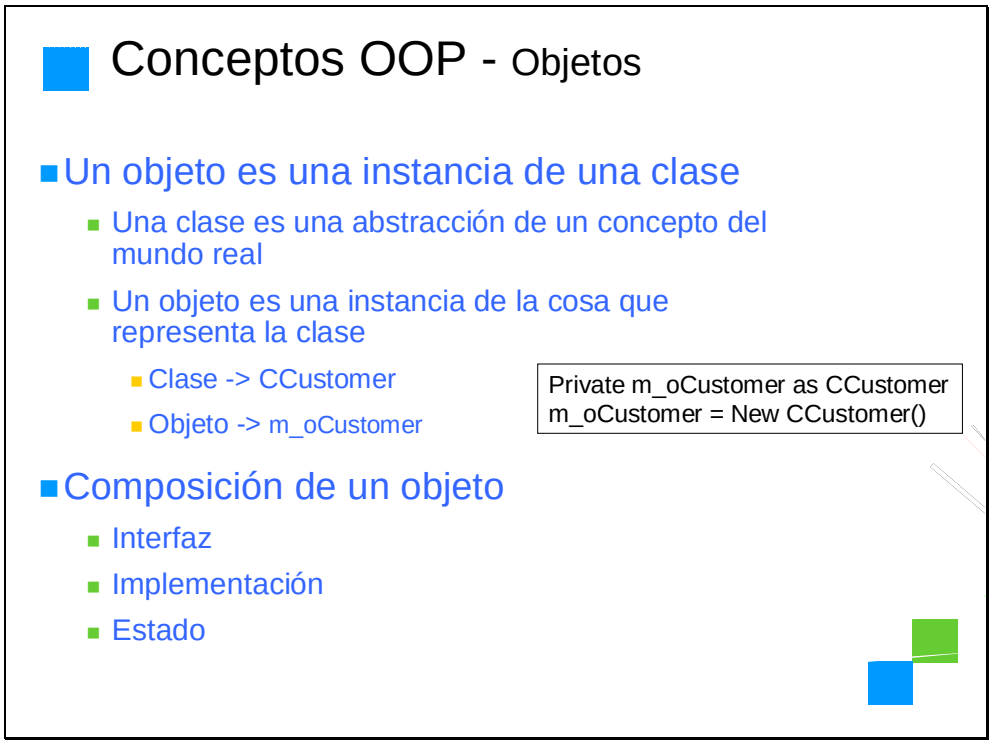
- Conceptos OOP
  - Objetos
  - Clases
  - Diseñar los objetos
  - Encapsulación
  - Ventajas de OOP
- Clases en VB.NET
  - Declarar clases
  - Miembros compartidos contra heredados
- Encapsulación en VB.NET
  - Declaración de métodos
  - Declaración de propiedades
  - Enfoques
- Objetos en VB.NET
  - Crear y destruir los objetos
  - Constructores
  - Terminación y limpieza

**Diapositiva 4**

## Programación orientada a objetos 2

- Conceptos OOP avanzados
  - Herencia
    - Derivar clases
    - Interfaces
    - Clases abstractas
  - Polimorfismo
    - Interfaces
    - Herencia
    - Clases abstractas
  - Espacios de nombre

## Diapositiva 5




Un objeto es la instancia de una clase. Una clase es la representación abstracta de un concepto en el mundo real, y proporciona la base a partir de la cual creamos instancias de objetos específicos.

Como ejemplo, puede crear una clase que defina a un cliente. Después puede crear una nueva instancia de la clase cliente para tener un objeto utilizable de Cliente:

Para poder crear un objeto de la clase cliente, debe crear una nueva instancia basada en esa clase. Por ejemplo:

```
Private m_oCustomer as CCustomer
m_oCustomer = New CCustomer()
```

Cada objeto es un elemento único de la clase en la que se basa. Si una clase es como un molde, entonces un objeto es lo que se crea a partir del molde. La clase es la definición de un elemento; el objeto **es** el elemento. El molde para una figura de cerámica en particular, es como una clase; la figura es el objeto.

Todos los objetos están compuestos de tres cosas:

**Interfaz**

La Interfaz es el conjunto de métodos, propiedades, eventos y atributos que se declaran como públicos en su alcance y que pueden invocar los programas escritos para usar nuestro objeto.

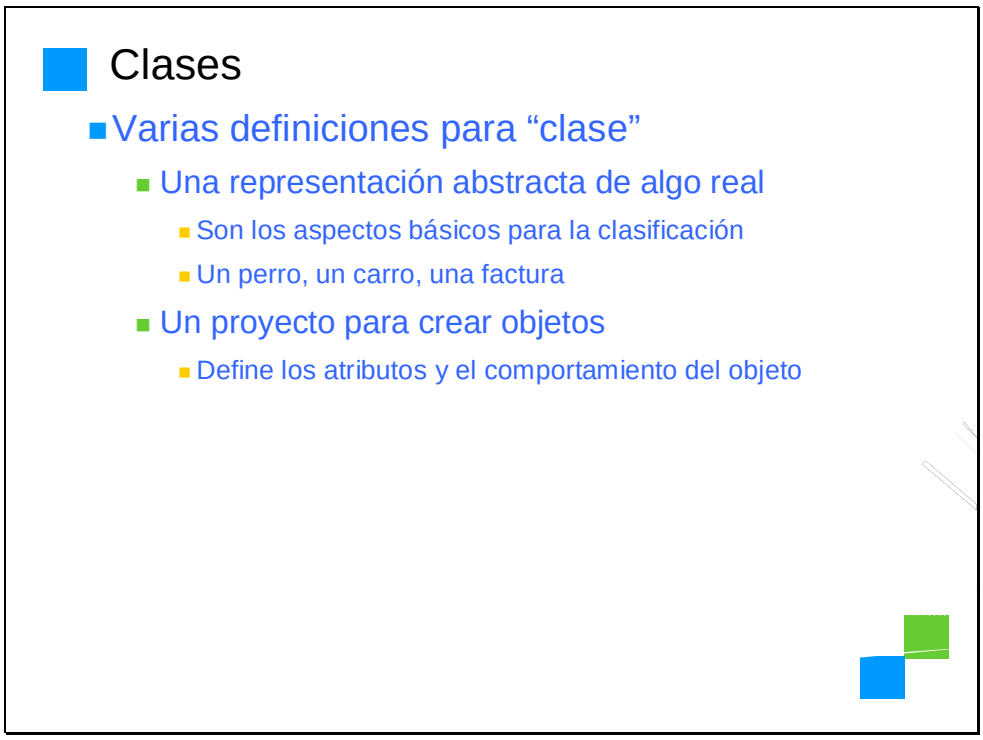
**Implementación**

Al código dentro de los métodos se le llama Implementación. Algunas veces también se le llama comportamiento, ya que este código es el que efectivamente logra que el objeto haga un trabajo útil. Es importante entender que las aplicaciones del cliente pueden utilizar nuestro objeto aunque cambiemos la implementación, siempre que no cambiemos la interfaz. Siempre que se mantengan sin cambio nuestro nombre de método, su lista de parámetro y el tipo de datos de devolución, podremos cambiar la implementación totalmente.

**Estado**

El estado o los datos de un objeto es lo que lo hace diferente de otros objetos de la misma clase. El estado se describe a través de las variables del Miembro o de la Instancia. Las variables del miembro son aquellas declaradas, de tal manera que están disponibles para todo el código dentro de la clase. Por lo general, las variables del miembro son Privadas en su alcance. Algunas veces, se les conoce como variables de la instancia o como atributos. Observe que las propiedades no son variables del Miembro, ya que son un tipo de método que funciona para recuperar y establecer valores.

# Diapositiva 6




Existen varias definiciones de **class** (clase). Una clase es un término que se utiliza no solamente en la programación.

**Una representación abstracta de algo real**

Los elementos básicos para una clasificación es una clase. Construir clases es un acto de clasificación, y es algo que todos hacen. Por ejemplo, aunque todo perro **real** es diferente al resto, existen comportamientos (todos ladran y caminan) y características (patas, color del pelo entre otros) similares. La gente utiliza el término **Perro** para clasificar a ese tipo de animales.

Podemos pensar en una clase llamada perro para crear la representación abstracta del tipo de animales.

**Un proyecto para crear objetos**

Esta definición es más técnica que la anterior.

Una **clase** es esencialmente un proyecto, a partir del cual puede crear objetos. Una clase define las características de un objeto, incluyendo las propiedades que definen los tipos de datos que ese objeto puede contener y los métodos que describen el comportamiento del objeto. Estas características determinan la manera en que otros objetos pueden acceder y trabajar con los datos que se incluyen en el objeto.

Por ejemplo, si desea construir objetos que representen perros, puede definir una clase **Perro** con ciertos comportamientos, como caminar, ladrar y comer, y propiedades específicas, como altura, peso y color. Una vez que haya definido la clase Perro, puede crear objetos con base en esa clase. Es importante observar que todos los objetos Perro creados con base en la clase perro compartirán los mismos comportamientos, pero tendrán sus propios valores específicos para el mismo conjunto de propiedades.

El siguiente ejemplo representa la definición de la clase Perro. Tome en consideración que ésta no es una sintaxis estricta de VB.NET; simplemente es un ejemplo de la definición de clase.

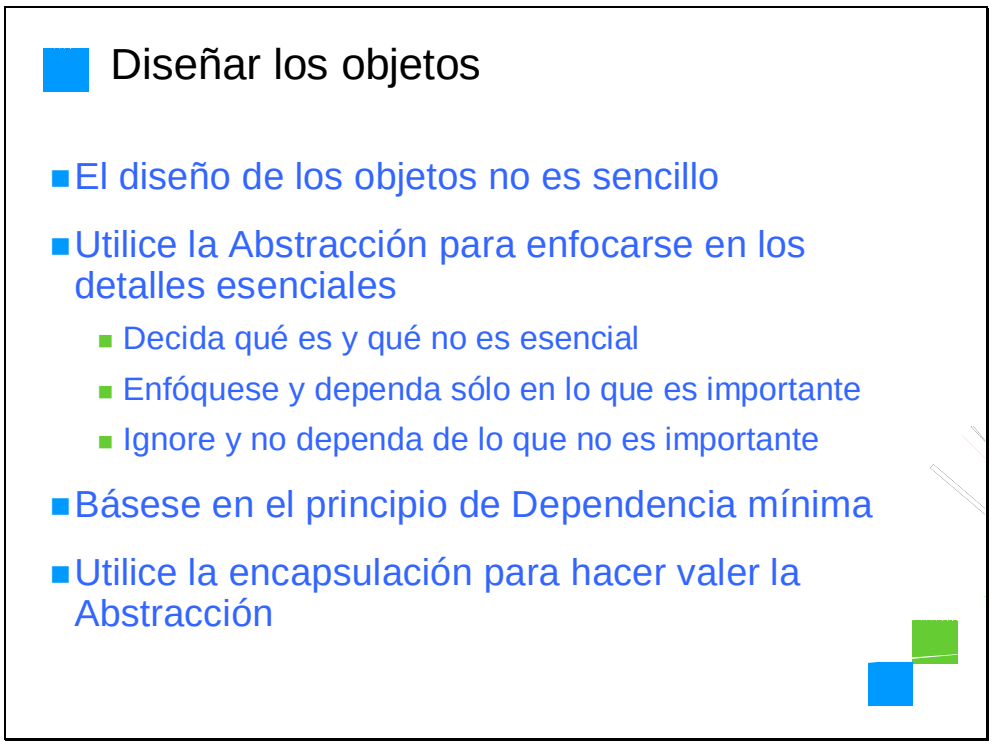
```
Public Class Perro
    Dim Altura As Decimal
    Dim Peso As Decimal
    Dim Color As String
    Sub Caminar(ByVal Pasos As Integer)

    End Sub
    Sub Ladrar()

    End Sub
    Sub Comer()

    End Sub
End Class
```

## Diapositiva 7




El diseño de objeto no es algo trivial. Debido a que las clases son representaciones abstractas de las entidades del mundo real, se puede ver abrumado con la gran cantidad de atributos y comportamientos que tienen estas entidades. Además, si este es el caso, el diseño de sus clases (y los objetos que utilizará más tarde) tendrá un gran conjunto de métodos y propiedades que nunca utilizará.
Para enfocarnos únicamente en los detalles importantes de las entidades que desea representar (los atributos y comportamientos que realmente utilizará), puede utilizar una Abstracción

**Abstracción** es la técnica de quitarle a una idea o a un objeto todos los acompañamientos innecesarios hasta que los deja en una forma esencial y mínima. Una buena abstracción elimina todos los detalles poco importantes y le permite enfocarse y concentrarse en los detalles importantes.
La abstracción es un principio de software importante. Una clase bien diseñada expone un conjunto mínimo de métodos cuidadosamente considerados que proporcionan el comportamiento esencial de una clase en una forma fácil de usar. Crear buenas abstracciones de software no es fácil. Encontrar buenas abstracciones generalmente requiere de un entendimiento muy claro del problema y de su contexto, gran claridad de pensamiento y amplia experiencia.

**Dependencia mínima**

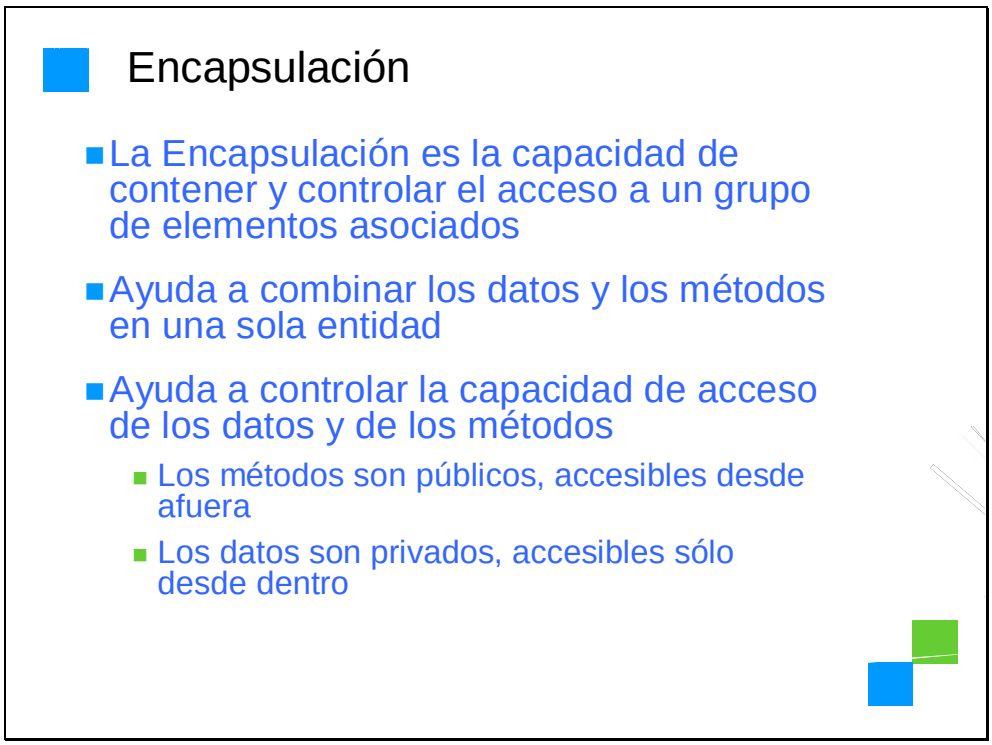
Las mejores abstracciones de software hacen que las cosas complejas sean simples. Logran esto al ocultar por completo los aspectos no esenciales de una clase. Estos aspectos no esenciales, una vez que han sido debidamente ocultados, no se pueden ver, ni usar, ni depender de ellos.

Este principio de dependencia mínima es lo que hace que la abstracción sea tan importante. El cambio es normal en el desarrollo de software. Lo mejor que puede hacer es minimizar el impacto de un cambio cuando éste sucede. Y cuanto menos dependa de algo, menos se verá afectado cuando cambie.

**Encapsulación**

Los lenguajes orientados a objetos proporcionan la Encapsulación. La encapsulación se puede utilizar para aplicar el concepto de Abstracción.

## Diapositiva 8




**Encapsulación**

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 para encapsular elementos. En el siguiente ejemplo, la clase BankAccount encapsula los métodos, campos y propiedades que se describen en una cuenta bancaria. Sin una encapsulación, usted necesitaría declarar procedimientos y variables por separado para almacenar y administrar información para la cuenta bancaria, y sería difícil trabajar con más de una cuenta bancaria a la vez. La encapsulación le permite utilizar los datos y procedimientos en la clase BankAccount como una unidad. Usted puede trabajar con varias cuentas bancarias al mismo tiempo sin confusión, debido a que cada cuenta está representada por una instancia única de la clase.

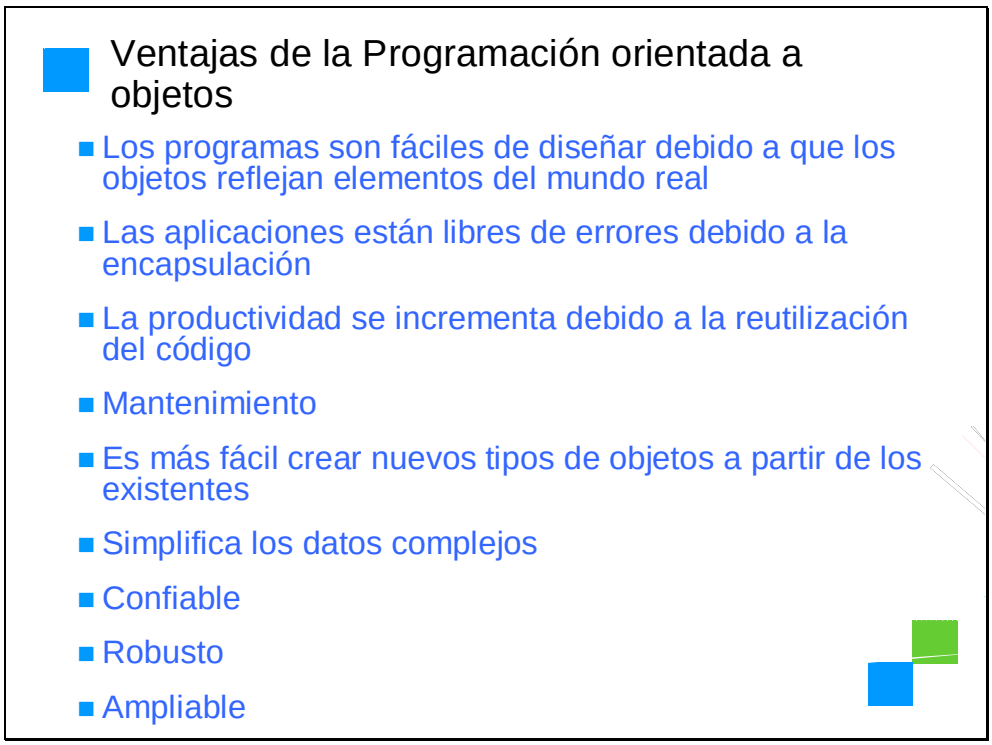
La encapsulación también le permite controlar la forma en que se utilizan los datos y los procedimientos. Puede utilizar modificadores de acceso, como **Private** o **Protected**, para evitar que los procedimientos externos ejecuten métodos de clase o lean y modifiquen datos en propiedades y campos. Usted debe declarar los detalles internos de una clase como **Private** para evitar que sean utilizados fuera de su clase; a esta técnica se le llama *ocultamiento de datos*. En la clase BankAccount, la información del cliente, como el saldo de la cuenta, se protege de esta forma. Una de las reglas básicas de la encapsulación es que los datos de la clase sólo se pueden modificar o recuperar a través de los procedimientos o métodos **Property**. Ocultar

los detalles de implementación de sus clases evita que se usen de maneras no deseadas, y le permite modificar esos elementos posteriormente sin riesgo de tener problemas de compatibilidad. Por ejemplo, versiones posteriores de la clase BankAccount enlistadas más adelante, podrían cambiar el tipo de datos del campo AccountBalance sin peligro de dañar la aplicación que depende de que este campo tenga un tipo de dato específico.

```
Class BankAccount
   Private AccountNumber As String
   Private AccountBalance As Decimal
   Private HoldOnAccount As Boolean = False

   Public Sub PostInterest()
      ' Add code to calculate the interest for this account.
   End Sub
   ReadOnly Property Balance() As Decimal
      Get
         Return AccountBalance 'Returns the available balance.
      End Get
   End Property
End Class
        Class CheckingAccount Inherits BankAccount
   Sub ProcessCheck() ' Add code to process a check drawn on this account.
   End Sub
End Class
```

## Diapositiva 9




Hacia mediados de los 80, los beneficios de la programación orientada a objetos empezaron a obtener reconocimiento, y el diseño de objetos pareció ser un enfoque sensato para la gente que deseaba utilizar el lenguaje de programación orientada a objetos.

Un enfoque orientado a objetos para programar ofrece muchos beneficios sobre un enfoque estructurado.

El análisis orientado a objetos y su diseño se enfoca en los objetos. Los objetos tienen ciertos comportamientos y atributos que determinan la manera en que interactúan y funcionan. No se intenta proporcionar un orden para las acciones al momento del diseño debido a que los objetos funcionan basados en la manera en que funcionan otros objetos.

La programación orientada a objetos ayuda a los desarrolladores a crear objetos que reflejan escenarios del mundo real.

Las implementaciones orientadas a objetos ocultan datos, lo cual significa que muestran únicamente los comportamientos a los usuarios y ocultan el código subyacente de un objeto. Los comportamientos que el programador expone son únicamente aquellos elementos que el usuario de un objeto puede afectar.

El enfoque orientado a objetos permite que los objetos estén autocontenidos. Los objetos existen por sí mismos, con una funcionalidad para invocar comportamientos de otros objetos. Al utilizar un enfoque orientado a objetos, los desarrolladores

pueden crear aplicaciones que reflejan objetos del mundo real, como rectángulos, elipses y triángulos, además de dinero, números de partes y elementos en un inventario.

En un enfoque orientado a objetos, los objetos, por definición, son modulares en su construcción. Esto quiere decir que son entidades completas y, por lo tanto, tienden a ser altamente reutilizables.

Las aplicaciones orientadas a objetos se construyen sobre el paradigma de los mensajes o de los eventos en donde los objetos envían mensajes a otros objetos, como el sistema operativo Microsoft® Windows®.

**Resumen de los beneficios**

En resumen, la programación orientada a objetos beneficia a los desarrolladores debido a que:

Los programas son fáciles de diseñar debido a que los objetos reflejan elementos del mundo real

Las aplicaciones son más sencillas para los usuarios debido a que los datos innecesarios están ocultos.

Los objetos son unidades autocontenidas

La productividad se incrementa debido a que puede reutilizar el código

Los sistemas son fáciles de mantener y se adaptan a las cambiantes necesidades de negocios

Es más fácil crear nuevos tipos de objetos a partir de los ya existentes

Simplifica los datos complejos

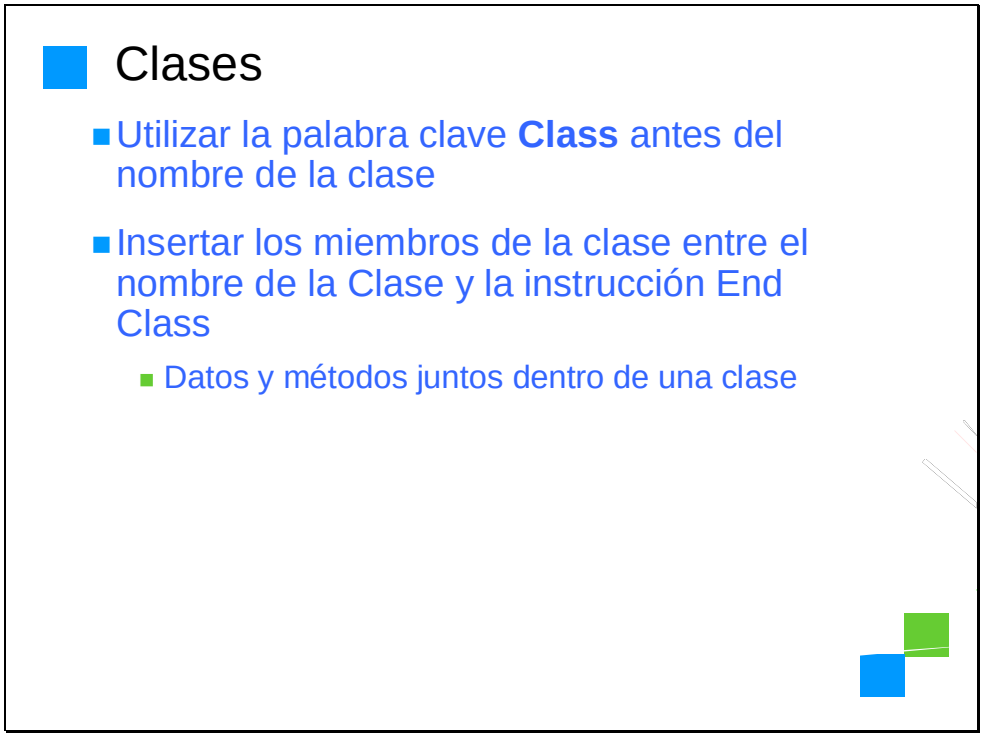
Reduce la complejidad de la transacción

Confiabilidad

Robustez

Capacidad de ampliación

## Diapositiva 10




Para definir una clase, se coloca la palabra clave **Class** antes del nombre de su clase, y después se insertan los miembros de la clase (datos y métodos) entre la definición del nombre de la clase y la instrucción **End Class**:

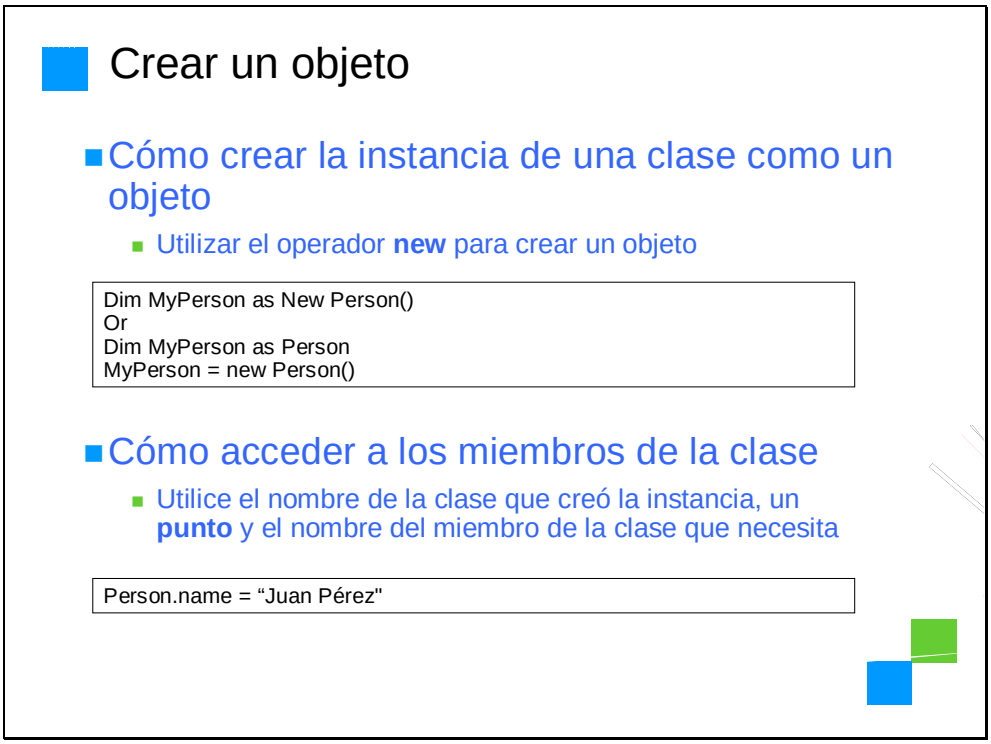
Si incluye los métodos, entonces el código de cada método también se debe incluir entre la declaración del método y el final del mismo.

El siguiente ejemplo define una nueva clase, **Persona**, con dos partes de información relevante asociadas: el nombre de la persona, su fecha de nacimiento y un Método que calcula la edad de la persona. A pesar de que la clase **Persona** se define en el ejemplo, no existe aún el objeto **Persona**. Deberán ser creados.

```
Public Class Persona
   Private mstrNombre As String
   Private mdtFechaNacimiento As Date
         ….
         ….
   Public Function Edad() As Integer
      Return DateDiff(DateInterval.Year, mdtFechaNacimiento, Now())
   End Function
End Class
```

Una clase es un tipo definido por el usuario en contraposición a un tipo proporcionado por el sistema. Al definir una clase, en realidad crea un nuevo tipo en su aplicación.

# Diapositiva 11




Para utilizar una clase que haya definido, deberá primero crear la instancia de un objeto de ese tipo utilizando la palabra clave new.

Sintaxis

```
<object_variable> = new <class>
MiPersona = new Persona()
```

También puede crear la instancia de un objeto cuando declara la variable. Por ejemplo:

```
Dim MiPersona as New Persona()
```

**Acceder a las variables de clase**

Una vez que ha creado la instancia de un objeto para acceder y utilizar los datos que contiene ese objeto, deberá escribir el nombre de la clase creada en la instancia, seguida por un punto y el nombre del miembro de la clase que desea acceder.

Por ejemplo, puede acceder al miembro Nombre de la clase Persona y asignarle un valor en su objeto MiPersona, el nombre de la clase creada en la instancia, como sigue:

MiPersona.Nombre = "Juan Pérez"

El siguiente código define una nueva clase llamada **Perro** con un miembro de clase llamado **Peso**, y crea una instancia de la clase Perro, un objeto llamado **Mascota**. El valor se asigna al miembro Peso en la clase Mascota.

```
Public Class Perro
    Public Peso As Integer
End Class
. . .
Dim Mascota As New Perro
Mascota.Peso = 100
```

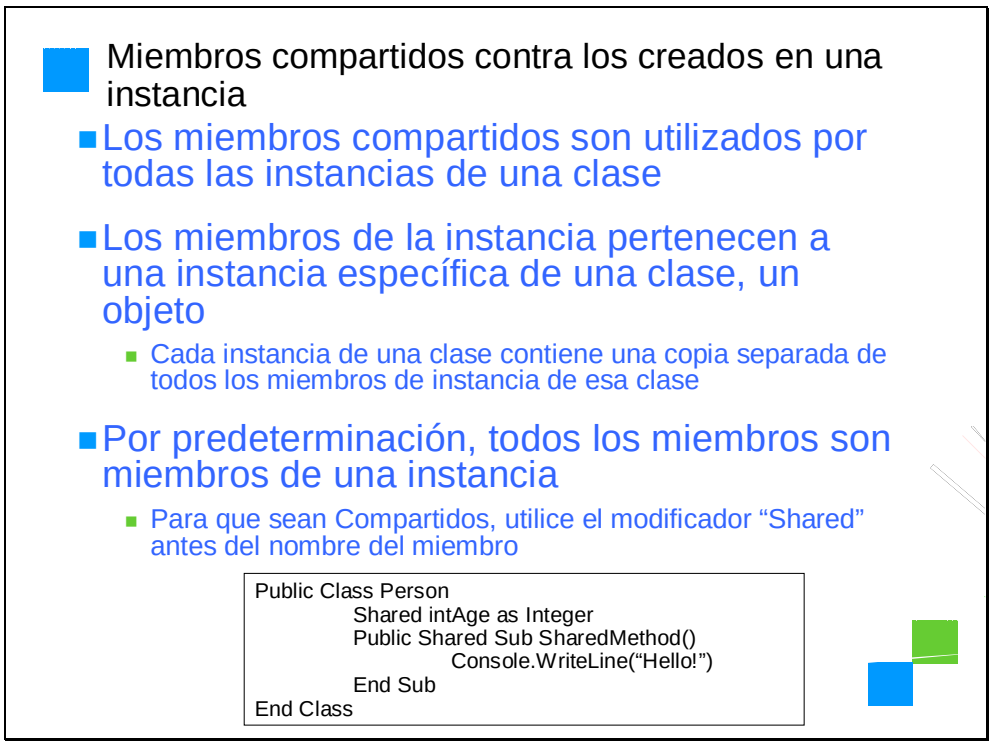
Las clases son tipos de referencia

Cada objeto **Persona** creado es un objeto separado, como aparece en el siguiente código:

```
    Dim miPersona1 As New Persona
    Dim miPersona2 As New Persona
    miPersona1.Nombre = "Juan Pérez"
```

El código anterior no cambia al miembro Nombre del objeto miPersons2. Ese valor sigue siendo nulo, que es el valor predeterminado para la cadena.

## Diapositiva 12




VB.NET nos permite crear variables y métodos que pertenecen a **class**, en lugar de a cualquier **object** específico. Otra forma de decir esto, es que estos valores y métodos pertenecen a todos los objetos de una clase dada y se comparten entre todas las instancias de la clase.

Podemos utilizar la palabra clave Shared para indicar las variables y los métodos que pertenecen a la clase, en lugar de a objetos específicos. Por ejemplo, nos puede interesar conocer el número total de objetos **Persona** creados al ejecutar nuestra aplicación, una especie de contador estadístico.

Ya que las variables regulares son únicas para cada objeto individual Persona, no nos permiten dar un fácil seguimiento al número total de objetos Persona creados. Sin embargo, si tuviéramos una variable que contara con un valor común en todas las instancias de la clase Persona, ese es el que utilizaríamos como contador.

```
Public Class Persona
   Private Shared sintContador As Integer
                  …
                  …
   Public Sub New()
      sintContador += 1
   End Sub
End Class
```

Consideraciones de diseño:

**Muchos miembros compartidos** – Ya que los miembros estáticos con frecuencia dan seguimiento a la información sobre grupos de instancias, usted puede estar tentado a agregar miembros que representen una abstracción grupal de la clase. En general, no complique el diseño con clases innecesarias, pero asegúrese de que cada clase represente una abstracción.

**Cuando las propiedades compartidas se convierten en datos globales** – En general, utilizar datos globales es una práctica deficiente de programación. Cuando hace globales a los datos, pierde el control de los mismos. Los datos globales pueden pasarse a cualquier método y luego cambiarse en formas inesperadas. Observe cuidadosamente su diseño si cuenta con una cantidad importante de datos públicos compartidos.

**Aplicaciones con subprocesos múltiples** – Si está trabajando con una aplicación con subprocesos múltiples, debe tomar en cuenta la sincronización. Suponga que tiene una clase con un arreglo compartido de integrales. Puede tener dos instancias de la clase, en subprocesos diferentes, que modifican el arreglo. Una instancia debe poder completar sólo parte de sus modificaciones antes de que la segunda empiece a modificar los datos, lo cual provoca resultados inesperados.

**Ejemplos**

**Campo Shared**

```
Shared m_Count as Integer
```

**Propiedad Shared**

```
Public Shared Property Number() As Integer
        Get
                Return m_Count
        End Get
        Set(ByVal  Value As Integer)
                m_Count = Value
        End Set
End Property
```

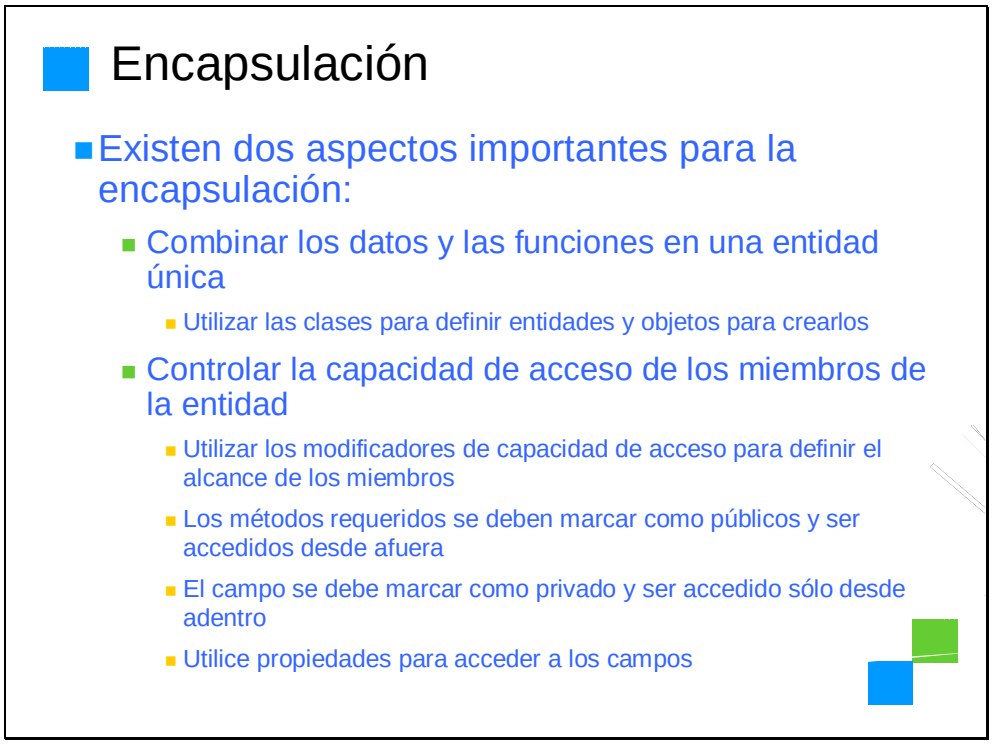
**Métodod Shared**

```
Public Shared Sub MyMethod()
End Sub
```

**Constructor compartido**

```
Shared Sub New()
End Sub
```

# Diapositiva 13




Existen dos aspectos importantes para la encapsulación:

- Combinar los datos y las funciones en una entidad única
- Controlar la capacidad de acceso de los miembros de la entidad

**Combinar los datos y las funciones en una entidad única**

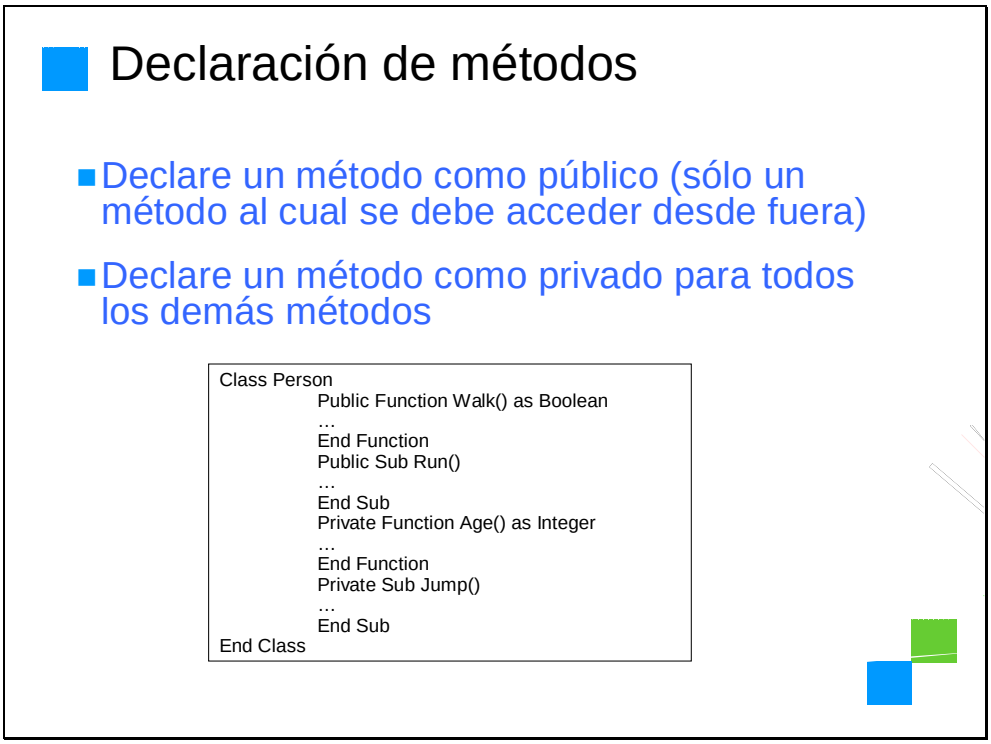
Como vimos antes, las clases proporcionan la abstracción necesaria para crear entidades, y los objetos se pueden utilizar para crear instancias únicas.

**Controlar la capacidad de acceso de los miembros de la entidad**

Una vez que se combinan los datos y las funciones en una entidad única, la entidad misma forma un límite cerrado, creando naturalmente una parte interna y una externa. Puede usar este límite para controlar selectivamente la capacidad de acceso de las entidades: algunas se podrán acceder solamente desde el interior; otras tanto desde el interior como desde el exterior. Los miembros que siempre son accesibles son **públicos**, y a los que sólo se puede acceder desde el interior son **privados**. A esto se le conoce como alcance de los miembros.

VB.NET, al igual que muchos otros lenguajes de programación orientados a objetos, ofrece completa libertad al elegir si los miembros deben ser accesibles o no. Puede, si lo desea, crear datos públicos. Sin embargo, se recomienda que los datos siempre estén marcados como privados y únicamente los métodos requeridos estén marcados como públicos. Si necesita proporcionar acceso a los campos, puede utilizar las propiedades, como lo veremos más adelante.

# Diapositiva 14




Al diseñar una clase debe considerar la manera en que los usuarios de esa clase accederán a sus miembros. Cuantos más miembros públicos ofrezca, más capacidad de acceso proporcionará.
Esto es válido para los métodos y para los campos.

Al declarar un método como **public**, el método estará accesible desde el exterior. Únicamente los métodos que desee tener accesibles desde el exterior deben marcarse como públicos.

Cuando declara un método como **private**, el método no estará accesible desde el exterior. Únicamente se puede acceder desde el interior de la clase, tanto para los métodos públicos como privados. La mayoría de los métodos en una clase se deben marcar como privados.

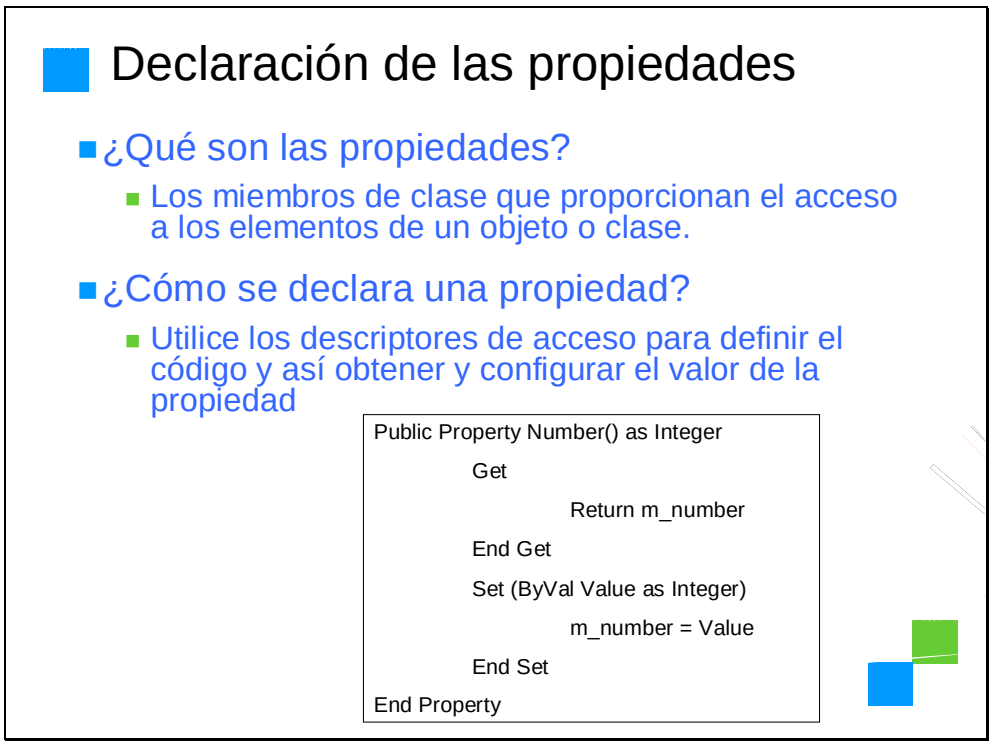
Una gran cantidad de diseño está relacionada a la decisión de colocar una función en el interior o en el exterior. Cuantas más funciones coloque en el interior (y mantenga su capacidad de uso), mejor.

**Ejemplo**

En el siguiente ejemplo, tanto el método **Caminar** como **Correr** serán accesibles desde el exterior. Cada objeto que pueda acceder a un objeto del tipo **Persona**, podrá invocar Caminar y Correr. Por otro lado, los métodos **Edad** y **Saltar** no se podrán acceder desde el exterior. El método Edad sólo será accesible desde los métodos Caminar, Correr y Saltar.

```
Class Persona
   Public Function Caminar() As Boolean
         …
   End Function
   Public Sub Correr()
         …
   End Sub
   Private Function Edad() As Integer
         …
   End Function
   Private Sub Saltar()
         …
   End Sub
End Class
```

## Diapositiva 15




La capacidad de acceso también se puede definir por campos (no sólo por métodos). A pesar de que puede controlar el acceso a los campos de clase utilizando los modificadores de acceso, una forma más poderosa de administrar el acceso es a través del uso de **propiedades**. Al utilizar las propiedades, puede administrar el acceso que otros objetos tienen a los datos en su clase.

**Properties** son miembros de clase que proporcionan acceso a los elementos de un objeto o clase.

**Properties** es una extensión de los campos y se accede a ellas utilizando la misma sintaxis. A diferencia de los campos, las propiedades no designan ubicaciones de almacenamiento. En lugar de eso, las propiedades tienen elementos de acceso que leen, escriben o calculan sus valores.

### Sintaxis

La sintaxis para definir una propiedad consiste en un modificador de acceso tal como **public** o **protected**, seguido por el tipo, el nombre de la propiedad, las palabras clave **get** y **set**, y el código de la propiedad para cada uno entre corchetes, como aparece en el siguiente código:

```
Public Property Numero() As Integer
    Get
        Return m_numero
    End Get
    Set(ByVal Value As Integer)
        m_numero = Value
    End Set
End Property
```

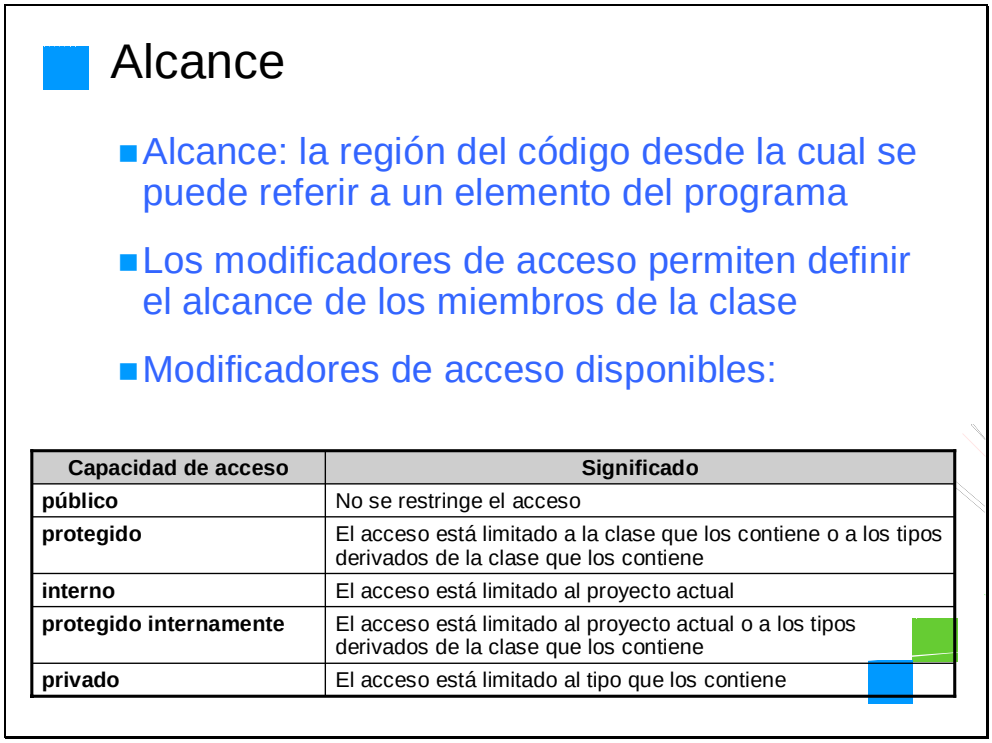
Las instrucciones **get** y **set** se llaman **accessors (**accesores**)**.

El elemento de acceso **get** puede devolver un tipo, que es el mismo que el tipo de propiedad, o uno que puede estar implícitamente convertido al tipo de propiedad.

El elemento de acceso **set** es equivalente a un método que tiene un parámetro implícito llamado **value**.

Al utilizar las propiedades, usted aplica la encapsulación. No se podrá acceder a campo alguno desde el exterior, y tendrá la capacidad de validar el acceso a los campos al agregar su propio código personalizado dentro de los elementos de acceso set y get.

# Diapositiva 16




**Definición del alcance**

El alcance se refiere a la región del código desde la cual se puede referenciar un elemento del programa. En el siguiente ejemplo, el miembro **Peso** de la clase **Perro** se puede acceder únicamente desde el interior de la clase **Perro**. Por lo tanto, el **alcance** del miembro **Peso** es la clase **Perro**.

Al utilizar los modificadores de acceso, puede definir el alcance de los miembros de la clase en sus aplicaciones. Es importante entender la manera en que los modificadores de acceso funcionan debido a que afectan su capacidad para utilizar una clase y sus miembros.

Cuando se permite el acceso a un miembro, se dice que es accesible. De otra manera, es inaccesible.

Utilice los modificadores de acceso, **public**, **protected**, **internal** o **private** para especificar una de las capacidades de acceso que se muestran en la diapositiva para los diferentes miembros.

Únicamente se permite un modificador de acceso para un miembro o tipo, a excepción de la combinación **protected internal**.

Los modificadores de acceso no están permitidos en los espacios de nombre. Los espacios de nombre no tienen restricciones de acceso (veremos espacios de nombre más adelante en este capítulo).

**Reglas**

Se aplican las siguientes reglas:

Los espacios de nombre siempre son públicos (implícitamente).

Las clases siempre son públicas (implícitamente).

Los miembros de clase son privadas, por predeterminación.

Sólo se puede declarar un modificador en un miembro de clase. Aunque *protected internal* son dos palabras, es sólo un modificador de acceso.

El alcance de un miembro nunca es mayor al del tipo que lo contiene.

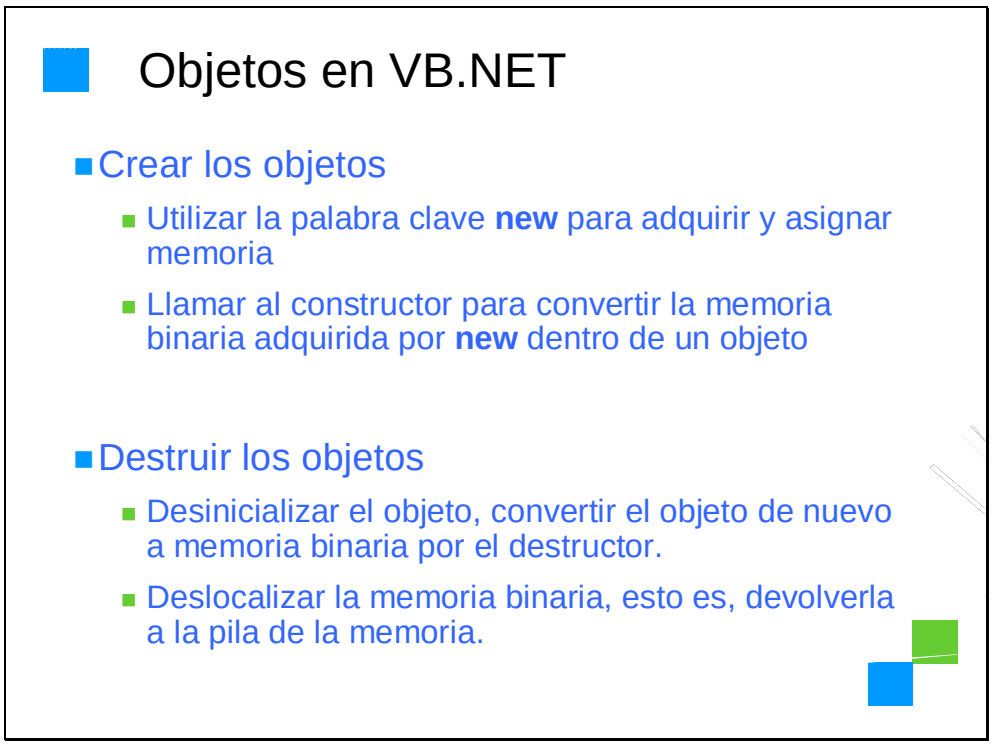
**Recomendaciones.**

La capacidad de acceso de los miembros de su clase determina el conjunto de comportamientos que ve el usuario de su clase. Si define un miembro de clase como privado, el usuario de esa clase no puede ver o utilizar ese miembro.

Debe hacer públicos únicamente aquellos objetos que los usuarios de su clase necesiten ver. Al limitar el conjunto de acciones que su clase hace pública, reduce la complejidad de su clase desde el punto de vista del usuario, y es más fácil para usted documentar y mantener su clase.

```
Class ClasePrincipal
    Public Class Perro
        Public Edad As Integer
        Private Peso As Integer
    End Class
    Shared Sub Main(ByVal args() As String)
        Dim miPerro As New Perro
        miPerro.Edad = 3
        ' the following line causes a compilation error
        miPerro.Peso = 75
    End Sub
End Class
```

## Diapositiva 17




**Crear objetos**

Crear un objeto VB.NET para un tipo de referencia es un proceso de dos pasos, como se muestra a continuación:

Utilice la palabra clave **new** para adquirir y asignar memoria

Llame al constructor para convertir la memoria cruda adquirida por **new** en un objeto

A pesar de que hay dos pasos en este proceso, deberá realizar ambos en una expresión. Por ejemplo, si **Perro** es el nombre de la clase, utilice la siguiente sintaxis para asignar memoria e inicializar el objeto **miPerro**:

```
Dim miPerro As new Perro( )
```

**Paso 1: Asignar memoria**

El primer paso en crear un objeto es asignar memoria para el objeto. Todos los objetos se crean utilizando el operador **new**. No hay excepción a esta regla. Puede hacerlo de manera explícita en su código, o el compilador lo hará por usted.

**Paso 2: Inicializar el objeto utilizando un constructor**

El segundo paso al crear un objeto es llamar a un constructor. Un constructor convierte la memoria asignada por **new** en un objeto. Existen dos tipos de constructores: constructores de instancias y constructores estáticos. Los constructores de instancias son constructores que inicializan los objetos. Los constructores estáticos son constructores que inicializan las clases.

**Destruir los objetos**

Destruir un objeto VB.NET también es un proceso de dos pasos:

Des-inicializar el objeto. Esto convierte nuevamente al objeto en una memoria cruda. En VB.NET esto se realiza mediante el destructor. Esto es lo inverso a la inicialización realizada por el constructor. Puede controlar lo que sucede en este paso al escribir su propio destructor o método de finalización.

La memoria cruda también se des-asigna; es decir, se devuelve a la pila de memoria. Esto es lo inverso de la asignación realizada por **new**.

No puede cambiar el comportamiento en este paso en forma alguna.

**Paso 1: Des-inicializar el objeto**

El tiempo de vida de un objeto no está vinculado con el alcance en el que se crea. Los objetos se inicializan en la pila de memoria asignada a través del operador **new**. Por ejemplo, en el siguiente código, la variable de referencia *eg* se declara dentro de la instrucción **for**.

Esto significa que *eg* sale del alcance final de la instrucción **for** y es una variable local.

No obstante, *eg* se inicializa con un objeto **new Example( )**, y este objeto no sale fuera del alcance con *eg*.

Recuerde, una variable de referencia y el objeto al que hace referencia son cosas diferentes.

```
Class Example
   Sub Method(int limit)
      Dim i As Integer
      For i = 0 To limit
         eg = New Example
                  ...
      Next i
      ' eg is out of scope
      ' Does eg still exist? No.
      ' Does the object still exist? Yes, it still exists in memory but you cannot reach it
(eg variable is out of scope at this point).
   End Sub
End Class
```

Esto significa que los objetos, por lo general, tienen las siguientes características:

**Destrucción no determinista**

Un objeto se crea cuando usted lo crea, pero a diferencia de un valor, no se destruye al final del alcance en el que se creó. La creación de un objeto es determinista, pero la destrucción de un objeto no lo es. No puede controlar con exactitud cuándo será destruido un objeto.

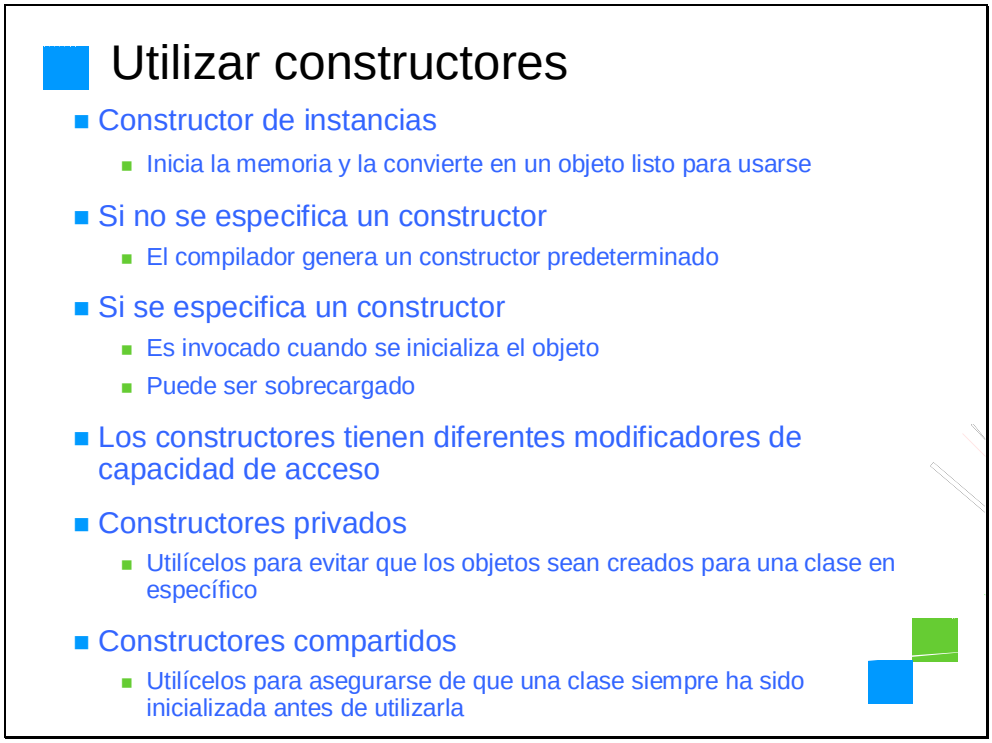
**Tiempos de vida más largos**

Debido a que la vida de un objeto no está vinculada con el método que genera, un objeto puede existir mucho más allá de una sola indicación del método . Pero, de nuevo, la variable de referencias utilizada para referenciar el objeto no lo hará. Cuando se destruye finalmente un objeto, se convierte de nuevo a memoria bruta. En VB.NET, no hay una manera de destruir explícitamente los objetos.

**Paso 2: Des-asignar memoria**

En este segundo paso, la memoria bruta se devuelve a la pila para ser reciclada. **El recolector de residuos** (garbage collector) automatiza completamente el segundo paso de este proceso por usted. La recolección de basura es un proceso automático no determinista que asegura que los objetos que no se alcancen sean destruidos, y que la memoria bruta del objeto esté libre al devolverla a la pila para ser reciclada.

## Diapositiva 18




**Cómo colaboran los constructores de new y de instancias**

Es importante observar qué tan de cerca colaboran los constructores **new** y de instancias para crear objetos. El único objeto de **new** es adquirir memoria bruta no inicializada. El único objeto de un constructor de instancias es inicializar la memoria y convertirla en un objeto preparado para usarse. En concreto, **new** no está involucrado con la inicialización en forma alguna y los constructores de instancias no están involucrados con la adquisición de memoria en forma alguna.

Al crear un objeto, el compilador VB.NET proporciona un constructor predeterminado si no desea escribirlo usted mismo. Considere el siguiente ejemplo:

```
Class Date
   Private yy As Integer
   Private mm As Integer
   Private dd As Integer
End Class
Class Test
   Shared Sub Main()
      Dim todayVar As New Date
      ...
   End Sub
End Class
```

La instrucción dentro de **Test.Main** crea un objeto **Date** llamado **todayVar** utilizando **new** (que asigna memoria desde la pila) e invocando un método especial que tiene el

mismo nombre que la clase (el constructor de instancias). Sin embargo, la clase **Date** no declara un constructor de instancias.

De manera predeterminada, el compilador genera automáticamente un constructor de instancias predeterminado.

**Características del constructor predeterminado**

Desde un punto de vista conceptual, el constructor de instancias que genera el compilador para la clase **Date**, se ve como el siguiente ejemplo:

```
Class Date
   Private yy As Integer
   Private mm As Integer
   Private dd As Integer
   Public Sub New()
      yy = 0
      mm = 0
      dd = 0
   End Sub
End Class
```

El constructor tiene las siguientes características:

**Su nombre SIEMPRE es New**

Este será el procedimiento que se utiliza cuando se llama, desde otra, la instrucción de creación de una nueva instancia. A continuación se presenta un ejemplo:

```
Dim todayVar As New Date( );
```

**No devuelve ningún tipo**

Esta es la segunda característica que define a un constructor. Un constructor nunca tiene un tipo de devolución, ni siquiera **nothing**.

**No se requieren argumentos**

Es posible declarar constructores que tomen argumentos. Sin embargo, el constructor predeterminado generado por el compilador no espera argumentos.

**Todos los campos se inicializan en cero**

Esto es importante. El constructor predeterminado generado por el compilador inicializa de manera implícita todos los campos no estáticos como se muestra a continuación:

Campos numéricos (como **int**, **double** y **decimal**) se inicializan en cero.

Los campos del tipo **bool** se inicializan en **false**.

Los tipos de referencia (cubiertos en un módulo anterior) se inicializan en **null**.

Los campos del tipo **struct** se inicializan para contener valores de cero en todos sus elementos.

**Accesibilidad pública**

Esto permite que se generen nuevas instancias del objeto.

**Escribir su propio constructor predeterminado**

Existen diferentes casos en los cuales puede no ser adecuado el constructor predeterminado generado por el compilador:

El acceso público en ocasiones no es adecuado.

La inicialización en cero a veces no es adecuada.

El código invisible es difícil de mantener. No puede ver el código del constructor predeterminado.

Si el constructor predeterminado generado por el compilador no es adecuado, deberá escribir su propio constructor. VB.NET le ayuda a hacer esto.

Si escribe su propio constructor, el compilador VB.NET no generará un constructor predeterminado y será el suyo el que se utilice.

Los constructores son tipos especiales de métodos. Igual que como puede sobrecargar los métodos, también puede sobrecargar los constructores.

**¿Qué es sobrecargar?**

Sobrecargar es un término técnico para declarar dos o más métodos en el mismo alcance con el mismo nombre. El siguiente código muestra un ejemplo:

```
Class Overload
   Public Sub Method()
         …
   End Sub
   Public Sub Method(ByVal X As Integer)
         …
   End Sub
End Class
Class Use
   Shared Sub Main()
      Dim o As New Overload
      o.Method()
      o.Method(42)
   End Sub
End Class
```

En este ejemplo de código, dos métodos llamados **Method** se declaran en el alcance de la clase **Overload**, y ambos se invocan en **Use.Main**. No existe ambigüedad,

debido a que el número y los tipos de argumentos determinan qué tipo de método se invoca.

**Constructores privados**

Imaginemos que desea crear una clase para proporcionar cierta funcionalidad, pero no desea que los usuarios de esa clase creen objetos con base en esa clase. ¿Cómo lo lograría? Puede lograrlo utilizando un método compartido únicamente dentro de la clase. En este caso, no hay necesidad de crear instancias en la clase (crear un objeto) para invocar su método.

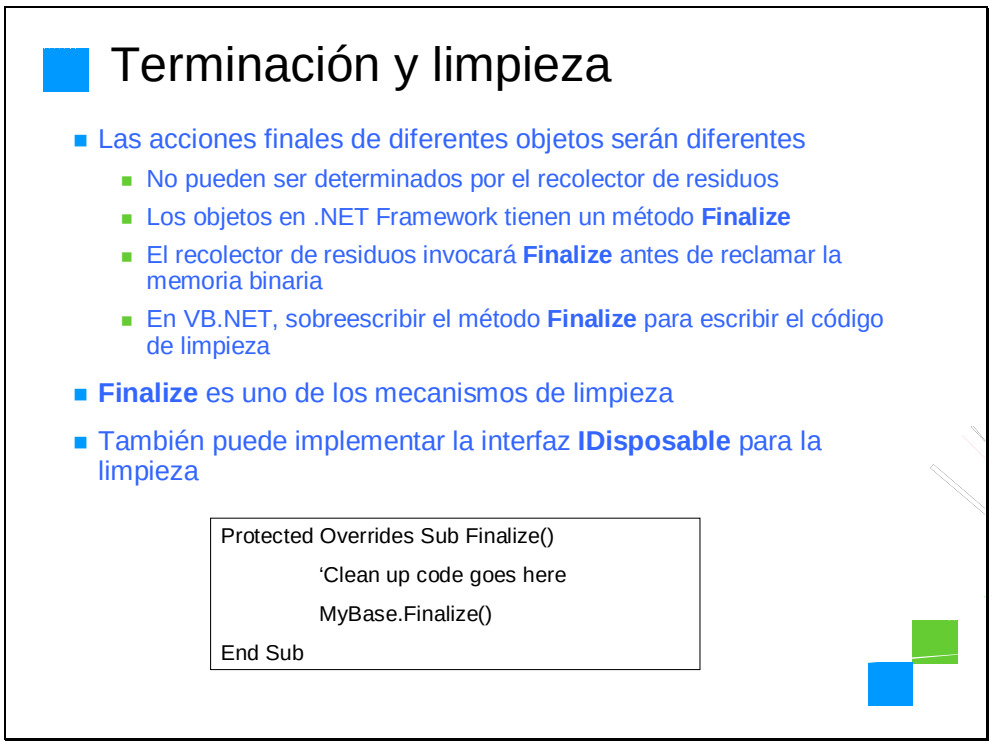
El compilador generará un constructor predeterminado con acceso público, permitiéndole crear objetos. Puede evitar que los objetos se creen al declarar un constructor privado para la clase y si también declara al constructor como privado, se dejarán de crear los objetos.

**Constructores compartidos**

Igual que como el constructor de instancias garantiza que un objeto se encuentre en un estado inicial bien definido antes de ser utilizado, un constructor compartido garantiza que la clase se encuentra en un estado inicial bien definido antes de ser utilizada.

Una vez que el cargador de clase carga una clase que será utilizada próximamente, pero antes de que continúe la ejecución normal, ejecuta el constructor estático para esa clase. Debido a este proceso, tiene garantizado que las clases siempre se inicialicen antes de ser utilizadas.

## Diapositiva 19




¿Cómo escribir un código de tal manera que nuestros objetos puedan realizar cualquier procesamiento requerido de limpieza antes de que terminen?. Existen dos puntos en el tiempo cuando es más adecuado hacer esta limpieza: inmediatamente cuando se libera la última referencia al objeto, y justo antes de que se destruye finalmente el objeto por parte del mecanismo de recolección de basura.
No hay una forma automática para manejar la limpieza al liberar la última referencia de un objeto. Implementar la interfaz IDisposable proporciona una posible solución.

**Finalize:**

El mecanismo de recolección de basura proporciona una forma por medio de la cual podemos ejecutar el código inmediatamente antes de terminar nuestro objeto. El código de recolección de residuos invocará el método **Finalize** del objeto.

```
Protected Overrides Sub Finalize()
    'Clean up code goes here
    MyBase.Finalize()
End Sub
```

**NOTA:** Este código utiliza el alcance **Protected** y la palabra clave **Overrides**. Esto es obligatorio para el método **Finalize**. Hablaremos con más detalle sobre de esto cuando nos refiramos a la Herencia. Observe que sólo podemos escribir algún código de limpieza, pero también podemos hacer que una invocación sea al método

**Finalize** de la clase base. También es importante saber que este método puede invocarse mucho después de que el objeto se des referencia por el último bit del código cliente, es posible que incluso algunos minutos después.

**IDisposable:**

Algunas veces, el comportamiento Finalize no es aceptable, como cuando tenemos un objeto que utiliza algún recurso costoso o limitado y necesitamos asegurar que el recurso esté liberado, tan pronto como el objeto deja de estar en uso.

Esto puede lograrse al implementar un método que invocará el código cliente para forzar a nuestro objeto a limpiar y liberar sus recursos.

Ésta no es la mejor solución, pero ayuda. Debe recordar que este método no se invocará automáticamente y debe invocarlo el código cliente utilizando al objeto.

.NET Framework proporciona la interfaz IDisposable que formaliza la declaración del método Dispose.

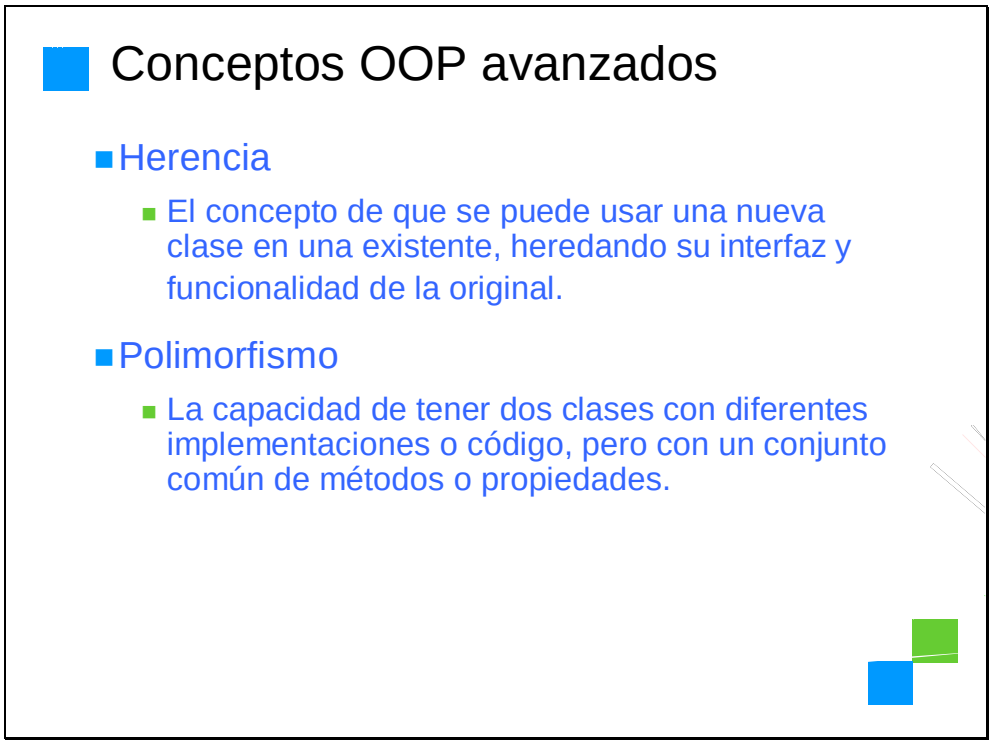
Cualquier clase que se derive de System.ComponentModel.Component adquiere ya esta interfaz. Esto incluye formularios y controles para las clases Interfaz y .NET, entre otras.

Ejemplo:

```
Public Class Person
   Implements IDisposable
         …
         …
         …
   Private Sub Dispose() Implements IDisposable.Dispose
      colNickNames = Nothing
   End Sub
End Class
….
….
Private Sub Form1_Closed(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Closed
   CType(mobjPerson, IDisposable).Dispose()
   mobjPerson = Nothing
End Sub
```

El método OnClosed se ejecuta al cerrar el formulario. Antes de que podamos des referenciar al objeto Person, podemos invocar ahora su método Dispose. Ya que este método es parte de la interfaz secundaria, necesitamos utilizar el método CType() para acceder a esa interfaz específica y así invocar el método. CType() nos permite indicar la interfaz específica por la que deseamos acceder al objeto. Una vez que utilizamos la interfaz, podemos llamar al método **Dispose** para pedirle al objeto que haga alguna limpieza antes de liberar nuestra referencia.

## Diapositiva 20




Dos de los conceptos de la OOP más importantes son la Herencia y el Polimorfismo.

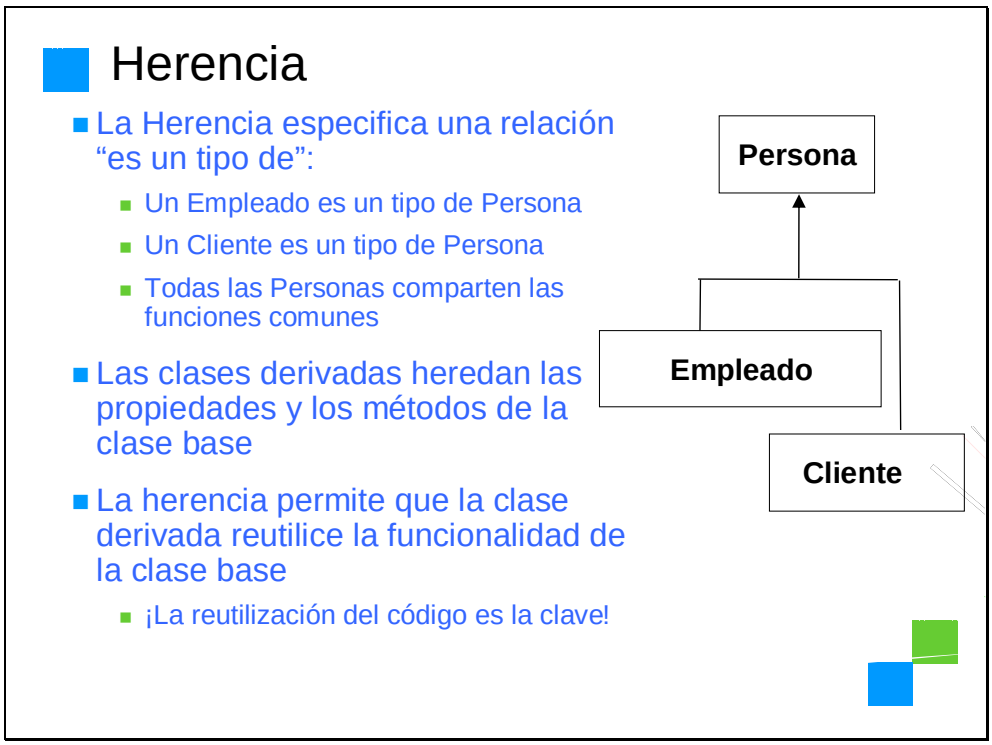
**Herencia**

Concepto que indica que se puede usar una nueva clase en una existente, heredando su interfaz y funcionalidad de la original.

**Polimorfismo**

La capacidad de tener dos clases con diferentes implementaciones o código, pero con un conjunto común de métodos o propiedades.

## Diapositiva 21




La Herencia es una relación entre una clase general y un tipo más específico de esa clase. La Herencia especifica una relación “es un tipo de”: una cosa (por ejemplo un Empleado) es un tipo de cosa más general (por ejemplo una Persona).
Cuando diseña una aplicación, por lo regular tiene que administrar artículos parecidos, pero no idénticos. El principio orientado a objetos de herencia le permite crear una clase generalizada y después generar clases más especializadas a partir de ella. La clase general se conoce como la clase **base**. Una clase más específica se conoce como la clase **derivada**. Las clases derivadas heredan las propiedades y los métodos de la clase base.
La herencia es la capacidad de utilizar todas las funcionalidades de una clase existente, y ampliar esas capacidades sin rescribir la clase original. La herencia le permite incluir un conjunto de comportamiento común, definido como propiedades y métodos, en una clase base y reutilizarlo en clases derivadas.

**Razones para utilizar la herencia**

La herencia se recomienda ya que usted quiere evitar escribir el mismo código una y otra vez. Si tiene dos clases separadas, y cada una tiene que implementar una propiedad **FirstName** y **LastName**, va a tener que duplicar el código. Si desea cambiar la implementación de una de estas propiedades, necesita encontrar todas las clases que tienen implementadas estas propiedades para realizar los cambios.

Esto no sólo lleva tiempo, sino que también eleva el riesgo de introducir un error en las diversas clases.

Algo que tiene que recordar al momento de considerar el uso de la herencia, es que la relación entre las dos clases debe ser un tipo de relación “es un”. Por ejemplo, un Empleado es una Persona, y un Gerente es una Persona, de manera que estas dos clases se pueden heredar a partir de la clase Persona. Sin embargo, usted no debe heredar una clase Cajero Electrónico a partir de una clase Persona, ya que un Cajero Electrónico no es una Persona.

**Ejemplo**

**1) Sin utilizar la herencia:**

```
Class Employee
   Public Firstname As String
   Public Lastname As String
   Public Sub Promote()
      Public EmployeeLevel As Integer
End Class
Class Manager
   Public Firstname As String
   Public Lastname As String
   Public Sub Promote()
      Public ManagedGroup As String
End Class
```

**2) Utilizando la herencia:**

```
Class Person
   Public Firstname As String
   Public Lastname As String
   Public Sub Promote()
End Class
Class Employee
        Inherits from Person
   Public EmployeeLevel As Integer
End Class
Class Manager
        Inherits from Person
   Public ManagedGroup As String
End Class
```

En este segundo ejemplo, tanto las clases Employee como Manager cuentan con los campos Firstname (Nombre) y Lastname (Apellido), y el método Promote, ¡a pesar de que no escribimos el código específico para incluirlos!

Ellos heredan estos campos y métodos a partir de la clase Person, al reutilizar el código de la clase Person.

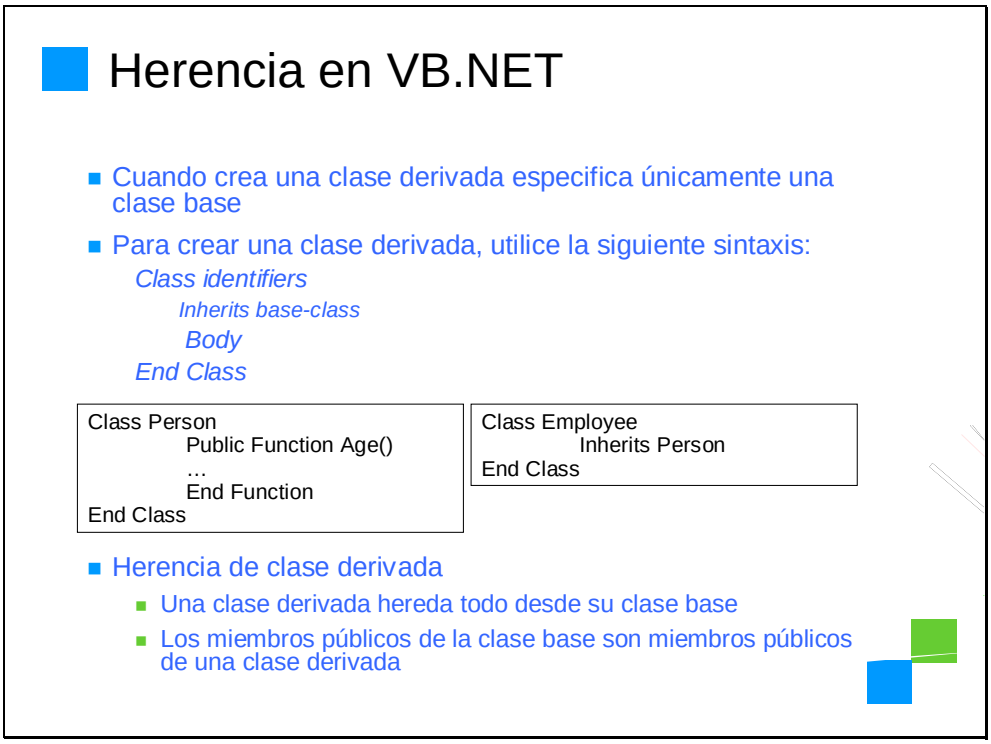
Existen tres tipos de herencia que soporta .NET: la **implementación**, la **interfaz** y la **visual**.

La herencia de **implementación** se refiere a la capacidad de utilizar propiedades y métodos de la clase base sin códigos adicionales.

La herencia de **interfaz** se refiere a la capacidad de utilizar únicamente los nombres de las propiedades y de los métodos, pero la clase derivada debe proporcionar la implementación.

La herencia **visual** se refiere a la capacidad de una forma derivada (clase) de utilizar las representaciones visuales (clase) de las formas base, así como los códigos implementados.

## Diapositiva 22




Como mencionamos anteriormente, la herencia le permite crear una clase generalizada y después generar más clases especializadas a partir de ella. La clase general se conoce como la clase **base**. Una clase más específica se conoce como la clase **derivada**. Las clases derivadas heredan las propiedades y los métodos de la clase base.

Cuando crea una clase derivada, especifica únicamente una clase base. No puede tener herencias múltiples en VB.NET.

Puede utilizar diversas clases que están disponibles en la biblioteca de clase Microsoft .NET, como clases base a partir de las cuales puede generar clases nuevas. Por ejemplo, cuando crea una aplicación Windows nueva en Visual Studio .NET, el formulario principal que se crea a través del ambiente de desarrollo integrado es una clase nueva que se genera a partir de la clase **System.Windows.Forms**.

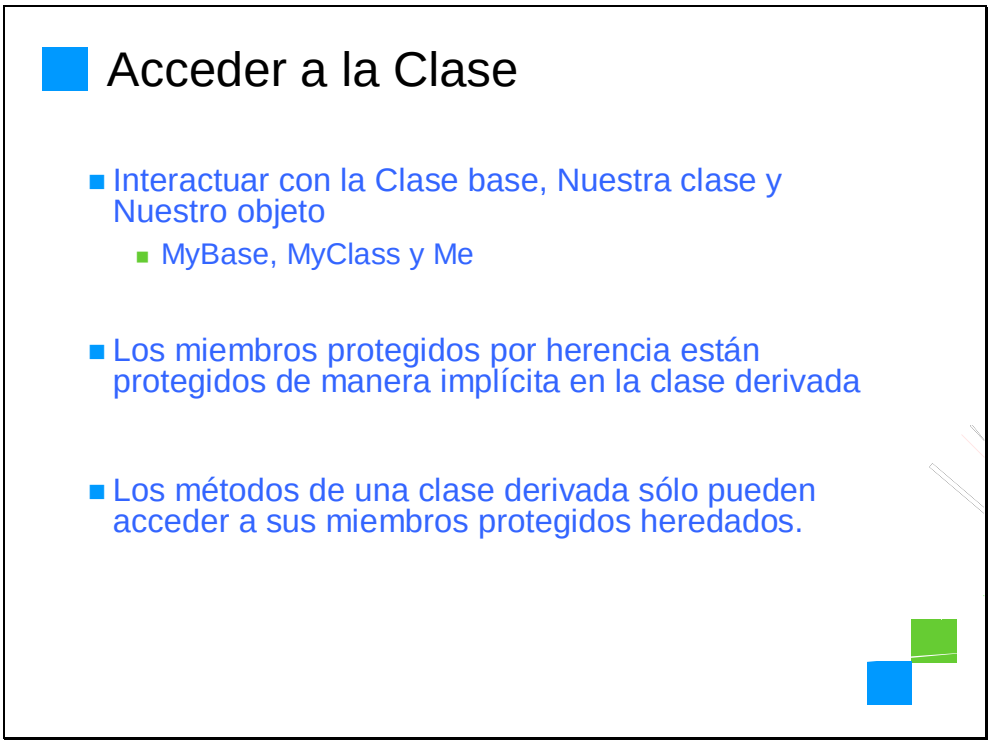
Cuando define una clase nueva, crea un tipo de referencia nuevo.

Evite sobre utilizar la herencia en sus aplicaciones. Por ejemplo, a pesar de que es posible crear nuevas versiones de componentes de la interfaz Windows, como botones, es muy raro que exista una razón para hacerlo.

Puede heredar a partir de cualquier clase que no esté sellada, de manera que pueda heredar las clases derivadas. Una clase base puede tener cualquier número de clases derivadas.

Un cambio a una clase base es automáticamente un cambio para todas las clases derivadas.

## Diapositiva 23




Existen tres palabras clave especiales que permiten interactuar con representaciones importantes de objeto y clase:

**MyBase –** Esta palabra clave hace referencia sólo a la clase padre inmediata, y trabaja como una referencia de objeto. Esto significa que podemos invocar métodos en **MyBase**, sabiendo que se invocan como si tuviéramos una referencia con un objeto o nuestro tipo de datos de la clase padre. Observe que no existe una manera de navegar directamente a la cadena de herencia más allá de nuestra clase padre inmediata.

Se puede utilizar la palabra clave **MyBase** para invocar o usar cualquier elemento Público, Amigo o Protegido de la clase padre. Esto incluye todos los elementos directamente en la clase base, y también los elementos que heredó la clase base de otras clases mayores en la cadena de herencia.

```
Public Shadows Property Name() As String
    Get
        Return MyBase.Name(NameTypes.Informal)
    End Get
    Set(ByVal Value As String)
        MyBase.Name = Value
    End Set
End Property
```

**MyClass** – Esta palabra clave se comporta como una variable de objeto que se refiere a la instancia actual de la clase como se implementó originalmente. **MyClass** es similar a **Me**, pero todas las invocaciones al método en la misma se tratan como si el método fuera **NotOverridable**. Por lo tanto, el método invocado no queda afectado por la sobrescritura en una clase derivada.

El siguiente ejemplo compara **Me** y **MyClass**.

```
Class BaseClass
   Public Overridable Sub MyMethod()
      MsgBox("Base class string")
   End Sub
   Public Sub UseMe()
      Me.MyMethod() ' Use calling class's version, even if an override.
   End Sub
   Public Sub UseMyClass()
      MyClass.MyMethod() ' Use this version and not any override.
   End Sub
End Class
Class DerivedClass
   Inherits BaseClass
   Public Overrides Sub MyMethod()
      MsgBox("Derived class string")
   End Sub
End Class
Class TestClasses
   Sub StartHere()
      Dim TestObj As DerivedClass = New DerivedClass
      TestObj.UseMe() ' Displays "Derived class string".
      TestObj.UseMyClass() ' Displays "Base class string".
   End Sub
End Class
```

Aunque DerivedClass sobrescribe MyMethod, la palabra clave **MyClass** en UseMyClass invalida los efectos de la sobrescritura, y el compilador resuelve la invocación a la versión de la clase base de MyMethod.

No se puede utilizar **MyClass** dentro de un método **Shared**, pero puede usarlo dentro de un método de instancia para tener acceso a un miembro compartido de la clase.

**Me** – Esta palabra clave proporciona una forma para hacer referencia a la instancia específica de una clase o estructura en la que se esté ejecutando actualmente el código. **Me** se comporta como una variable de objeto o una variable de estructura que hace referencia a la instancia actual. Usar **Me** es particularmente útil para pasar información sobre la instancia que se ejecuta en ese momento de una clase o estructura a un procedimiento en otra clase, estructura o módulo.

Por ejemplo, suponga que tiene el siguiente procedimiento en un módulo:

```
Sub ChangeFormColor(FormName As Form) Randomize() FormName.BackColor =
Color.FromArgb(Rnd() * 256, Rnd() * 256, Rnd() * 256) End Sub
```

Puede invocar este procedimiento y pasar la instancia actual de la clase **Form** como un argumento, utilizando la siguiente instrucción:

```
ChangeFormColor(Me)
```

Se utiliza la palabra clave **Me** en los contextos de las instrucciones Class y Structured.

El significado del modificador de acceso **protected** depende de la relación entre la clase que tiene el modificador y la clase que busca el acceso a miembros que utilizan el modificador.

Los miembros de una clase derivada pueden acceder a todos los miembros protegidos de sus clases base. Para una clase derivada, la palabra clave **protected** se comporta como la palabra clave **public**.

Sin embargo, entre dos clases que no están relacionadas por una relación de clase derivada y de clase base, los miembros protegidos de una clase actúan como los miembros privados para la otra clase.

**Miembros protegidos por la herencia**

Cuando una clase derivada hereda un miembro protegido, ese miembro también es un miembro protegido de implícitamente de una clase derivada. Esto significa que los miembros protegidos están accesibles para todas las clases derivadas de manera directa e indirecta de la clase base.

Esto se muestra en el siguiente ejemplo:

```
Class Base
   Protected name As String
End Class
Class Derived
   Inherits Base
End Class
Class FurtherDerived
   Inherits Derived
   Sub Compiles()
      Console.WriteLine(name)   'No errors
   End Sub
End Class
```

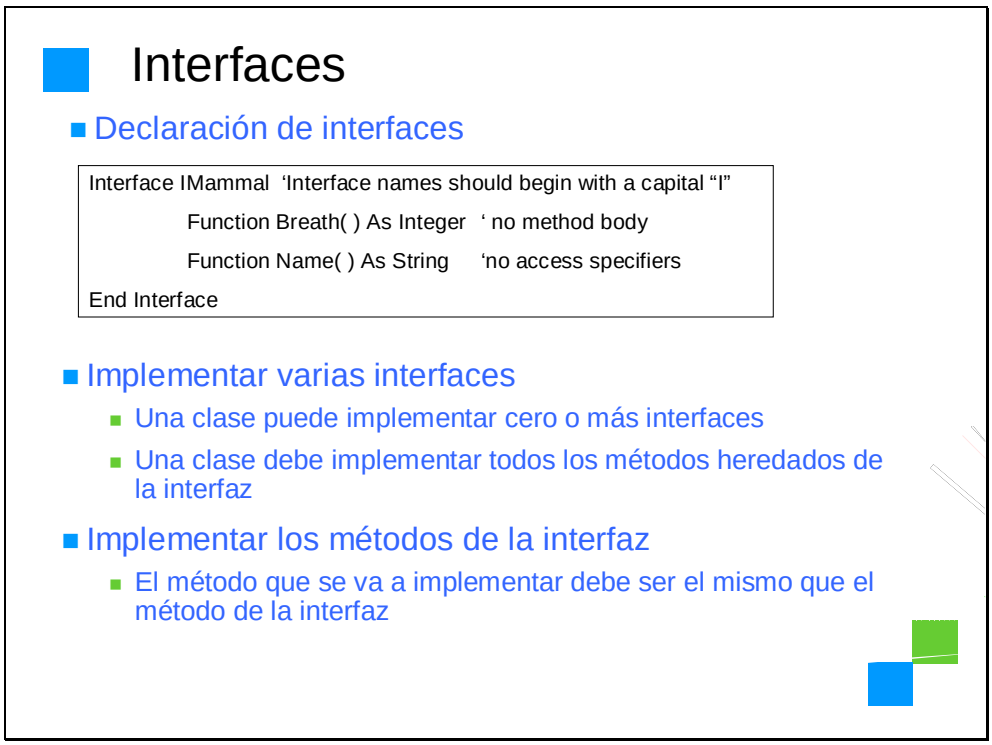
**Miembros y métodos protegidos**

Los métodos de una clase derivada sólo pueden acceder a sus miembros protegidos heredados. No pueden acceder a los miembros protegidos de la clase base a través de las referencias a la clase base.

Por ejemplo, el siguiente código generará un error:

```
Class Dog
   Inherits Animal
   Sub Fails(ByVal a As Animal)
      Console.WriteLine(a.Age)  'Compile-time error
   End Sub
End Class
```

# Diapositiva 24




Una interfaz especifica un contrato sintáctico y semántico al cual se deben adherir todas las clases derivadas. Específicamente, una interfaz describe la parte del **what** (qué) del contrato, y las clases que implementan la interfaz describen la parte del **how** (cómo) del mismo.

**Declarar interfaces**

Una interfaz parece una clase sin código. Usted declara una interfaz de una manera parecida a la manera en que declara una clase. Para declarar una interfaz en VB.NET, utilice la palabra clave **Interface** en lugar de **Class**.

Se recomienda que todos los nombres de la interfaz tengan como prefijo la letra mayúscula "**I**" antes.

Por ejemplo, utilice **IMammal** en lugar de **Mammal**.

**Función de las interfaces**

Las siguientes son dos funciones importantes de las interfaces.

- **Los métodos de la interfaz son públicos implícitamente**
  Los métodos declarados en una interfaz son públicos implícitamente. Por lo tanto, no se permiten los modificadores de acceso **públicos** explícitos.
- **Los métodos de interfaz no contienen cuerpos de métodos**.

**Implementar varias interfaces**

A pesar de que VB.NET permite únicamente la herencia única, le permite implementar varias interfaces en una clase única.

**Implementación de la interfaz**

Una clase puede implementar cero o más interfaces, pero puede ampliar explícitamente no más de una clase. Por el contrario, una interfaz puede ampliar cero o más interfaces.

**Capacidad de acceso**

Una clase puede ser más accesible que sus interfaces base. Por ejemplo, puede declarar una clase pública que implemente una interfaz privada. Sin embargo, una interfaz no puede ser más accesible que cualquiera de sus interfaces base. Es un error declarar una interfaz pública que amplíe una interfaz privada.

**Métodos de la interfaz**

Una clase debe implementar todos los métodos de cualquier interfaz que amplía, independientemente de si las interfaces se heredan de manera directa o indirecta.

**Implementar los métodos de la interfaz**

Cuando una clase implementa una interfaz, debe implementar todo método declarado en esa interfaz. Este requisito es práctico, ya que las interfaces no pueden definir sus propios cuerpos de métodos.

El método que la clase implementa debe ser idéntico al del método de la interfaz en todos los sentidos. Debe tener el mismo:

**Acceso**

Debido a que el método de interfaz es público implícitamente, esto significa que el método de implementación se debe declarar público explícitamente. Si se omite el modificador de acceso, entonces el método por predeterminación será privado.

**Tipo de devolución**

Si el tipo de devolución de la interfaz se declara como **T**, entonces el tipo de devolución en la clase de implementación no se puede declarar como un tipo derivado de **T**; debe ser **T**. En otras palabras, la co-variación del tipo de devolución no está soportada en VB.NET.

El siguiente ejemplo muestra la forma de usar la instrucción **Implements** para implementar las propiedades, métodos y eventos de una interfaz.

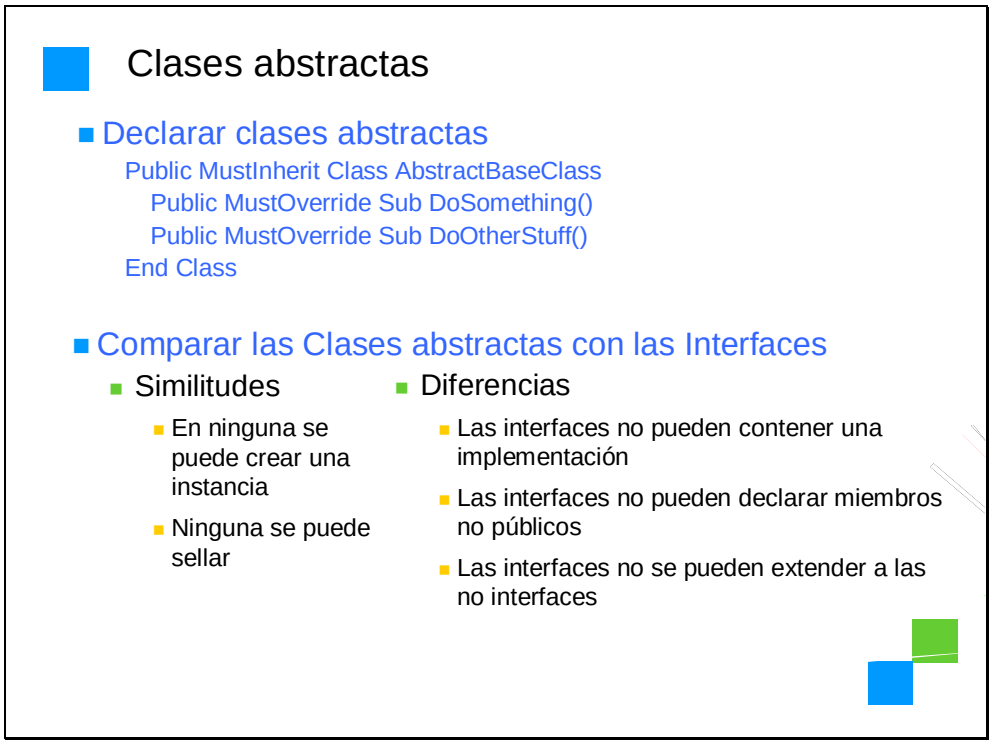
Este ejemplo define una interfaz llamada **ICustomerInfo** con una propiedad, método y un evento.

La Clase **CustomerInfo** implementa a todos los miembros definidos en la interfaz.

```
Interface ICustomerInfo
    'Declare the interface
    Property CustomerName() As String
    Sub UpdateCustomerStatus()
    Event UpdateComplete()
End Interface
Public Class CustomerInfo
    'The CustomerInfo class implements the ICustomerInfo interface
    Implements ICustomerInfo
    Private CustomerNameValue As String
    Public Event UpdateComplete() Implements ICustomerInfo.UpdateComplete
    Public Property CustomerName() As String Implements
ICustomerInfo.CustomerName
        Get
            Return CustomerNameValue
        End Get
        Set(ByVal Value As String)
            CustomerNameValue = Value
        End Set
    End Property
    Public Sub UpdateCustomerStatus() Implements
ICustomerInfo.UpdateCustomerStatus
        'code to update the status
        RaiseEvent UpdateComplete()
    End Sub
End Class
Public Sub TestImplements()
    Dim Cust As New CustomerInfo

    AddHandler Cust.UpdateComplete, AddressOf HandleUpdateComplete
    Cust.CustomerName = "Federico"
    MsgBox("Customer name is: " & Cust.CustomerName)
    Cust.UpdateCustomerStatus()
End Sub
Sub HandleUpdateComplete()
    MsgBox("Update is complete.")
End Sub
```

# Diapositiva 25




Por lo regular, resulta útil crear una clase que contenga métodos que las clases derivadas deben implementar, pero no la misma clase base.

Un **método abstracto** es un método vacío, sin implementación. En lugar de eso, las clases derivadas deben proporcionar una implementación.

Una **clase abstracta** es una clase que puede obtener miembros abstractos, aunque no se requiera hacerlo de esta forma. Cualquier clase que contenga miembros abstractos debe ser abstracta. Una clase abstracta también puede contener miembros no abstractos.

Debido a que el propósito de una clase abstracta es actuar como una clase base, no es posible crear la instancia directamente de una clase abstracta, tampoco se puede sellar.

Las clases abstractas se usan para proporcionar implementaciones parciales de clase que pueden completar las clases concretas derivadas. Las clases abstractas son particularmente útiles para proporcionar una implementación parcial de una interfaz que puede ser reutilizada por varias clases derivadas.

**Declarar clases abstractas**

Usted declara una clase abstracta utilizando la palabra clave **MustInherit**.

**Ejemplo**

El siguiente ejemplo muestra cómo crear una clase abstracta **Animal** con un método **Comer**, y cómo generarlo para crear una clase **Perro**:

```
Public MustInherit Class Animal
    Public MustOverride Sub Comer()
End Class
Public Class Perro
    Inherits Animal
    Public Overrides Sub Comer()
                    …
    End Sub
End Class
```

Las normas reguladoras de uso de una clase abstracta son casi iguales a las mismas que regulan las clases no abstractas. La única diferencia entre utilizar clases abstractas y no abstractas son:

*No puede crear la instancia de una clase abstracta*.

En este sentido, las clases abstractas son como las interfaces.

*Puede crear un método abstracto en una clase abstracta*.

Una clase abstracta puede declarar un método abstracto, pero no puede declarar una clase no abstracta.

*Las funciones comunes de las clases abstractas y de las clases no abstractas son*:

- Extensibilidad limitada

  Una clase abstracta puede ampliar a lo sumo una clase o una clase abstracta. Observe que una clase abstracta puede emplear una clase no abstracta
- Interfaces múltiples

  Una clase abstracta puede implementar varias interfaces
- Métodos heredados de interfaz

  Una clase abstracta debe implementar todos los métodos heredados de interfaz

**Comparar las clases abstractas con las interfaces**

Tanto las clases abstractas como las interfaces existen para derivarse (o implementarse). Sin embargo, una clase puede emplear a lo sumo una clase abstracta, de manera que debe tener más cuidado al derivar a partir de una clase abstracta que al derivar a partir de una interfaz. Reserve el uso de clases abstractas para implementar relaciones verdaderas “es un”.

Las similitudes entre las clases abstractas y las interfaces son que:

- No pueden crear instancias.

Esto significa que no se pueden utilizar directamente para crear objetos.

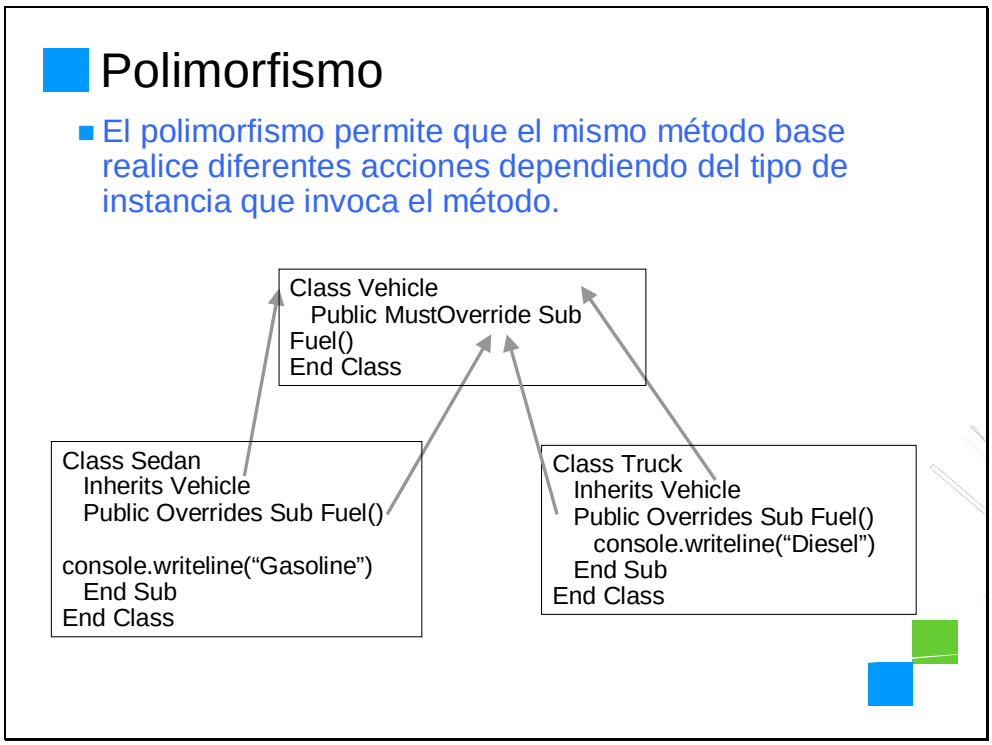
- No se pueden sellar.
  Esto es aceptable, ya que una interfaz sellada no se puede implementar.

Las diferencias entre las clases abstractas y las interfaces son las siguientes:

- Interfaces
  No puede contener implementaciones
  No puede declarar miembros no públicos
  Puede ampliar únicamente otras interfaces
- Clases abstractas
  Puede contener implementaciones
  Puede declarar miembros no públicos
  Puede ampliar otras clases, que pueden ser no abstractas

Al comparar las similitudes y las diferencias entre las clases abstractas y las interfaces, piense en las clases abstractas como clases no terminadas que contienen planos, para lo que es necesario que se termine.

## Diapositiva 26




Literalmente, polimorfismo significa **muchas formas** o **muchas figuras**.

Es el concepto de que un método declarado en una clase base se puede implementar en muchas formas distintas en las diferentes clases derivadas.

El polimorfismo permite que el mismo método base realice diferentes acciones dependiendo, del tipo de tiempo de ejecución de la instancia que invoca el método.

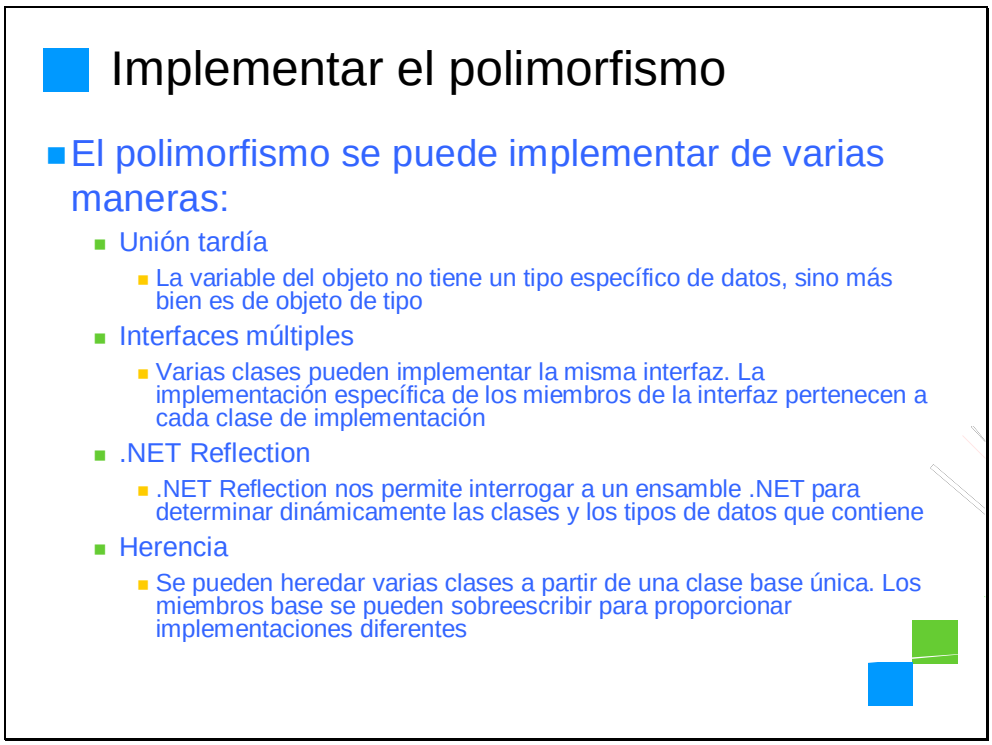
El polimorfismo es un concepto orientado a objetos que le permite tratar sus clases derivadas de una manera parecida, a pesar de que son diferentes.

Cuando crea clases derivadas, proporciona más funcionalidades especializadas; el polimorfismo le permite tratar a estos nuevos objetos de una forma general.

Para proporcionar el polimorfismo, se debe de llevar a cabo lo siguiente:

- El nombre de método debe residir en la clase base
- Las implementaciones del método deben residir en las clases derivadas

## Diapositiva 27




El polimorfismo es la capacidad de las clases para proporcionar diferentes implementaciones de métodos que se invocan con el mismo nombre.

El polimorfismo permite que un método de una clase se invoque sin importar de qué implementación específica proviene.

El polimorfismo se puede implementar de diversas maneras:

**Unión tardía**

La variable de objeto no tiene un tipo específico de datos, sino más bien es de tipo objeto.  Entonces en el tiempo de ejecución, cuando se invoca efectivamente el código, tratará de invocar dinámicamente la variable o método del objeto.

Obviamente, existe un nivel de peligro cuando se usa una unión tardía, ya que un simple error de captura puede introducir errores que sólo se descubrirán cuando se ejecute en realidad la aplicación.

Sin embargo, también hay una gran flexibilidad, ya que el código que hace uso de la unión tardía puede hablarle a cualquier objeto desde cualquier clase, siempre que esos objetos implementen los métodos requeridos.

Observe que no existe una penalidad sustancial por usar una unión tardía.

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles button1.Click
        Dim obj As New Encapsulation
        ShowDistance(obj)
    End Sub
    Private Sub ShowDistance(ByVal obj As Object)
        MsgBox(obj.DistanceTo(10, 10))
    End Sub
```

**Interfaces multiples**

Varias clases pueden implementar la misma interfaz.

La implementación específica de los miembros de la interfaz pertenece a cada clase implementada.

En este ejemplo, observará que en lugar de aceptar el parámetro utilizando el tipo de datos genérico Object, ahora estamos aceptando un parámetro IShared, un tipo sólido de datos conocido tanto por IDE como por el compilador.

Dentro del código mismo, invocará el método CalculateDistance, como se define en esa interfaz.

Esta rutina ahora puede aceptar cualquier objeto que implemente IShared, sin importar la clase en la que se creó ese objeto, o las interfaces que puede implementar ese objeto.  Todo lo que le importa es que implementa IShared.

```
Public Interface IShared
    Function CalculateDistance(ByVal X As Single, ByVal Y As Single) As Single
End Interface
Public Class Encapsulation
    Implements IShared
    Private msngX As Single
    Private msngY As Single
    Public Function DistanceTo(ByVal X As Single, ByVal Y As Single) As Single
Implements IShared.CalculateDistance
                Return Sqrt((X-msngX) ^ 2 + (Y – msngY) ^ 2)
    End Function
End Class
Public Class Poly
    Implements IShared
    Public Function DistanceTo(ByVal X As Single, ByVal Y As Single) As Single
Implements IShared.CalculateDistance
        Return X + Y
    End Function
End Class
…
…
Private Sub ShowDistance(ByVal obj As IShared)
    MsgBox(obj.CalculateDistance(10, 10))
End Sub
```

**.NET Reflection**

.NET Reflection nos permite interrogar a un ensamble .NET para determinar dinámicamente las clases y los tipos de datos que contiene. Entonces, podemos utilizar la reflexión para cargar el ensamble en nuestro proceso, crear instancias de esas clases e invocar sus métodos. Cuando utilizamos la unión tardía, VB.NET hace uso del espacio de nombre System.Reflection de .NET detrás de las escenas en representación nuestra.

**Herencia**

La mayoría de los sistemas de programación orientada a objetos, proporcionan polimorfismo a través de la herencia.

El polimorfismo basado en herencias involucra definir los métodos en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas.

Por ejemplo, podría definir una clase, BaseTax, que proporcione una funcionalidad de línea de base para calcular el impuesto sobre las ventas en el estado.

Las clases derivadas de BaseTax, tal como CountyTax o CityTax, pueden implementar métodos como CalculateTax, según sea apropiado.

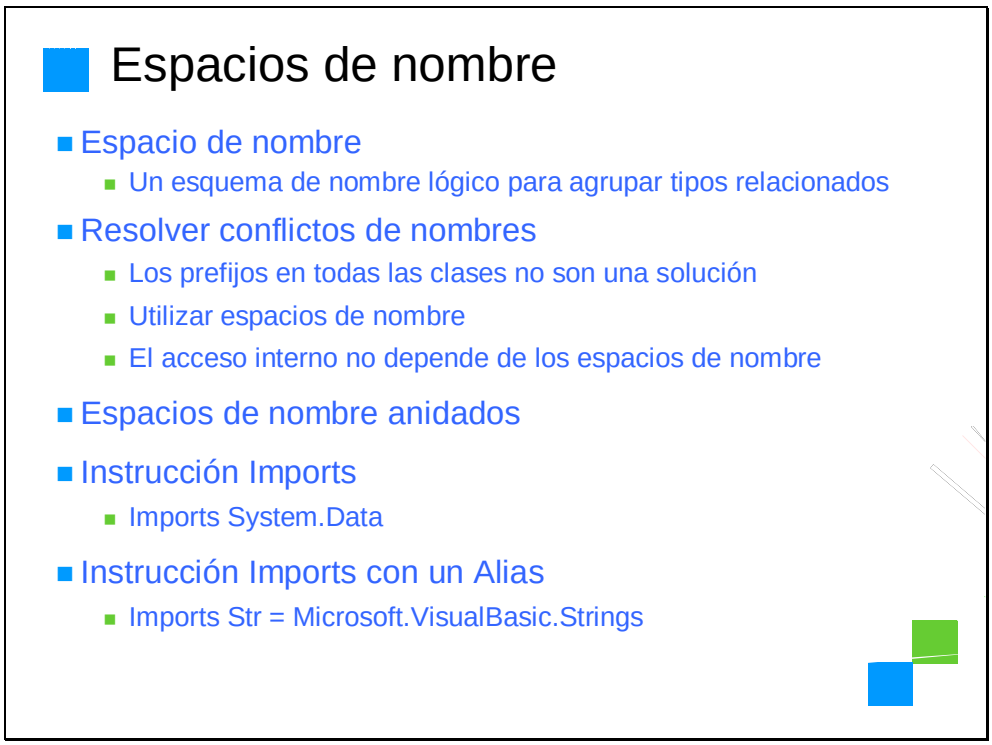
El polimorfismo proviene del hecho de que usted podría invocar el método CalculateTax de un objeto que pertenece a cualquier clase derivada de BaseTax, sin saber a qué clase pertenecía el objeto.

El procedimiento TestPoly en el siguiente ejemplo demuestra el polimorfismo basado en la herencia:

```
Const StateRate As Double = 0.053 ' %5.3 State tax
Const CityRate As Double = 0.028 ' %2.8 City tax
Public Class BaseTax
   Overridable Function CalculateTax(ByVal Amount As Double) As Double
      Return Amount * StateRate ' Calculate state tax.
   End Function
End Class
Public Class CityTax
   ' This method calls a method in the base class
   ' and modifies the returned value.
   Inherits BaseTax
   Private BaseAmount As Double
   Overrides Function CalculateTax(ByVal Amount As Double) As Double
      ' Some cities apply a tax to the total cost of purchases,
      ' including other taxes.
      BaseAmount = MyBase.CalculateTax(Amount)
      Return CityRate * (BaseAmount + Amount) + BaseAmount
   End Function
End Class
Sub TestPoly()
   Dim Item1 As New BaseTax
   Dim Item2 As New CityTax
   ShowTax(Item1, 22.74) ' $22.74 normal purchase.
   ShowTax(Item2, 22.74) ' $22.74 city purchase.
End Sub
Sub ShowTax(ByVal Item As BaseTax, ByVal SaleAmount As Double)
   ' Item is declared as BaseTax, but you can
   ' pass an item of type CityTax instead.
   Dim TaxAmount As Double
   TaxAmount = Item.CalculateTax(SaleAmount)
   MsgBox("The tax is: " & Format(TaxAmount, "C"))
End Sub
```

En este ejemplo, el procedimiento ShowTax acepta un parámetro llamado Item del tipo BaseTax, pero también podría pasar cualquiera de las clases derivadas de la clase shape, como CityTax. La ventaja de este diseño es que puede agregar nuevas clases derivadas de la clase BaseTax sin cambiar el código de cliente en el procedimiento ShowTax.

## Diapositiva 28




**Espacio de nombre**

Se utilizan los espacios de nombre como un sistema organizacional - una forma de presentar los componentes del programa que están expuestos a otros programas y aplicaciones.

Los espacios de nombre siempre son Public (Públicos); por lo tanto, la declaración de un espacio de nombre no puede incluir modificadores de acceso. Sin embargo, los componentes dentro del espacio de nombre pueden tener acceso a Public o Friend. Si no se establece, el tipo predeterminado de acceso es Friend.

**¿Por qué se requieren los espacios de nombre?**

El alcance de un nombre es la región del texto del programa al que se puede referir para el nombre sin calificación.

Cuando no está al alcance un nombre, usted no puede utilizarlo sin calificación. Esto sucede por lo regular, ya que ha terminado el alcance en el que se declara el nombre.

¿Cómo puede manejar los problemas potenciales de dos clases en el mismo alcance y que tienen el mismo nombre? No es una buena idea agregar prefijos a todos los nombres de clase.  Usted puede utilizar nombres de espacio para resolver el conflicto de nombres. El acceso interno no depende de los espacios de nombre.

**¿Qué sucede si no utiliza los espacios de nombre?**

Si no utiliza los espacios de nombre, es probable que ocurran conflictos de nombre. Por ejemplo, en un proyecto grande que tiene muchas clases pequeñas, puede cometer fácilmente el error de dar el mismo nombre a dos clases.

Tome como ejemplo un proyecto grande que se divide en un número de subsistemas y que cuentan con equipos de trabajo separados en subsistemas separados. Suponga que los subsistemas están divididos de acuerdo con los servicios de arquitectura, como se presenta a continuación:

**Servicios del usuario**

Una manera de permitir que los usuarios interactúen con el sistema.

**Servicios empresariales**

En la lógica de negocios se utiliza para recuperar, validar y manipular los datos de acuerdo con las reglas y negocios.

**Servicios de datos**

Datos almacenados de algún tipo y la lógica para manipular los datos.

En este proyecto con varios equipos es muy probable que ocurran conflictos de nombres. Después de todo, los tres equipos están trabajando en el mismo proyecto.

**Utilizar prefijos como una solución**

Colocar prefijos a cada clase con un calificador de subsistemas no es una buena idea, ya que los nombres se vuelven:

Largos y carecen de una capacidad de administración

Los nombres de la clase se vuelven rápidamente largos.  Incluso si se trabajan al primer nivel de granularidad, no pueden seguir trabajando sin que los nombres de clase se vuelvan verdaderamente inmanejables.

**Complejo**

Los nombres de clase se vuelven simplemente difíciles de leer.  Los programas son una forma de escritura. La gente lee programas. Entre más fácil sea de leer y de comprender un programa, más fácil será de mantener.

**Espacios de nombre anidados**

Usted puede anidar espacios de nombre dentro de otros espacios de nombre.  Las bibliotecas de clase .NET hacen esto en varios lugares para proporcionar un nivel

aún más profundo de organización de tipos. El anidado de los espacios de nombre también se puede lograr utilizando la siguiente anotación corta:

```
Namespace N1.N2
        …
        …
End Namespace
```

El siguiente ejemplo declara dos espacios de nombre:

```
Namespace N1
   Namespace N2
      Class A
                    …
      End Class
   End Namespace
End Namespace


Namespace N1.N2
   Class A
              …
   End Class
End Namespace
```

**Instrucción Imports**

El espacio de nombre Imports nombra los proyectos y ensambles de referencia. También el espacio de nombre Imports nombra los definidos dentro del mismo proyecto como el archivo en el que aparece la instrucción.

Cada archivo puede contener cualquier cantidad de instrucciones **Imports**. Las instrucciones **Imports** deben colocarse antes de cualquier declaración, incluyendo las instrucciones **Module** o **Class**, y antes de cualquier referencia a los identificadores.

El alcance de los elementos que pone a disposición una instrucción **Imports** depende de lo específico que sea usted al utilizar la instrucción **Imports**.

Por ejemplo, si sólo se especifica un espacio de nombre, todos los miembros nombrados de manera única en ese espacio de nombre, y los miembros de los módulos dentro de ese espacio de nombre, están disponibles sin calificación.

Si se especifican tanto el espacio de nombre como el nombre de un elemento de ese espacio de nombre, sólo los miembros de ese elemento están disponibles sin calificación.

No se permite definir un miembro a nivel de módulo con el mismo nombre que un alias importado.

El siguiente ejemplo importa la clase **Microsoft.VisualBasic.Strings** y le asigna un alias, Str, que se puede utilizar para acceder al método **Left**.

```
Imports Str = Microsoft.VisualBasic.Strings
Class MyClass1
    Sub ShowHello()
        MsgBox(Str.Left("Hello World!", 5)) 'This displays the word “Hello”
    End Sub
End Class
```

## Diapositiva 29


- Troelsen. *Visual Basic .NET y la Plataforma .NET: Una guía avanzada.* APRESS, 2002.
- Reynolds-Haertle. *OOP con Microsoft Visual Basic .NET y Microsoft Visual C# .NET paso por paso.* Microsoft Press, 2002.
- WWW.ASP.NET
- WWW.DOTNETJUNKIES.COM
- http://msdn.microsoft.com/vbasic/using/understanding/default.aspx
- http://www.microsoft.com/seminar/mmcfeed/mmcdisplayfeed.asp?Lang=en&Product=103364
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconimplementinginterfacesincomponents.asp
- Libros de referencia de OOP http://support.microsoft.com/default.aspx?scid=kb%3ben-us%3b138569