PROGRAMACIÓN ORIENTADA A OBJETOS

¿Qué es la POO?

• Es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones.

• Es un modo de empaquetar código de manera que se pueda volver a utilizar y mantener más fácilmente.

• Los datos (atributos) y el código de manipulación de esos datos (comportamiento) están juntos en una única entidad denominada objeto.-

Ventajas

• Reutilizar el código

• Modificar, una sola vez, varias partes de una aplicación

• Ordenar y agrupar código, los procedimientos están asociados a su estructura de datos.

• Analizar con mayor rigor la aplicación

• Eliminar código en redundancia

• Ocultar la complejidad de la implementación

Desventajas

• El tiempo de análisis se alarga considerablemente

• Debemos organizarnos con suma precisión

ETAPAS DE UN PROYECTO SOFTWARE

- ***La Fase de análisis***

*• **Define** lo qué se supone que el sistema debe cumplir.*

*• **Se realiza** mediante un conjunto de actores (usuarios, dispositivos y otros sistemas) que interactúan con el sistema y actividades a formar parte del mismo*

*• **Debe identificar** los objetos del dominio (físicos y conceptuales) que el sistema manipulará y los comportamientos e interacciones entre ellos, los cuales implementan las actividades que éste debe soportar.*

- ***La fase de diseño***

*• **Definir** cómo el sistema debe lograr sus objetivos.*

*• **Crear** un modelo de actores, actividades, objetos y comportamientos para el sistema.*

• ***Utilizar*** *UML (Lenguaje unificado de modelado) como herramienta para la programación orientada a objeto.*

- ***La fase de desarrollo***

• Es la escritura del algoritmo en código de lenguaje de programación trasladando a éste las especificaciones del diseño y aplicando los principios de la programación orientada a objetos.

- ***La fase de prueba***

• Es la escritura del código que va a permitir evaluar el funcionamiento correcto del código escrito en la etapa de implementación. Éste debe ejecutarse programáticamente para poder repetir los resultados las veces que fuera necesario.

- ***La fase de Implementación***

Consiste en poner a disposición del cliente el producto.

- ***Clases***

• Son representaciones simbólicas de objetos; describen las propiedades, campos, métodos y eventos que forman objetos

• Permiten agrupar elementos relacionados como una unidad, así como controlar su visibilidad y accesibilidad en otros procedimientos. Las clases también pueden heredar y reutilizar código definido en otras clases. Se comunican entre sí mediante mensajes.

• Una clase contiene información sobre cuál debe ser la apariencia y el comportamiento de un objeto, describen su estructura.

• Los objetos son instancias de clases que se pueden utilizar.; la acción de crear un objeto se denomina "instanciación".

• Con la analogía de plano, una clase es un plano y un objeto es un edificio construido a partir de ese plano.

(clases y objetos en el mundo analógico Clases y objetos en el mundo digital)

- ***Creación de una clase***

• Crear un Proyecto de librería de clases: cuando se tiene un diagrama de clases con la lógica de negocio este proyecto se denomina de entidades. En el mismo van todas las clases concebidas en el diagrama.

• Se crea un archivo por clase. El nombre del archivo debe coincidir con el nombre de la clase más la extensión ".vb"

• Para declarar la clase, se usa la palabra reservada "Class" y con el modificador "public" para hacerlo visible a toda la solución, este debe ser el primer comando del archivo.

• Sintaxis:

Public Class <nombre_de_clase>

<código>

End Class

Constructor

• *El constructor permite inicializar los campos.*

• *Sintaxis:*

Public Sub New()

<código del constructor>

End Sub

• *New es un procedimiento del tipo constructor que se ejecuta automáticamente cuando una clase es instanciada.*

<Objecto> = New <Clase>()

Prueba de las clases

• *Para poder probar una clase deben realizarse instancias de la misma.*

• *Las instancias para pruebas se construyen en módulos ejecutables.*

• *Para esto se crea un proyecto de prueba de tipo aplicación de consola.*

• *Es preferible que todo el código de prueba se ejecute sin interacción del usuario.*

Pasos iniciales para la prueba

• *Establecer el namespace del proyectos de entidades (de forma predeterminada, es el nombre original que recibió el proyecto).*

• *Agregar un proyecto de test basado en "Aplication console".*

• *Establecer el proyecto de test como Startup Project (proyecto de inicio)*

• *Agregar la referencia en el proyecto de prueba al proyecto de entidades.*

• *Crear un módulo que tenga un Sub Main donde se instanciará la clase a probar.*

• *En cada módulo, incluir la claúsula Import como primera línea del módulo referenciando al namespace de las entidades.*

Instanciar e inicializar la clase

• *Para instanciar la clase se debe crear un objeto basado en ella*

<modificador> <objeto> as <clase>

• *Luego se la debe inicializar*

<objeto> = New <clase>

• *Ejemplo*

Dim oPepe as Persona

oPepe = New Persona

Los objetos deberían ser construidos según los principios de:

- ***Abstracción:*** *se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador. Es el proceso por el cual obtenemos, de una entidad real, sus características fundamentales en una situación particular, planteada por el sistema.*

• Las características de un objeto son el estado (atributos) y el comportamiento del mismo (operaciones).

• Esto nos permite modelar en código y proporcionar su funcionalidad y comportamiento.

• En VB .Net los atributos se almacenan en campos y propiedades y las operaciones se realizan mediante métodos y eventos.

(Ejemplo) Un alumno de una institución, tiene datos (nombre, fecha de nacimiento, documento, un legajo, estatura corporal, notas de exámenes, ahorros bancarios) y además realiza actividades como estudiar, asistir a clases, transacciones bancarias, rendir exámenes y hacer deportes.

• Aplicando la abstracción una entidad alumno sería:

Atributos *Nombre y apellido – Documento - Legajo*

Operaciones *Asistir a clases - Rendir examen - Informes de asistencia - Informes de notas*

La abstracción y las etapas de un proyecto de software

► Captura de requerimientos: Un banco necesita información de las cuentas que tienen sus clientes, de los depósitos y extracciones que realizan en ellas y el estado de cuenta.

►Análisis: cada cliente puede tener dos tipos de cuentas: caja de ahorro y cuenta corriente; puede hacer depósitos, extracciones y ver el balance.

• Una caja de ahorro gana intereses mensuales. El banco le permite al cliente ingresar dinero en la misma y se incrementará mensualmente, según un interés dado.

• Las cuentas corrientes permiten al cliente hacer depósitos y extracciones. El banco le permitirá sobregirar la cuenta hasta un determinado importe.

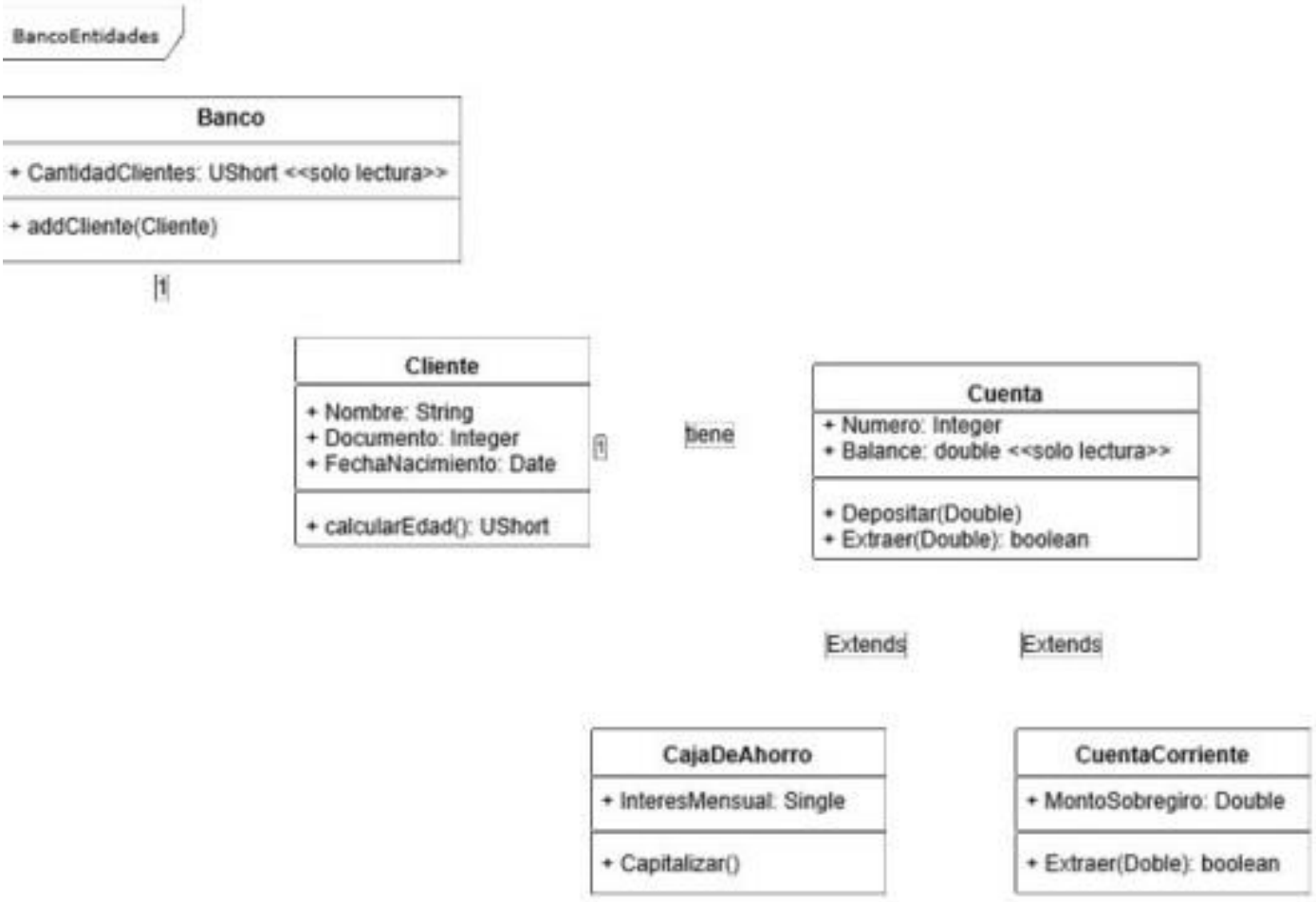
► Diseño: Se tienen los siguientes objetos: banco, cliente y dos tipos de cuentas.

• Un banco es una agregación de múltiples clientes.

• *Las cuentas están asociadas a clientes.*

• *Las cuentas pueden dividirse en dos tipos: cuenta corriente y caja de ahorro.*

Ejemplo de diagrama de clases:



- ***Encapsulamiento:*** *Es el proceso en el cual un componente de software se comporta como una caja negra, exponiendo como público solo lo necesario y ocultando la complejidad del funcionamiento interno al exterior.*

• *Es la capacidad de contener y controlar el acceso a un grupo de elementos asociados.*
• *Oculta las implementaciones privadas detrás de las interfaces públicas.*
• *De esta manera vemos lo que el objeto puede hacer y no como lo hace, es decir que su funcionalidad esta encapsulada en sí mismo.*

Ventajas
• *Protege la modificación de los datos internos directamente.*
• *Da la posibilidad de incluir controles de validación y manejo de errores.*
• *El usuario del objeto debe preocuparse de saber qué hace el objeto y no como lo hace. Cómo funcionan las propiedades y métodos quedan ocultos en el objeto, esto crea una abstracción a la complejidad del mismo. Solo vemos que operaciones se pueden realizar.*

Implementación

• Al crear una clase, se establecen propiedades y métodos para acceder a sus valores internos,

• Las propiedades van a responder a un solicitante externo para devolver o establecer un valor (Getter o Setter)

• Los métodos pueden realizar una operación que afecten algún valor interno de la clase o devuelva un valor basado en valores internos de la misma.

• Los valores internos son privados y las propiedades y métodos son públicos.

Miembros de la clase

• Son los elementos que componen una clase.

• Su alcance puede ser definido mediante un modificador pero por convención se escriben con un modificador específico

• Campos (privados)

• Propiedades

• Métodos

• Eventos

Campos

• Variables internas de la clase que mantiene valores del objeto durante su ejecución y se declaran como privados.

• Declarando los campos como privados se asegura la encapsulación de la clase pues estos serán únicamente accesible mediante las propiedades u otros métodos y no son visibles desde fuera del objeto.

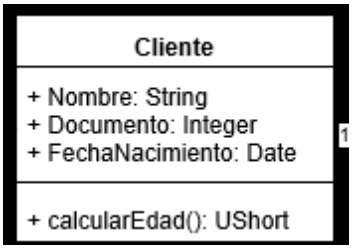
• Los campos se declaran en el encabezado de la clase y por convención llevan un prefijo al nombre de la propiedad que almacenan.

• Sintaxis:

Private <prefijo><nombre del campo> as <Tipo de dato>

• Ej.:

```
Private _Nombre as String
Private _Documento as Integer
Private _FechaNacimiento as Date
```



Propiedades

• Forman parte de la interfaz pública de la clase.

• Son accesos a los campos. Se escribe una propiedad por campo para poder escribirlo o leerlo.

• La propiedad típicamente es del mismo tipo de dato que el campo correspondiente.

• Pueden incorporar validaciones y controles de errores.

• Se crean las propiedades con el modificador "Public" y la palabra reservada "Property"
• Mediante los descriptores de acceso "Get" y "Set" se pueden obtener y establecer los valores de los campos.
• Si tienen getter ► lectura y si tienen setter ►escritura.
Sintaxis:

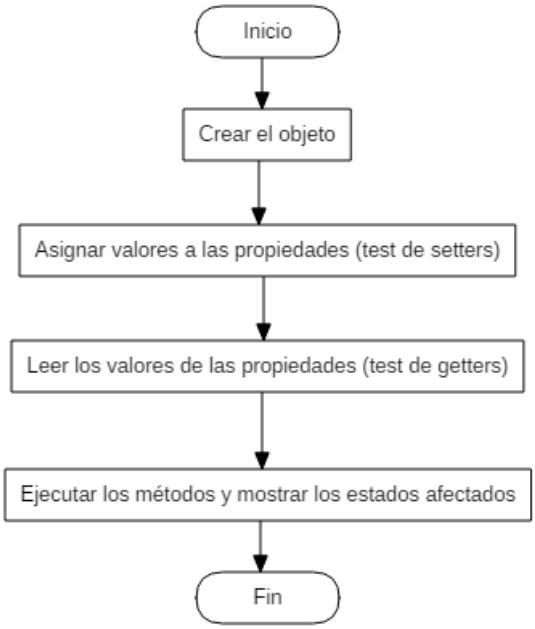
```
Public [ReadOnly|WriteOnly] Property <nombre_metodo> As <tipo_de_dato>
        [GET
        [<código>]
        Return <nombre_campo>
    End Get]
    [Set(ByVal <Value> As <tipo_de_dato>)
       [<código>]
       <nombre_campo> = <Value>
    EndSet]
End Property
```

Ejemplo:

```
Public Property Nombre As String
    GET
       Return _nombre
    End Get
    Set(ByValValue As String)
       _nombre = Value
    EndSet
End Property
```

Prueba de las clases
• Proyecto de prueba (aplicación de consola): Módulos de prueba que crean instancias de las clases y ejecutan sus métodos.
• Se hace un módulo de prueba por clase (test unitario).
• Se establecen valores a sus propiedades (setter)
• Se leen las propiedades (getter)
• Se ejecuta los métodos para probar el comportamiento.
• Si algún método afecta el estado se vuelve a leer la propiedad correspondiente.

Test unitario de una clase:



:

Probar las propiedades clase

• *En un módulo establecer un valor:*

Objeto.propiedad = <valor>

• *Obtener un valor:*

<objeto> = objeto.propiedad

<función>(objeto.propiedad)

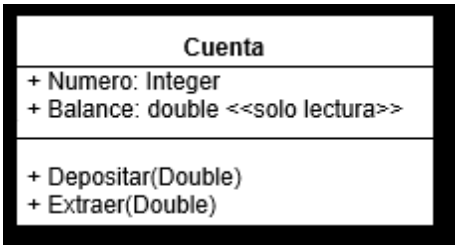
• *Para probar en consola:*

Console.WriteLine(objeto.propiedad)

• *Importante: Todas las propiedades debes ser probadas mediante la asignación de un valor y la obtención de un valor, excepto cuando no corresponda por ser de solo lectura o solo escritura.*

Si un atributo es un resultado o no puede cambiarse se usa la palabra clave "ReadOnly" al declarar la propiedad. Ejemplo:

```
Public ReadOnly Property Balance as double
    Get
        Return _balance
    End Get
End Property
```



Métodos

• *Son procedimientos o funciones asociadas a la clase.*

• *Se utilizan procedimientos cuando no devuelven valores, y cuando sí lo hacen, funciones.*
• *Pueden ser públicos o privados según la necesidad de que estos formen parte de la interfaz Pública de la clase.*

Implementar los métodos de una clase:

Public Sub <nombre>([<parámetros]>)

Public Function <nombre>([<parámetros>]) as <tipo de dato>

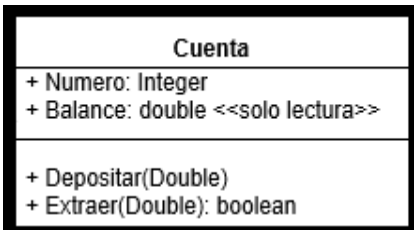
Para invocar se utilizan las siguientes formas

• *Si es un método está basado en una sub:*

<Objeto>.<método>([<parámetros>])

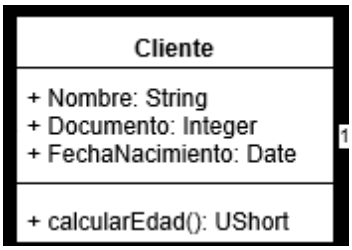
• *Si es un método esta basado en una function:*

<destino> = <Objeto>.<método>([<parámetros>])



Un método internos que no forman parte de la interfaz pública se declara con el modificador "Private"

```
Private Function CalcularEdad(ByVal Fecha As
Date) As Ushort
```



Constructor sobrecargado

• *Permite especificar los valores iniciales (propiedades) del objeto asignándolos como parámetros*.

```
Sub New(<valor1, valor2, …, valorN>)
    propiedad1 = <valor1>
    propiedad2 = <valor2>
    …
    propiedadN = <valorN>
End Sub
```

• *Para instanciar e inicializar*

<objeto> = Objeto.New([<lista_parámetros>])

Método ToString()

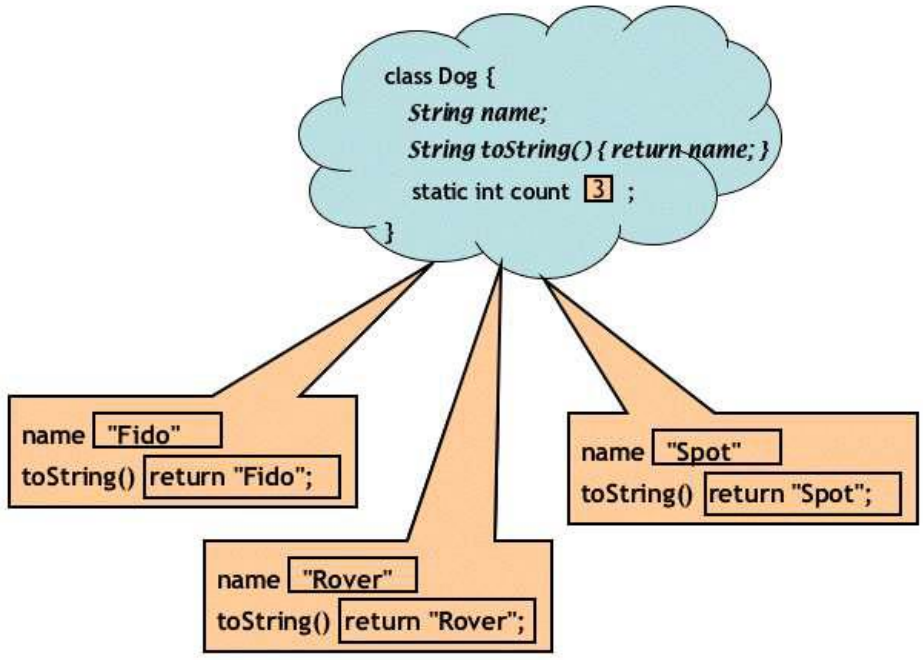
• Devuelve una cadena que representa al objeto actual, una descripción del objeto instanciado.

• Muestra la clase como una cadena de texto, para ello lo mejor es devolver las propiedades más calificativas del objeto.

• ToString() es un método de la clase Object que es heredado implícitamente por todas las clases. Se recomienda sobrescribirla, si no es así muestra un nombre predeterminado del objeto.

• Sintaxis:

```
Public Overrides Function
        ToString() As String
        Return <cadena>
End Function
```



- ***Herencia***

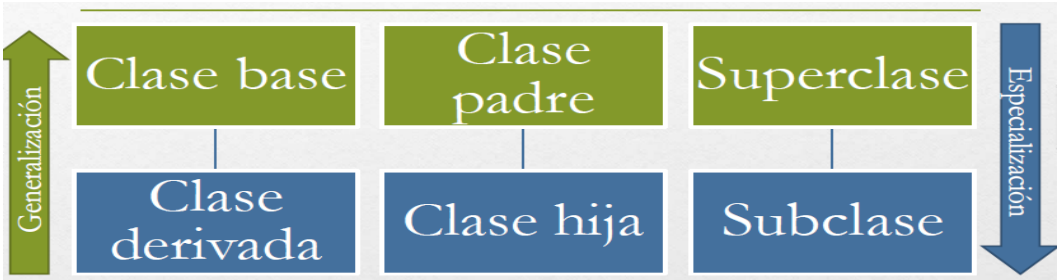
• Es el mecanismo por el cual una clase obtiene toda la funcionalidad de otra denominada clase base.

• Esto permite reutilizar lo existente sin necesidad de escribirlo de nuevo.

• Permite que los programas sean más compactos y extensibles.

• Una clase puede heredar de otra su apariencia y comportamiento y también podrá agregar sus propios.

Ejemplo • De una clase base se crean subclases con otros atributos u otros comportamientos.



En Visual Basic solo se puede heredar de una clase a la vez, pero permite la implementación de múltiples interfaces.

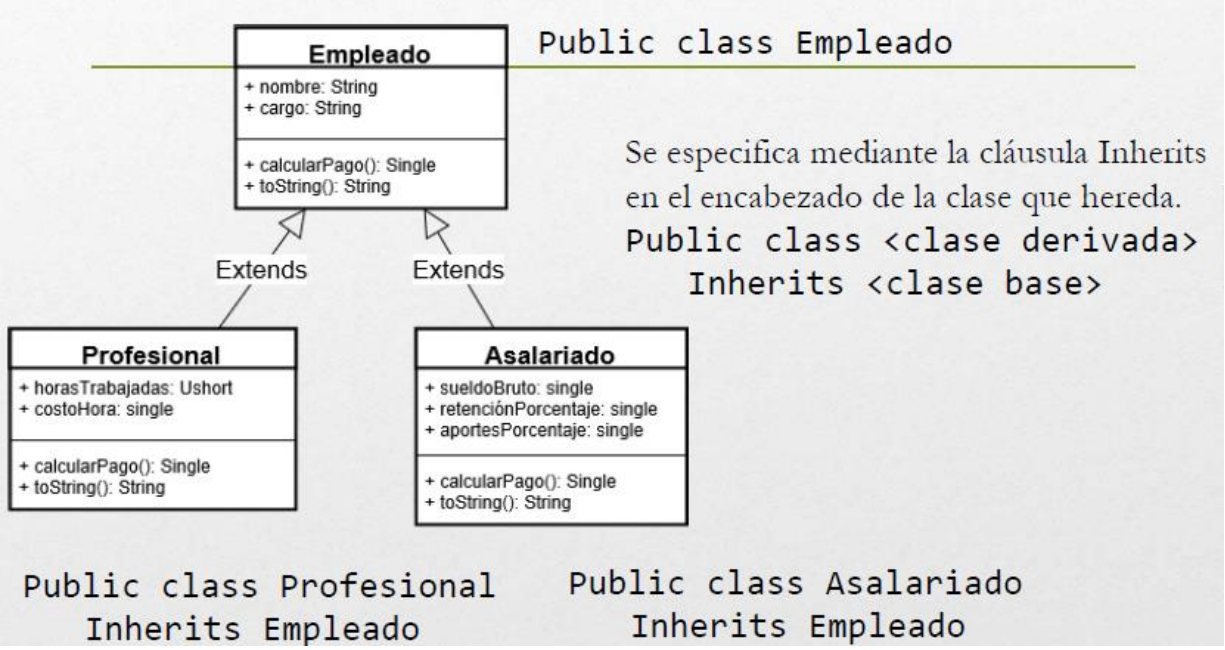
• Si no se especifica una clase padre, implícitamente se hereda de la clase Object; esta posee algunos métodos comunes a todas las clases del framework.

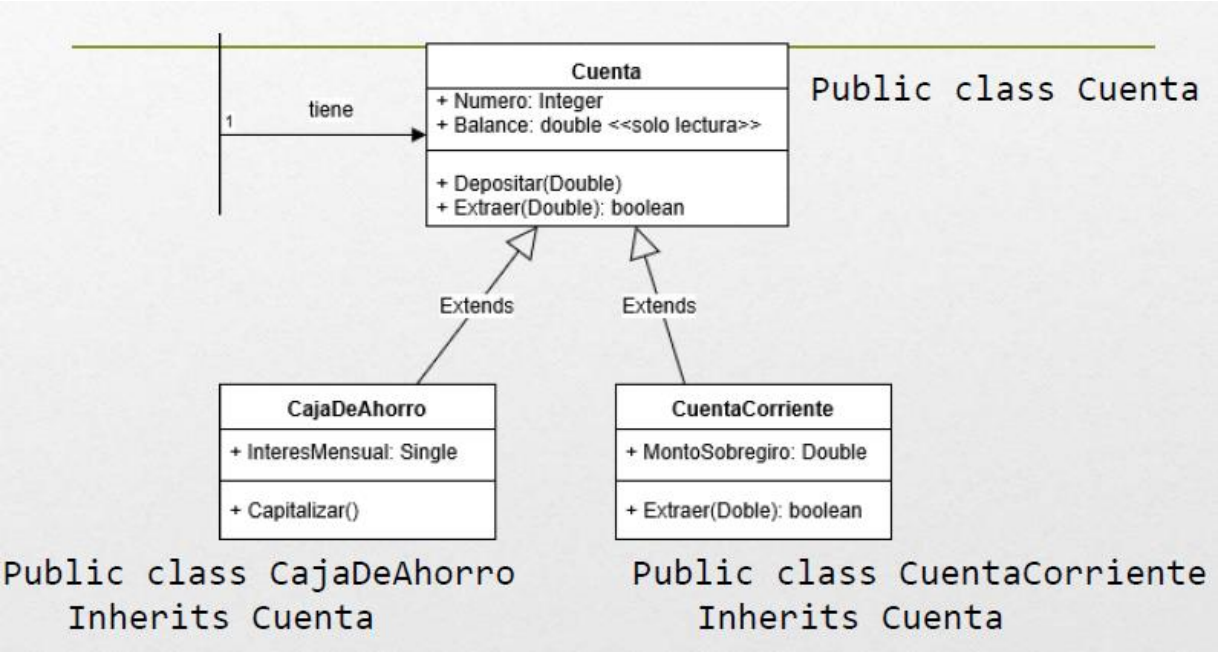
• La subclase tendrá los mismos atributos y métodos que la clase base, en otras palabras, permite reutilizar todas las propiedades y métodos de la clase base.

• No se heredan los constructores por lo que hay tener especial atención en su implementación.

• Tampoco se pueden acceder a los miembros privados de la clase base.

Implementación de la Herencia





Acceso a la clase base

- *La palabra reservada MyBase permite acceder a los miembros de la clase base.*
- *Se usa para aprovechar la implementación existente de la clase base y adaptar a la subclase*

  MyBase.<propiedad | método>
- *El método constructor no se hereda.*
- *Ej. Permite acceder al constructor de la clase base:*

  myBase.new()

- ***Polimorfismo** quiere decir "muchas cosas".*

• Es la habilidad de mostrarse o comportarse de manera diferente según el contexto del sistema.

• Es la posibilidad que tienen las propiedades y métodos de mantener una respuesta unificada, con la misma semántica, aunque con distinta implementación.

• La herencia y las interfaces son la base del polimorfismo.

Polimorfismo basado en herencia

• Varios objetos que parten de una misma clase, pueden tener propiedades y métodos con el mismo nombre pero pueden actuar de forma diferente.

• El método o la propiedad heredada puede ser sobrescrita.

• Es posible cambiar el comportamiento de la clase base, en la subclase, mediante la implementación del polimorfismo.

Miembros virtuales

*• Un miembro de la clase base, puede ser **redefinido** en las clases derivadas de ésta y puede ser accedido mediante una referencia a la clase base, resolviéndose la llamada en función del tipo del objeto referenciado.*

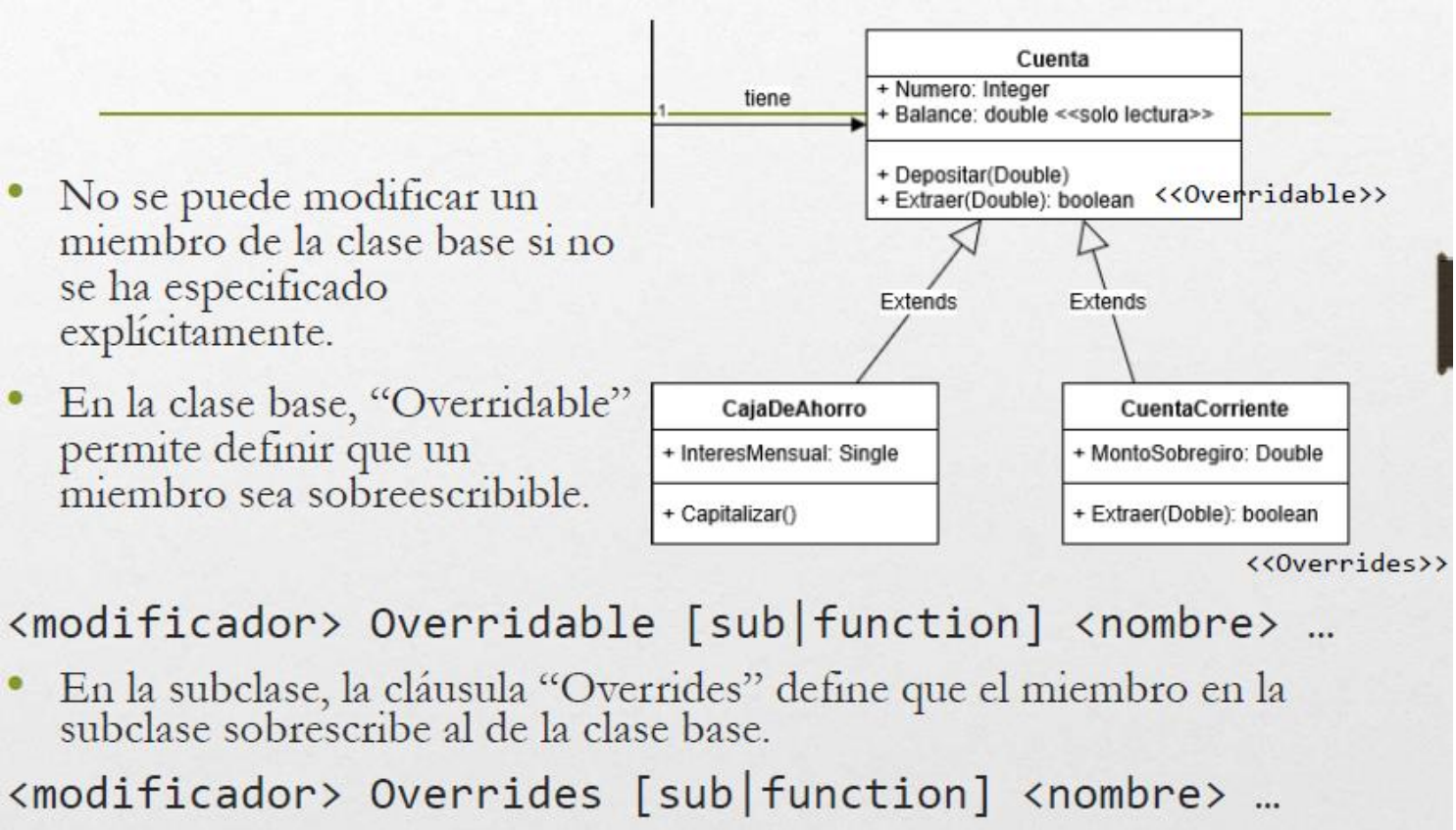
• Se implementa declarando con Overridable en la clase base y Overrides en la subclase.

<modificador> Overridable [miembro] <nombre>

…

<modificador> Overrides [miembro] <nombre>

Redefinición



- No se puede modificar un miembro de la clase base si no se ha especificado explícitamente.
- En la clase base, "Overridable" permite definir que un miembro sea sobreescribible.

```
<modificador> Overridable [sub|function] <nombre> …
```

- En la subclase, la cláusula "Overrides" define que el miembro en la subclase sobrescribe al de la clase base.

```
<modificador> Overrides [sub|function] <nombre> …
```

Herencia y constructores

• Los constructores no se heredan.

• Cuando crea un objeto de una subclase se ejecutan los de todas las clases base. Para ello, se ejecuta en primer lugar el constructor de la clase, que ocupa el nivel más alto en la jerarquía de herencia y se continúa de forma ordenada con el resto de las subclases.

Herencia y constructores sobrecargados

• Mientras no se especifique explícitamente, en la clase base se ejecutará siempre el constructor sin parámetros.

• Para ejecutar un constructor sobrecargado hay que invocarlo explícitamente.

• La invocación de un constructor a otro constructor se debe hacer en la primer línea del mismo.

Clases abstractas

• Son como cualquier clase pero estas no se pueden instanciar, esto obliga a que sólo sean utilizadas mediante la herencia.

• Se declaran con la clausula MustInherit

• Por convención una clase abstracta tiene el sufijo "Base" junto al nombre de la clase.

```
Public MustInherit Class <clase_base>Base
```

Acceso protegido

• *Modificadores conocidos:*

Dim

Private -

Friend ~

Public +

Protected #

• *Establece la visibilidad en la clase actual y en las que heredan.*

• *Permite que desde la subclase se pueda acceder a miembros de la clase base sin ser accesibles por otras clases que no sean de la jerarquía.*

Protected <propiedad | método>…

Miembros abstractos

• *Proporciona operaciones que deben ser definidas en la clase derivada, con la intención de adaptarla a las particularidades de ésta.*

• *Es un miembro calificado con MustOverride y tiene la particularidad que no tener cuerpo.*

• *Es implícitamente un método virtual y sólo se puede declarar en clases abstractas.*

[modificador] MustOverride [miembro] <nombre>…

Clases y métodos finales

• *Una clase sellada o final no permite que una clase o un método se pueda heredar.*

• *Una clase final se declara con la palabra NotInherable*

Public NotInheritable Class <nombre>

• *Un método final se declara con la palabra NotOverridable*

<modificador> NotOverridable Function|Sub …

Polimorfismo basado en herencia

• *Se utiliza cuando uno quiere aplicar pequeños cambios en la clase heredada y mantener intacta la clase base.*

• *Una clase de tipo polimórfico tiene que implementar miembros virtuales.*

• *Se aplica instanciando el objeto según la clase base e iniciándola con la subclase:*

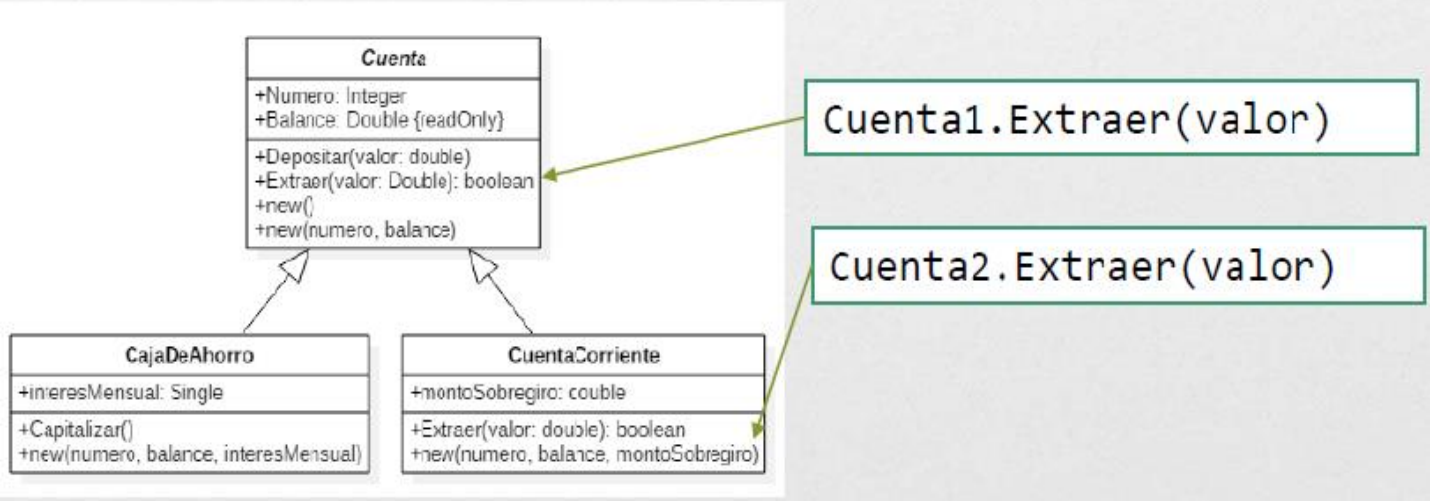
<modificador> <objeto> as <Clase base>

<Objeto> = New <sub clase>()

• *Con esto el compilador retarda hasta el tiempo de ejecución: cual miembro va a utilizar: el de la clase base o la subclase.*

Ejemplo: Si se realiza el cambio de una clase, éste se reflejará en todas las subclases que dependan de ella.

```
Dim cuenta1 as Cuenta
Dim cuenta2 as Cuenta
cuenta1 = New CajaDeAhorro()
cuenta2 = New CuentaCorriente()
```



- ***Asociaciones***

• *Son relaciones entre las clases.*

• *Existen varios tipos de asociaciones que se determinan según la vinculación que tenga una clase con otra.*

• *Una asociación en programación significa que una clase tiene un atributo cuyo tipo es la otra clase.*

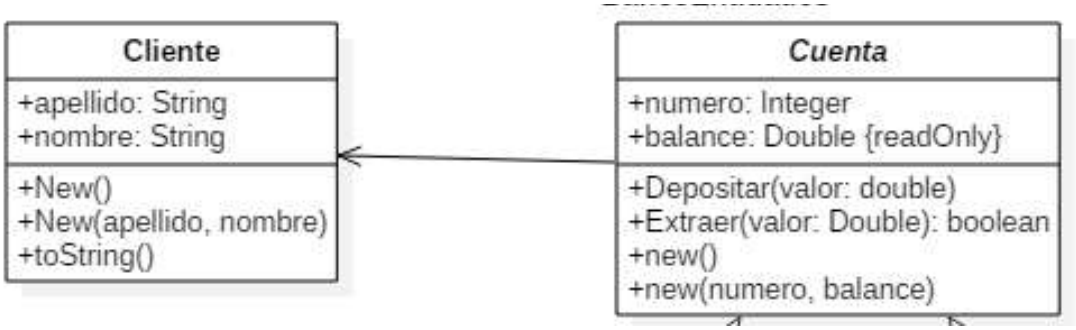
• *Las asociaciones pueden tener* ***cardinalidad****, lo que indica cuantos atributos forman la asociación.*

• *Las asociaciones en UML se representan con flechas que indican el sentido de navegación de la relación.*

Asociación simple

• *Es cuando una instancia de una clase esta asociada únicamente a otra instancia.*

• *En este caso la clase destino es un atributo en la clase origen.*



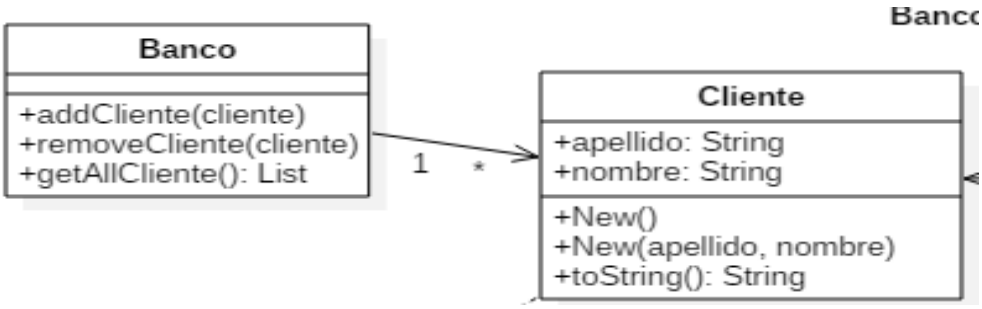
Private <campo> as <Clase>

Public Property <propiedad> as <Clase>

Tanto en el campo, como en la propiedad, se declara el tipo de dato como la clase destino (ésta debe existir previamente)

Asociación múltiple

• *Cuando varias instancias de una clase, están asociadas a otra. La clase destino es una lista de atributos en la clase origen.*



En la clase origen se implementa un campo como una lista del tipo de datos de la clase destino.

Private _<lista> as List(of <clase>)

• *No se implementa propiedad (setter, getter).*

• *Se implementan métodos para agregar, remover, obtener la lista completa y obtener instancias mediante búsquedas por sus atributos.*

• *Se aconseja usar la notación propia de .Net*

• *Agregar:*

Sub add<Clase>(<objeto> as <Clase>)

    _<lista>.add(<objeto>)

• *Remover:*

Sub remove<Clase>(<objeto> as <Clase>)

    _<lista>.remove(<objeto>)

• *Obtener la lista completa:*

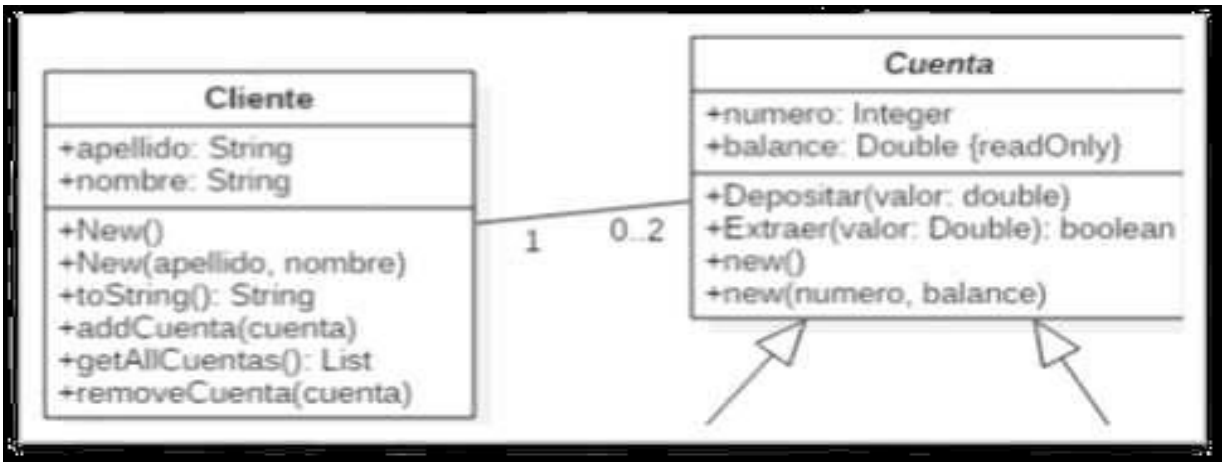
```
Function getAll<Clase>() as List(of <Clase>)

    Return _<lista>
```

Asociación bidireccional

• *La navegabilidad se da en ambas direcciones.*

• *En cada clase existe un atributo que referencia a la otra, con código que asegura la sincronización de ambos extremos.*

• *Una de las clases es responsable por la asignación en ambos lados de la asociación y la otra colabora con una propiedad o método de asignación con visibilidad a nivel de proyecto (Friend).*



Interfaz (Interface)

• *Es un conjunto de operaciones que especifican un servicio.*

• *Sirve para encapsular un conjunto de métodos y propiedades, sin asignar esta funcionalidad a ningún objeto en particular ni expresar nada respecto al código que lo va a implementar.*

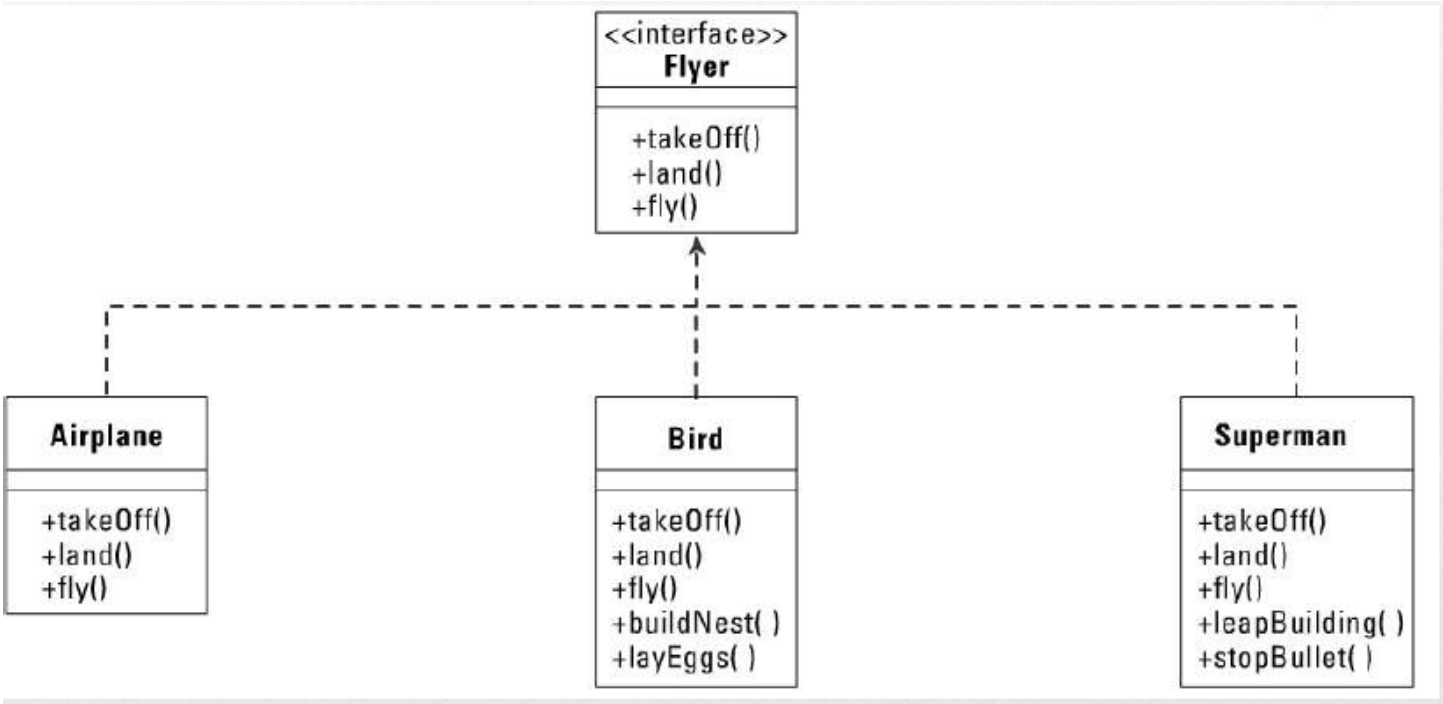
• *Las clases "realizan" la interfaz.*

• *Desde el punto de vista de diseño la interfaz especifica un contrato: Las Interfaces definen las propiedades, métodos y eventos que deben implementar las clases.*

• *Indican qué se debe implementar en la clase y no como hacerlo.*

• *Permite que clases diferentes tengan comportamiento similar o bien una característica común compartida.*

Ejemplo imaginario de interfaz



Uso de las Interfaces

• *Declarar los métodos que se espera que una o más clases implementen.*

• *Mostrar una interfaz de programación de un objeto, sin revelar el cuerpo de la clase.*

• *Capturar la similitud entre clases no relacionadas, sin forzarlas a una relación de clases.*

• *Simular herencia múltiple, declarando una clase que implementa varias interfaces.*

• *Se declaran con la palabra reservada Interface. Ej.:*

```
Public Interface <nombre_interfase>
```

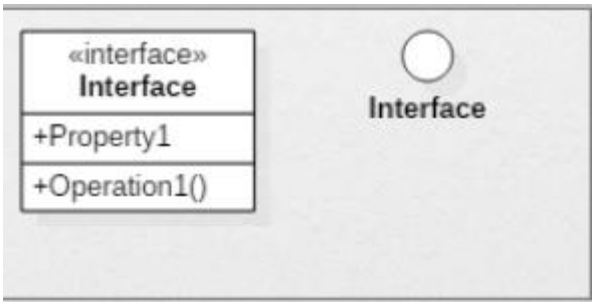
• *El cuerpo únicamente tiene declaraciones. Ej.:*

```
Interface IAsset
    Property Division() As String
    Function GetID() As Integer
End Interface
```

• *Para implementarlas en la clase mediante la palabra reservada*

Implements. Ej.:

```
Public Class <nombre_clase>
    Implements <nombre_clase_base>
```



Interfaz vs Herencia

• *Las interfaces admiten mejor las situaciones en las cuales las aplicaciones necesitan el mayor número de tipos de objetos no relacionados posible para proporcionar determinadas funciones.*

• *Las interfaces son más flexibles que las clases de base, porque puede definir una única implementación que puede implementar interfaces múltiples.*

• *Las interfaces son mejores en situaciones en las que no es necesario heredar una implementación de una clase de base.*

• *Las interfaces son útiles cuando no se puede usar la herencia de clases. Por ejemplo, las estructuras no pueden heredarse de las clases, pero pueden implementar interfaces.*

Miembros estáticos

• *Un miembro estático de la clase (compartido) se comparte entre todas las instancias de una clase.*

• *Sintaxis:*

<Modificador> Shared <nombre_miembro> […]

• *Ejemplo:*

Public Shared SampleString As String = "Sample String“

Public Shared Sub UltimoElemento()

• *Un miembro estático sólo puede acceder a otros miembros estáticos y puede ser accedido por cualquier miembro.*

• *Para acceder desde otra clase debe hacerse desde el nombre de la clase.*

Sintaxis:

<nombre_clase>.<nombre_miembro_estático>

• *Ejemplo:*

SampleClass.SampleString

• *Un constructor estático se ejecuta al crear la primera instancia de la clase.*