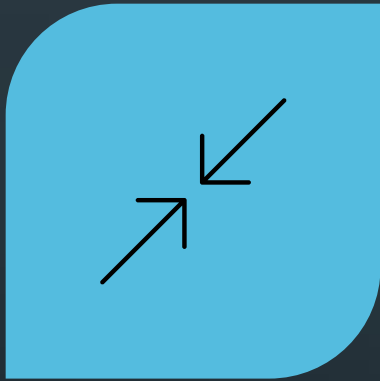


A decorative graphic consisting of thin, dark gray lines that resemble a circuit board. These lines extend horizontally from the left and right edges of a central black rectangle, featuring several small circles at their ends, suggesting connection points or components.

HERENCIA

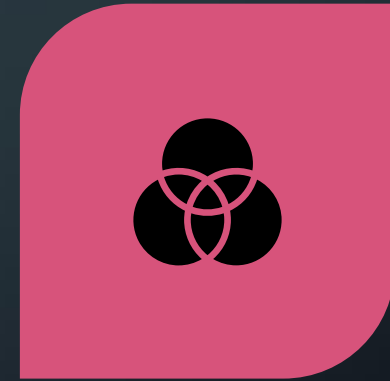
# REPASANDO: INTERFACES




GRACIAS AL USO DE INTERFACES  
PODEMOS IMPLEMENTAR  
COMPORTAMIENTOS SIMILARES EN  
DISTINTAS CLASES.



SI BIEN LUEGO PUEDEN VARIAR,  
DISTINTAS CLASES VAN A TENER UN  
MÉTODO SIMILAR.



ESTE MÉTODO RECIBIRÁ LO MISMO Y  
DEVOLVERÁ LO MISMO.

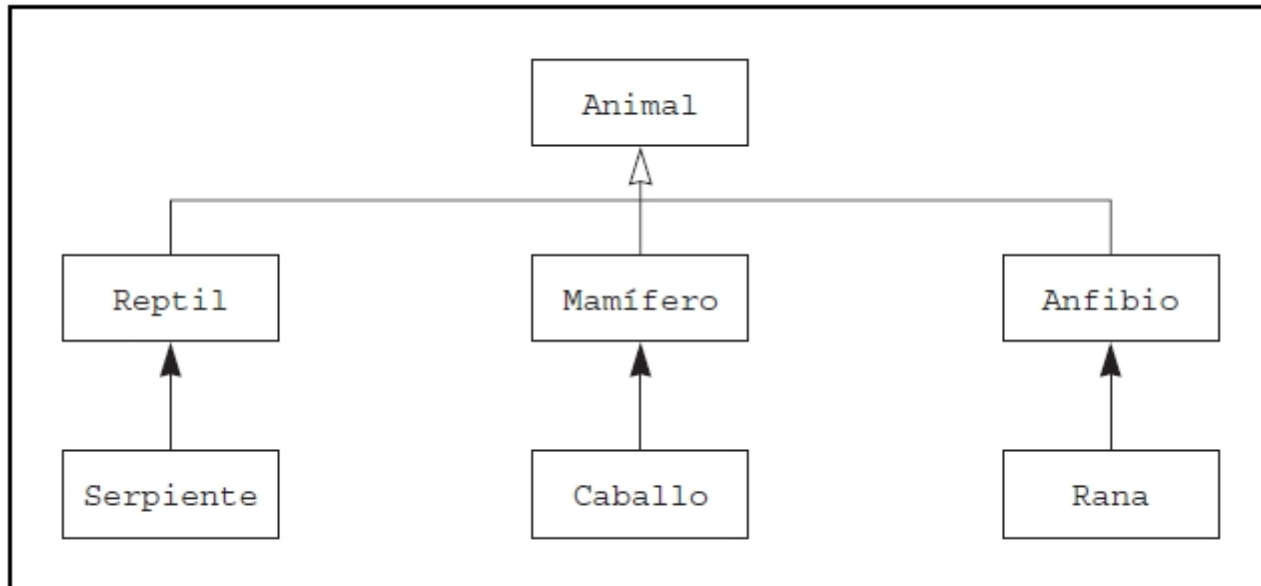


## REPASANDO: EL MODELO DE SISTEMA

Como vimos anteriormente, el paradigma orientado a objetos busca modelizar el dominio de un sistema.

Ahí identificamos objetos que pertenecen a determinada clase.

¿Puede pasar que objetos de distintas clases compartan algunas cosas?

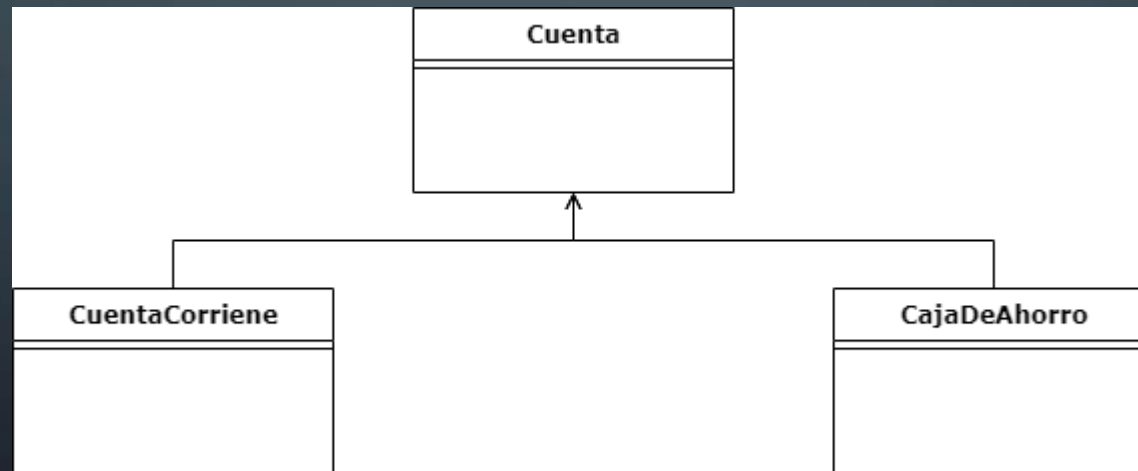


## GENERALIZACIÓN Y ESPECIALIZACIÓN

- Puede pasar que tengamos muchas clases que compartan elementos entre ellas pero que tienen alguna característica que las distingue
- En la vida cotidiana vemos eso como clases y subclases.
- Cuando programamos a eso lo determinamos como especialización y generalización

# CONCEPTO DE HERENCIA

- Es un mecanismo de la programación orientada a objetos.
- Nos permite implementar una jerarquía de especialización-generalización.
- Una clase será el caso general de varios casos especiales.



# UTILIZACIÓN

- Mediante la herencia vamos a poder reutilizar recursos y características de la clase principal (o padre) en las clases hijas.
- Vamos a poder usar comportamientos o atributos de la clase padre.
- También vamos a poder implementar polimorfismo acá.

# DESVENTAJAS

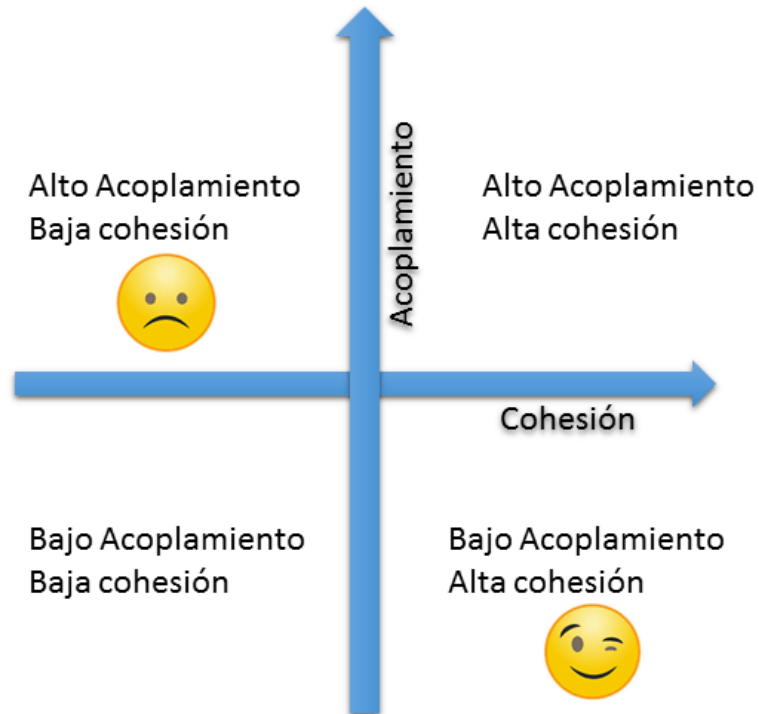
- Aumenta la complejidad de una solución.
- El uso de herencia aumenta el acoplamiento entre clases. Una modificación en la clase padre será propagada a todas las clases hijas.

# CONCEPTO DE COHESIÓN Y ACOPLAMIENTO


- La cohesión es el grado de delimitación y la capacidad de los elementos de una misma clase de trabajar entre sí, con los mismos tipos y una temática común. Siempre apuntamos hacia la alta cohesión.
- El acoplamiento es el grado de dependencia que tiene un elemento de un sistema con respecto a otros. Un sistema con mucho acoplamiento es propenso que al realizar un cambio este impacte de formas no deseadas en otros lugares



# ALTA COHESION VS BAJO ACOPLAMIENTO



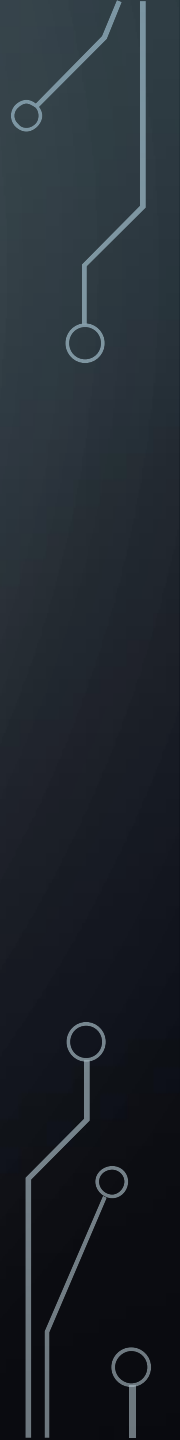
- Siempre tenemos que intentar reducir el acoplamiento.
- ¿Como realizamos esto?
  - Debemos definir limites claros para lo que hace cada clase.
  - Debemos tener módulos que sean lo mas reutilizables posible.



## NATURALEZA DE LA HERENCIA EN C#

Simple: Una clase derivada solo puede tener un único padre. No es el caso para las interfaces donde se pueden implementar varias.

Estrictas: Las clases hijas derivan todo de la clase padre. No quita que algunas cosas se puedan sobrescribir.



The background is a dark blue gradient. In the corners, there are decorative white line art elements resembling circuit boards or neural networks, with lines and small circles.

# TIPOS DE HERENCIA

# PRIMER CASO DE HERENCIA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace App_bancariaBien
{
    class CajaAhorro:Cuenta
    {
    }
}
```

- En este caso la clase heredada será una copia exacta de la clase padre.
- Tendrá los mismos métodos y atributos.
- Lo que no tendrán heredados son los accesos a los atributos y métodos.

## SEGUNDO CASO: HERENCIA CON AGREGADO

- Además de obtener todo lo de la clase padre, se agregan a la clase padre más elementos (atributos y métodos).

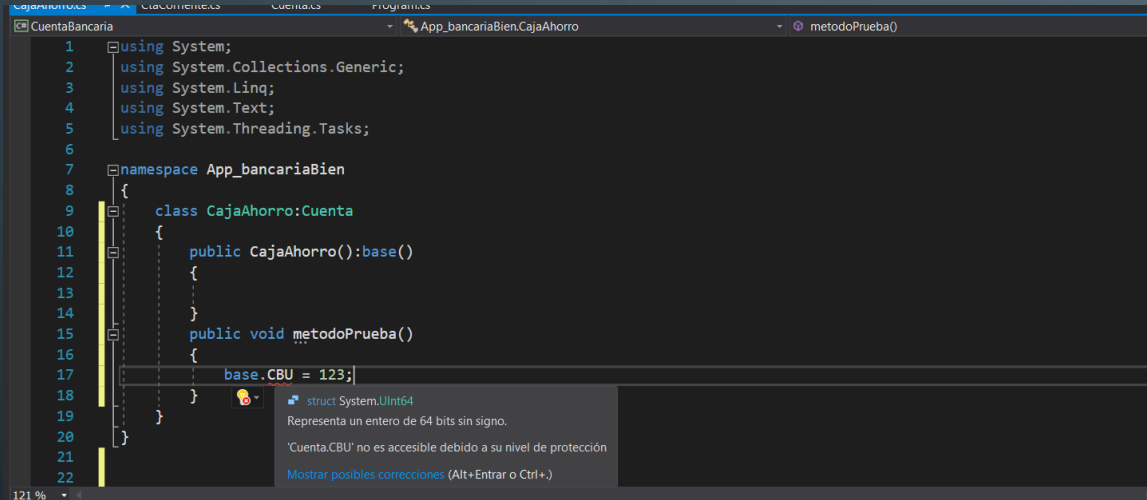
# TERCER CASO: HERENCIA CON REDEFINICIÓN

- En estos casos, se toman elementos de la clase padre o clase base y se redefinen.
- Esto es común para redefinir comportamientos o métodos de clase o de instancia.
- Acá vamos a tener dos tipos de métodos:
  - Shadowing
  - Overriding: Para sobrescribir un método primero debemos definir el método de la clase base como virtual.

# ACCESOS EN CLASES DERIVADAS

- Heredar atributos no significa que se hereden los accesos.
- Desde una clase derivada no se podrán acceder a los atributos o métodos privados.
- Para poder acceder debemos usar los métodos getters y setters correspondientes.

# ACCESOS EN CLASES DERIVADAS



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace App_bancariaBien
8 {
9     class CajaAhorro:Cuenta
10     {
11         public CajaAhorro():base()
12         {
13         }
14
15         public void metodoPrueba()
16         {
17             base.CBU = 123;
18         }
19     }
20 }
21
22
```

struct System.UInt64  
Representa un entero de 64 bits sin signo.  
'Cuenta.CBU' no es accesible debido a su nivel de protección  
[Mostrar posibles correcciones](#) (Alt+Entrar o Ctrl+.)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace App_bancariaBien
{
    class CajaAhorro:Cuenta
    {
        public CajaAhorro():base()
        {
        }

        public void metodoPrueba()
        {
            base.setCBU(123);
        }
    }
}
```



# ACCESO – ATRIBUTOS PROPIOS

- Cuando desde la clase hija queremos acceder a un elemento de instancia propio usamos `this`.
- Cuando desde la clase hija queremos acceder a un elemento de clase propio usamos el nombre de la clase hija.

# REFERENCIACIÓN – ATRIBUTOS CLASE PADRE

- Cuando desde la clase hija queremos hacer referencia a un elemento de instancia de la clase padre usamos `base`.
- Cuando desde la clase hija queremos hacer referencia a un elemento de clase usamos el nombre de la clase padre.

# ACLARACIONES

- Referenciar no es lo mismo que acceder. A los atributos de la clase padre por lo general les hacemos referencia (a menos que se cambie el modificador de ámbito).
- Salvo que se haga sobreescritura o se redefinan, el comportamiento heredado de la clase hija es idéntico al de la clase padre.

# EJEMPLOS

```
class CajaAhorro:Cuenta
{
    static float interesRetorno;
    string planCuenta;
    ulong tarjetaVinculada;
    public static void setInteres(float interes)
    {
        CajaAhorro.interesRetorno = interes;
    }
    public CajaAhorro():base()
    {
        this.planCuenta = "basico";
        this.tarjetaVinculada = 0000;
    }
    public string imprimirDatosCliente()
    {
        return base.getCBU().ToString() + " " + base.getCliente().ToString() + " " + this.planCuenta;
    }
}
```

# CLASES ABSTRACTAS

- Estas son clases que no pueden instanciarse.
- Los miembros solo sirven para que se puedan reutilizar a través de la herencia.
- Pueden contener métodos de clase los cuales se pueden invocar.
- Pueden contener constructores.
- Se declaran con `abstract`

# CLASES SELLADAS O FINALES

- Estas clases no se pueden heredar
- Si se pueden instanciar.
- Se declaran con sealed

# CLASES ESTÁTICAS

- No pueden heredarse
- No pueden instanciarse
- Su única funcionalidad es contener miembros de clase (atributos o métodos) que pueden ser invocados.
- Se declaran con `static`.

# CLASES PARCIALES

- Son clases que se definen o completan en dos o mas archivos .cs
- Esto me permite trabajar una parte de la clase en un archivo y otra parte en otro.
- Me facilita la organización en clases muy extensas o en caso de estar trabajando en equipos.



# FORMULARIO HERENCIA:

- <https://forms.gle/jnW1XQhRDtQ4nVkS6>