



PROGRAMACIÓN II

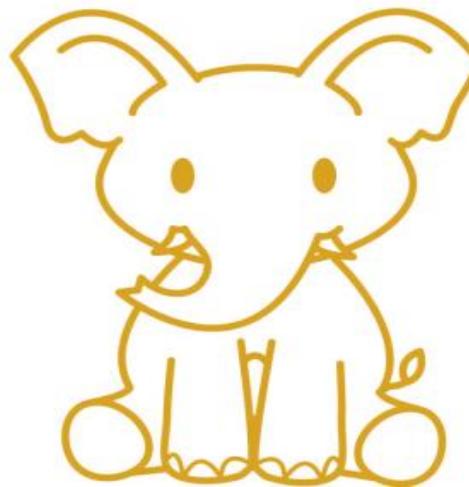
2 PROGRAMACIÓN ORIENTADA A OBJETOS

Patricio Michael Paccha Angamarca
Magister en ingeniería de software



AULA VIRTUAL

PROHIBIDO LOS ...



TROMPUDOS



BRAVOS



< x%!.../>





- Definición de objeto, clases, atributos y métodos, encapsulamiento, abstracción.
- Definición e implementación en JAVA de herencia, clase abstracta, métodos estáticos, métodos abstractos.
- Uso de punteros estáticos y dinámicos en C++ para el manejo de memoria estática y dinámica. Utilizar estructuras de datos unidimensionales o bidimensionales.



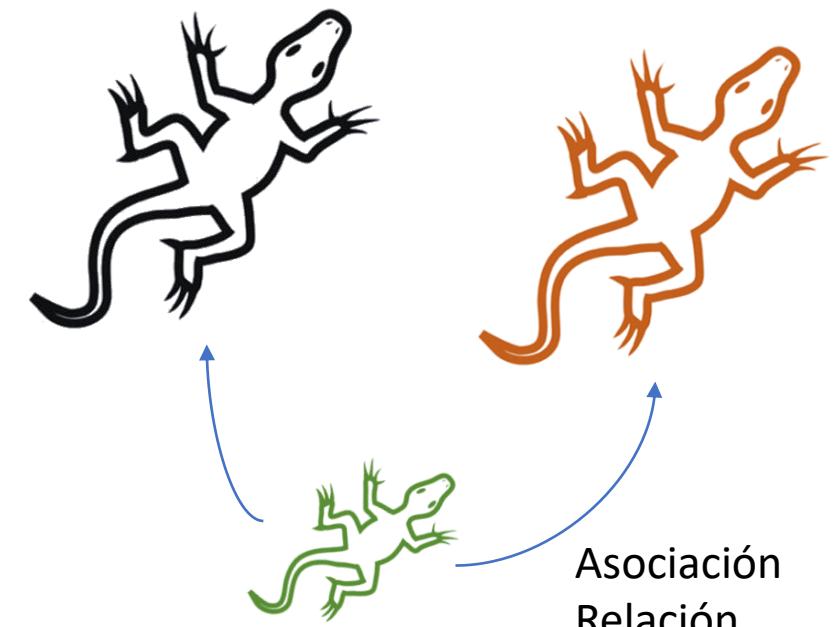


PROGRAMACIÓN ORIENTADA A OBJETOS - P.O.O

PROGRAMACIÓN ESTRUCTURADA

- Procedimientos → void ABC(<params>) { code }
- Funciones → <TipoDato> ABC(<params>) { code ...; return <TipoDato> }
- Variables globales y locales
- Librerías → *.h

```
main ()  
{  
    addLagartija(); → struct lagartija [int vidaMeses, int dedos, string nombre, .... ]  
    struct lagartija [int vidaMeses, int dedos, string nombre, .... ]  
  
    restar();  
    ...  
}
```



Asociación
Relación
Herencia
Dependencia

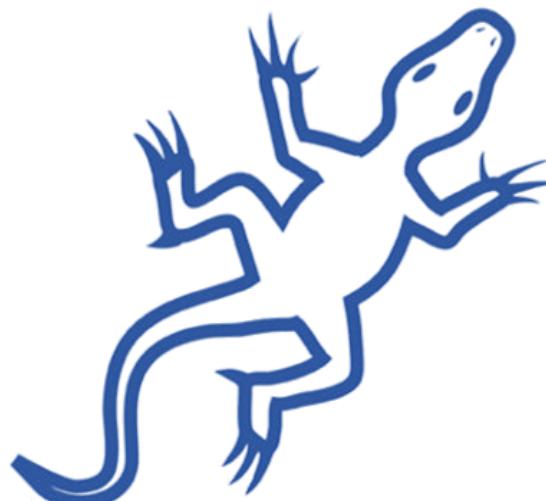
...
Mensaje





PROGRAMACIÓN ESTRUCTURADA

- Procedimientos → void ABC(<params>) { code }
- Funciones → <TipoDato> ABC(<params>) { code ...; return <TipoDato> }
- Variables globales y locales
- Struct
- Librerías → *.h
- evento



REPTILES

PROGRAMACIÓN 0.0.

- Métodos → <**ámbito**> void ABC(<params>) { code }
- Propiedades, atributos → <**ámbito**> Variables globales y locales
- Clases
- Objetos
- Encapsulamiento
- Abstracción
- Especificaciones
- Generalización
- Evento - mensaje
- ...
- Librerías, Paquetes

Ámbito:
+ publico
- Privado
~ protegido



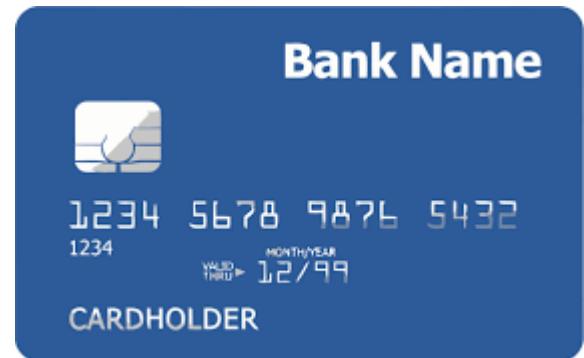
Reptil

Características sustantivos

Nombre
Tamaño
Color

Acciones verbo

Comer(..)
Correr(..)
Reproducirse(...)



CreditCard

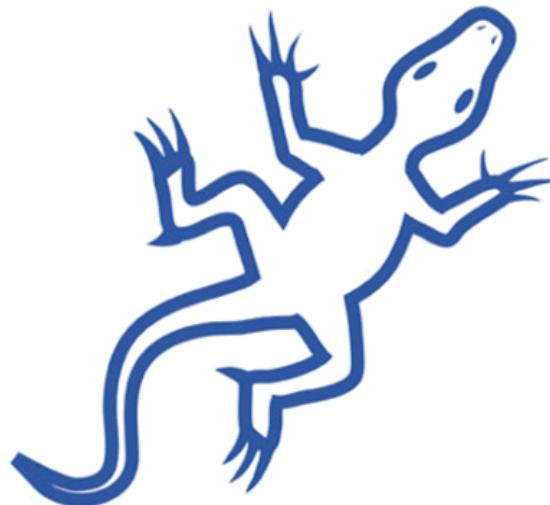
Nombre del Banco
Número de la tarjeta
Nombre propietario
Color
Fecha de caducidad
Tamaño
CVC
...

Características sustantivos

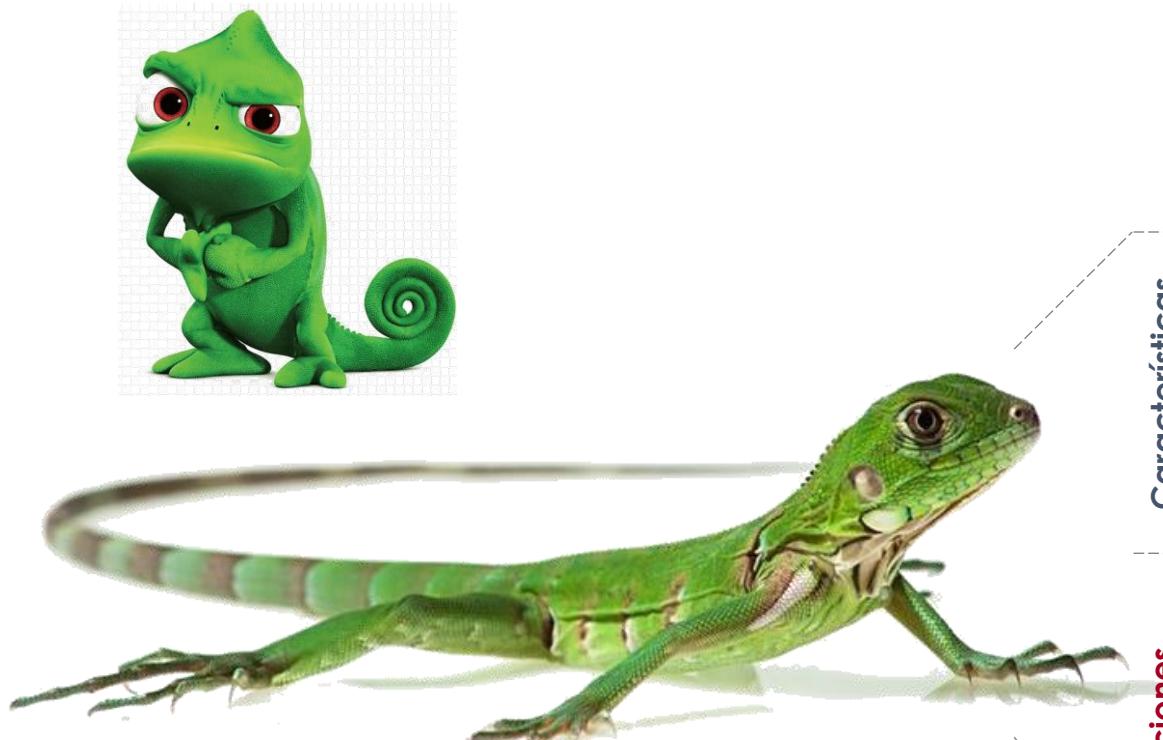
Pagar(\$monto)
Retirar Dinero (\$ monto)
Comprar (\$monto)

Acciones verbo

préstamo - RetirarDinero
validarMonto()
asignar un número único
...



REPTILES



Características

Reptil

Nombre

Tamaño

Color

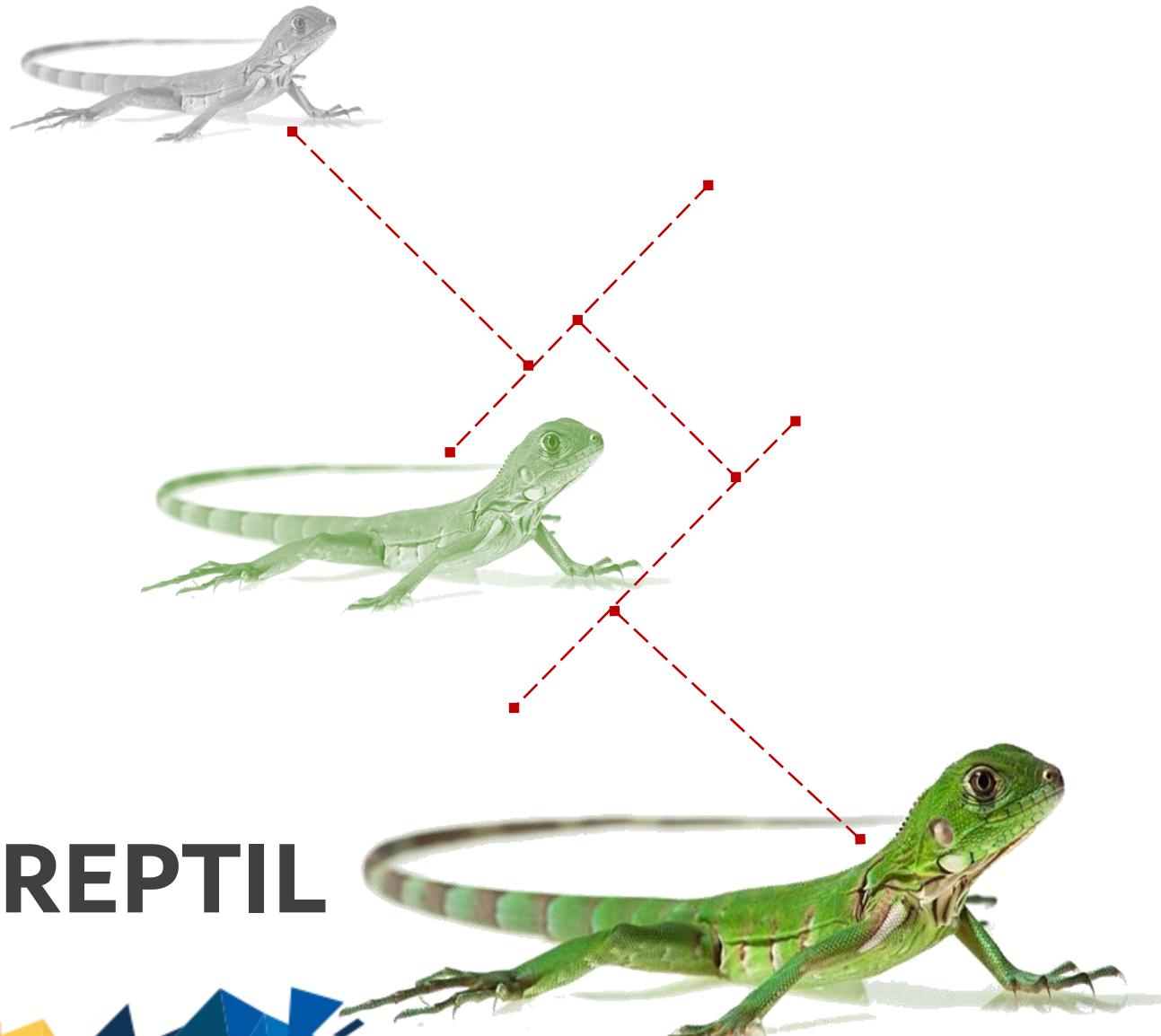
Acciones

Comer

Correr

Reproducirse





CLASE DEMOSTRATIVA :

| |
|---|
| <p>«Class» Reptil</p> |
| <ul style="list-style-type: none">- Nombre : string- Tamano : double- Color : string- Peso : double- Taxonomia : Class- ADN: Class |
| <p>#Comer(Alimento : String, Cantidad: int): bool #Comer(Alimento : TipoAlimentoClass , Cantidad: int): b +AbstractDormir()</p> |
| <p>Responsibilities</p> <ul style="list-style-type: none">– Resp1– Resp2 |

DEMOSTRACIÓN

#HágalePues!



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE
ESCUELA POLITÉCNICA NACIONAL



#ComoTas!

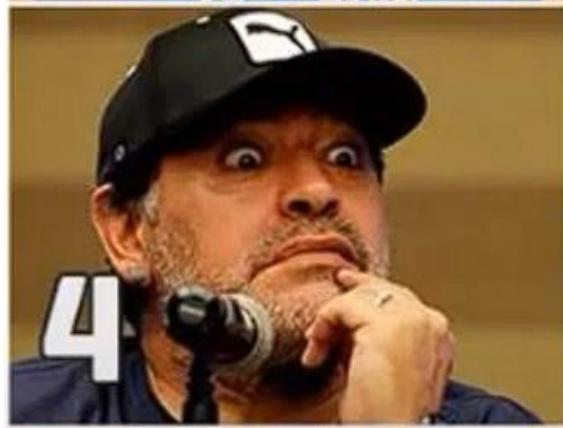




Diagrama de caso de uso

- Identifica los componentes principales que forma al sistema
- Captura los requerimientos fundamentales del sistema
- Actor
 - Interactúa con el sistema para poder alcanzar un objetivo
 - Cada es un caso de uso
 - Puede ser una persona u otro sistema
 - Un rol
- Nos provee una vista de alto nivel
- Indica que actores están asociados al caso de uso
- Las asociaciones se colocan con flechas
- La dependencia indica que es necesaria una relación entre casos de uso

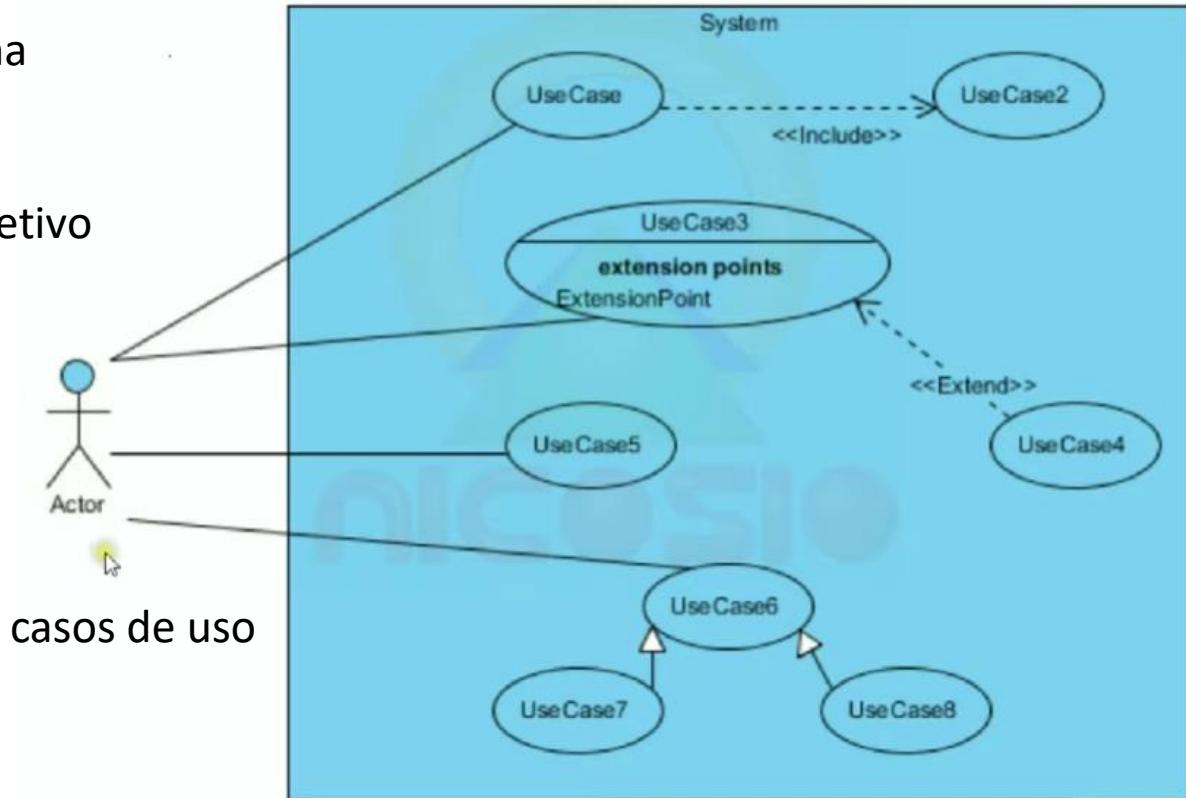
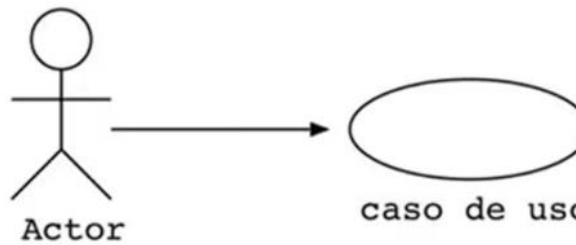
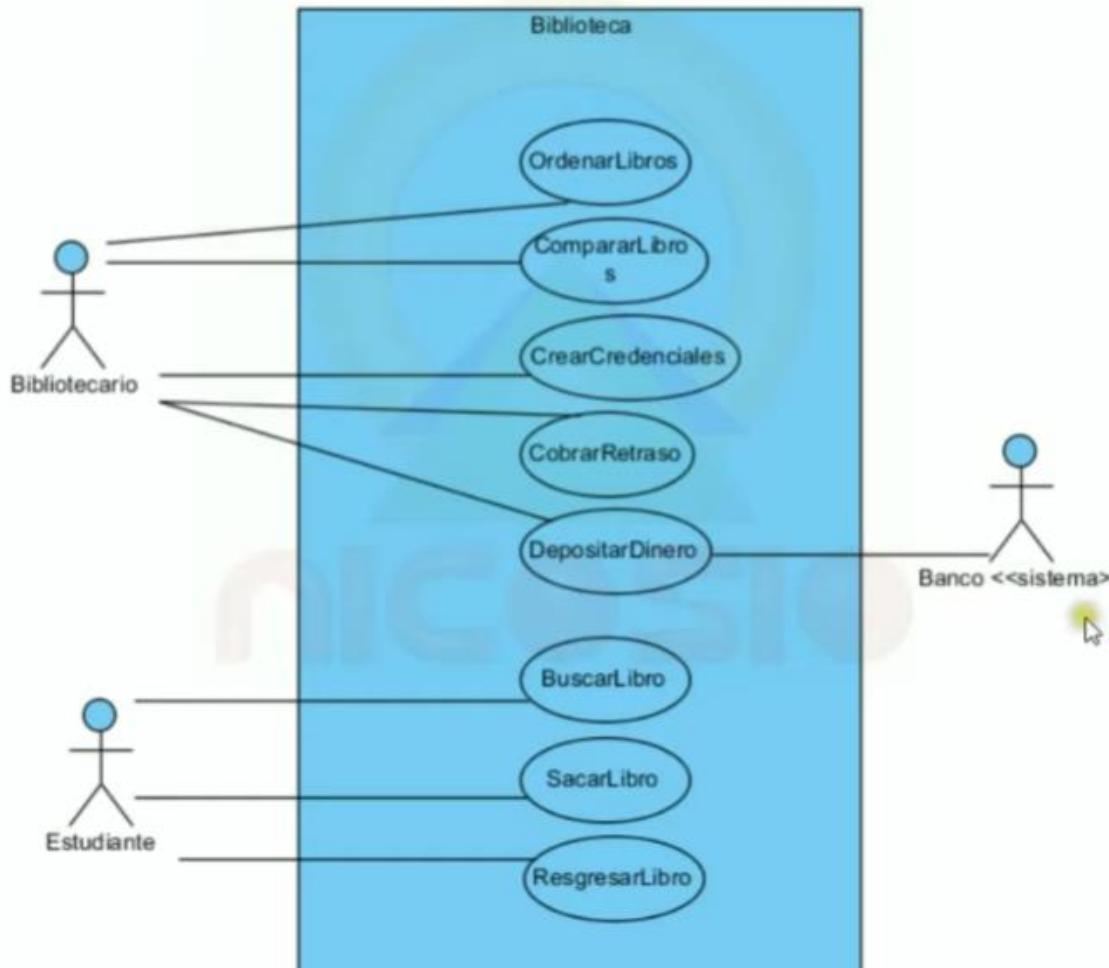




Diagrama de caso de uso



- Los casos de uso se representan mediante elipses con el nombre del caso
- Los actores pueden representarse mediante un monigote o mediante rectángulos en que se indique << actor >>
- En los diagramas, tanto los actores como los casos de uso representan no las instancias particulares, sino los conjuntos de todos los actores de un tipo y de todos los escenarios.





CASOS DE USO : toma de requerimientos

Sistema de venta de Música Online

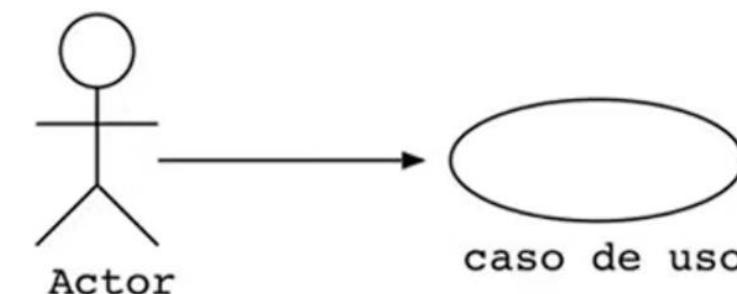
- Los usuarios comprarán créditos para adquirir canciones.
- El sistema debe registrar la información de los usuarios y los créditos que poseen.
- Los usuarios buscarán las canciones que deseen y las pagarán con créditos.
- El sistema debe almacenar información sobre las canciones que se pueden adquirir y su precio en Créditos.
- El Sistema deberá sugerir canciones acorde a los intereses del usuario.



- Los casos de uso son una técnica para la especificación de requisitos funcionales propuesta inicialmente por Jacobson (1992).

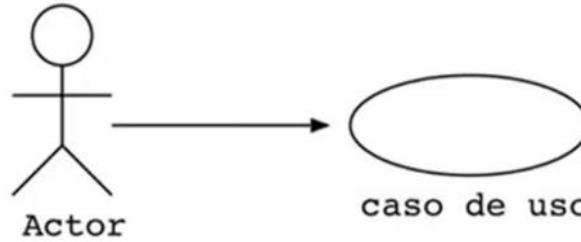
- Son, ante todo, requisitos funcionales.

- A pesar de ser una técnica ampliamente aceptada, existen múltiples propuestas para su realización.
- Los casos de uso responden a la pregunta: ¿Qué se supone que el sistema debe hacer para los usuarios?
- Presentan una ventaja sobre la descripción textual de los requisitos funcionales.





CASOS DE USO : componentes



- Se compone de
 - Diagrama
 - Actores
 - Funcionalidades (casos de uso)
 - Relaciones
- Especificación
 - Escenarios
 - Restricciones

Actor

- Un actor es alguien o algo que interactúa con el sistema (una persona, una organización, un programa o sistema de hardware o software).
- Un actor estimula al sistema con algún evento o recibe información del sistema.
- Un actor es externo al sistema.
- Un actor cumple un rol funcional definido (Ej.: Cliente, Banco, empleado).

Relaciones

- Son relaciones que usamos para ligar gráficamente dos casos de uso, cuyos flujos de eventos están unidos, normalmente en una sola sesión del usuario.

- En otras palabras, un caso de uso que está ligado, por medio de una de estas dos relaciones, a otro caso de uso; también podría leerse o ejecutarse como un sólo caso de uso en lugar de dos.

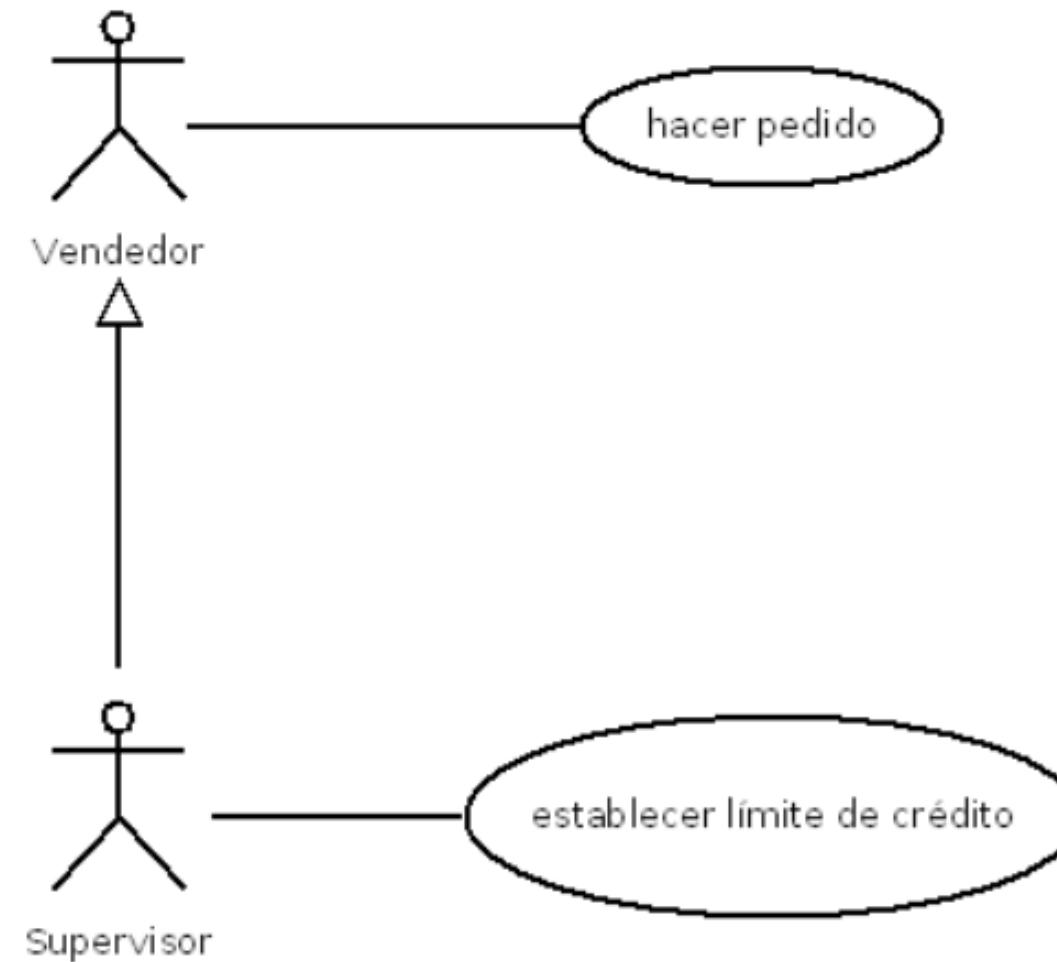
Relaciones fundamentales

- Uso
- Inclusión
- Extensión
- Generalización





Generalización - especialización entre actores



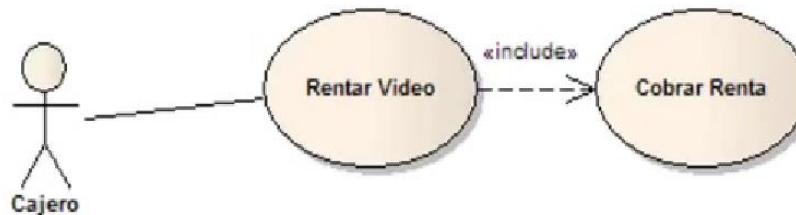


Modelado O.O

Relaciones

« include »

- Una relación de inclusión entre casos de uso especifica que un **caso de uso base** incorpora explícitamente el comportamiento de otro caso de uso en el lugar establecido en el caso de uso base.
- Una relación de inclusión se representa como una dependencia que declara que un caso de uso utiliza información y servicios de otro.



Relaciones

« extend »

- Una relación de *extensión* entre casos de uso especifica que un **caso de uso base** incorpora implícitamente el comportamiento de otro caso de uso en el lugar especificado por el caso de uso base que extiende.
- La funcionalidad de un caso de uso incluye un conjunto de pasos que ocurren sólo en algunas oportunidades.
- La excepción consiste en interrumpir el caso de uso A y pasar a ejecutar otro caso de uso B.



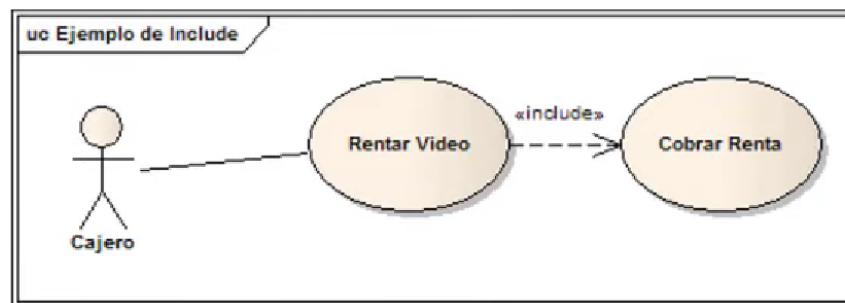


Modelado O.O

Especificación

Documentación del caso de uso

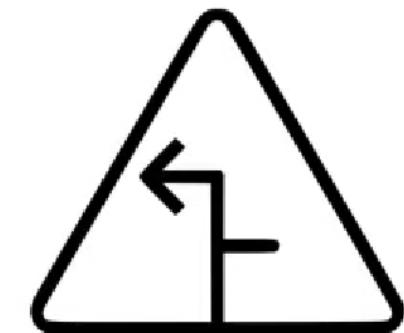
- Nombre
- Actores
- Extensiones
- Pre condición
- Escenario principal
 - Paso 1..n
- Post condición
- Escenarios alternativos.



Escenario

- Un caso de uso no trivial describe un conjunto de secuencias, no una única secuencia. Es conveniente separar el flujo principal de los flujos alternativos.
- Es un texto muy simple con cierto formato que describe cómo se debería comportar un sistema ante la interacción con uno o más usuarios
- Cada secuencia específica del caso de uso se denomina **Escenario**.
- Un **escenario** es una secuencia específica de acciones entre los actores y el sistema (es una instancia de un caso de uso).

- Tipos
 - Principal
 - Alternativo



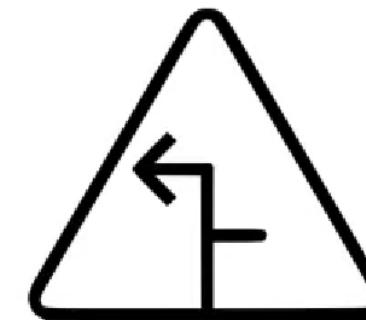


Modelado O.O

Escenario

Descripción del caso de uso

- Un caso de uso no trivial describe un conjunto de secuencias, no una única secuencia. Es conveniente separar el flujo principal de los flujos alternativos.
- Es un texto muy simple con cierto formato que describe cómo se debería comportar un sistema ante la interacción con uno o más usuarios
- Cada secuencia específica del caso de uso se denomina **Escenario**.
- Un **escenario** es una secuencia específica de acciones entre los actores y el sistema (es una instancia de un caso de uso).
- Tipos
 - Principal
 - Alternativo



Ejemplo:

Escenario principal

1. El actor solicita crear una nueva factura.
2. El sistema muestra una pantalla y solicita selección de cliente y productos a facturar.
3. El actor selecciona un cliente
4. El sistema informa que el cliente es válido para facturar.
5. El actor selecciona un producto para facturar.
6. El sistema confirma que el producto es válido para facturar.
7. El actor finaliza la factura
8. El sistema confirma que la factura se emitió correctamente e informa el número de comprobante

Escenarios alternativos

- 4.1. El sistema informa que el cliente posee deuda y no puede facturarse.
- 4.2. Se finaliza el caso de uso





Pre condiciones: establece lo que siempre debe cumplirse antes de comenzar un caso de uso. No se prueban en el caso de uso, se asumen que son verdaderas.

Post condiciones: establece qué debe cumplirse cuando el caso de uso se completa con éxito.

Ejemplo:

Precondición:

- Qué el cliente no posea deuda.

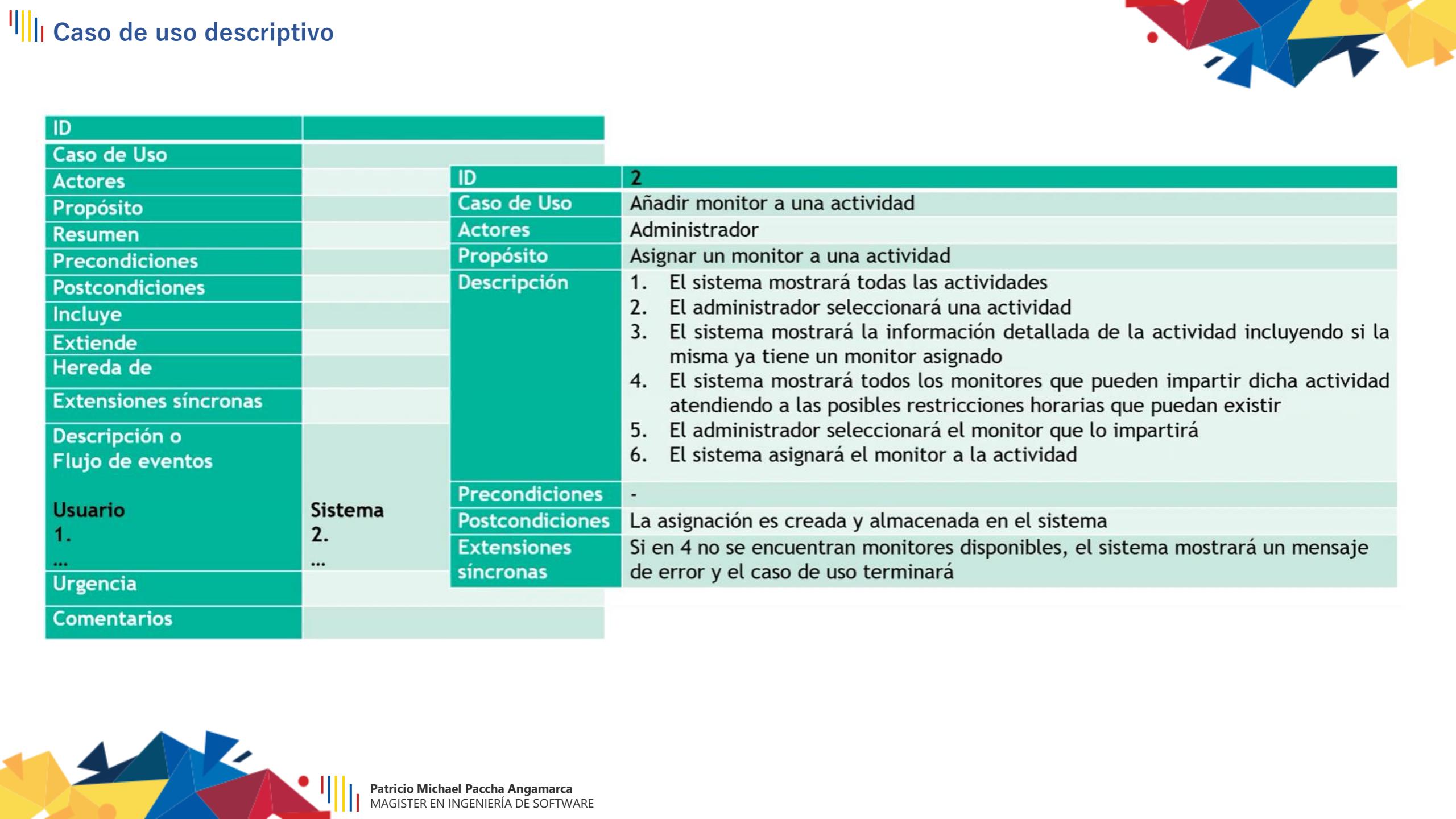
Postcondición:

- Una nueva factura se genera en el sistema.
- Una nueva deuda se le imputa al cliente

Caso de uso. Plantilla tipo

| ID | |
|-----------------------------------|----------------------|
| Caso de Uso | |
| Actores | |
| Propósito | |
| Resumen | |
| Precondiciones | |
| Postcondiciones | |
| Incluye | |
| Extiende | |
| Hereda de | |
| Extensiones sincronas | |
| Descripción o Flujo de eventos | |
| Usuario 1. ... Urgencia | Sistema 2. ... |
| Comentarios | |
| Frecuencia esperada | |
| Rendimiento | |
| Importancia | |

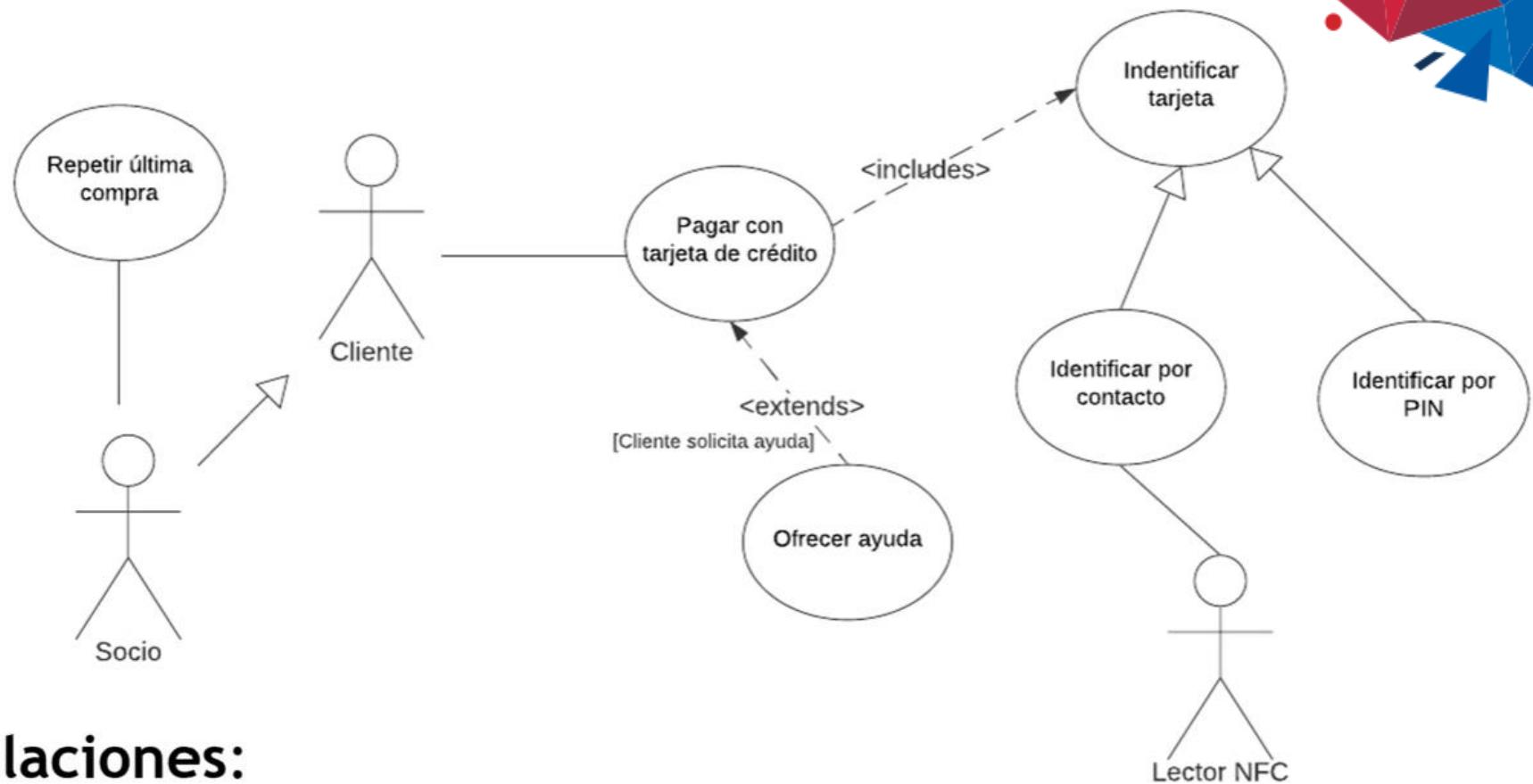




Caso de uso descriptivo

| | |
|-----------------------------------|----------------------|
| ID | |
| Caso de Uso | |
| Actores | |
| Propósito | |
| Resumen | |
| Precondiciones | |
| Postcondiciones | |
| Incluye | |
| Extiende | |
| Hereda de | |
| Extensiones síncronas | |
| Descripción o Flujo de eventos | |
| Usuario 1. ... Urgencia | Sistema 2. ... |
| Comentarios | |

| | |
|--------------------------|--|
| ID | 2 |
| Caso de Uso | Añadir monitor a una actividad |
| Actores | Administrador |
| Propósito | Asignar un monitor a una actividad |
| Descripción | <ol style="list-style-type: none">El sistema mostrará todas las actividadesEl administrador seleccionará una actividadEl sistema mostrará la información detallada de la actividad incluyendo si la misma ya tiene un monitor asignadoEl sistema mostrará todos los monitores que pueden impartir dicha actividad atendiendo a las posibles restricciones horarias que puedan existirEl administrador seleccionará el monitor que lo impartiráEl sistema asignará el monitor a la actividad |
| Precondiciones | - |
| Postcondiciones | La asignación es creada y almacenada en el sistema |
| Extensiones síncronas | Si en 4 no se encuentran monitores disponibles, el sistema mostrará un mensaje de error y el caso de uso terminará |



Tipos de relaciones:

- CU1 *<includes>* CU2
- CU1 *<extends>* CU2
- CU1 hereda de CU2
- Actor1 participa en CU1
- Actor1 hereda de Actor2



#HágalePues!



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE
ESCUELA POLITÉCNICA NACIONAL

Polireto



Observador



Lobo



Caperucita



Uvas



Problema

- Cruzar todos de un lado del río al otro extremo

RECURSOS:

- Team
- Pizarra
- Marcador
- Sticky-note





Relación de herencia:

- *todos los* coches *son* vehículos

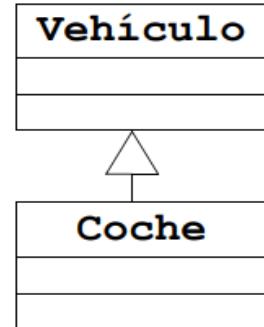
La herencia es un mecanismo por el que se pueden crear nuevas clases a partir de otras existentes,

- heredando, y posiblemente modificando, y/o añadiendo operaciones
- heredando y posiblemente añadiendo atributos

Observar que una operación o atributo no puede ser suprimido en el mecanismo de herencia

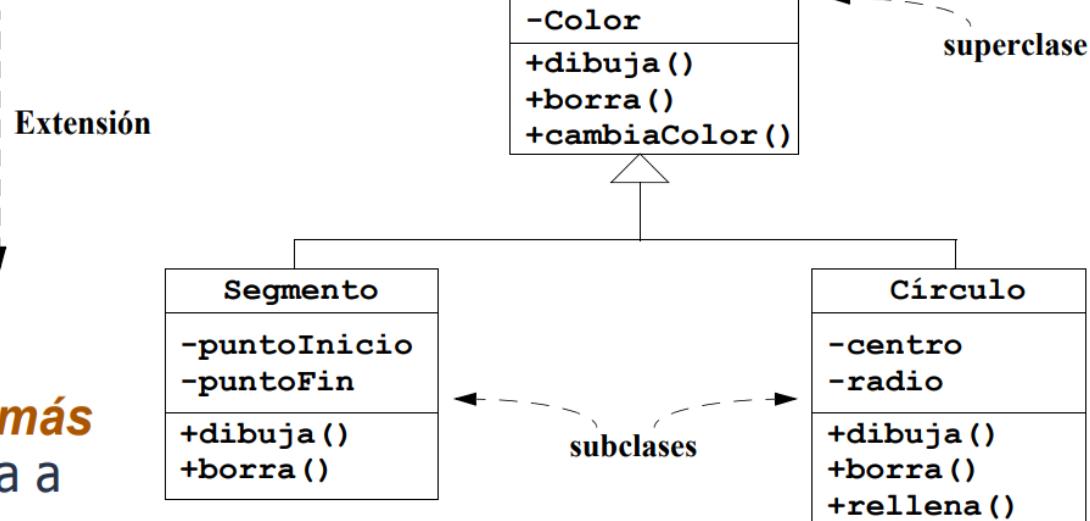
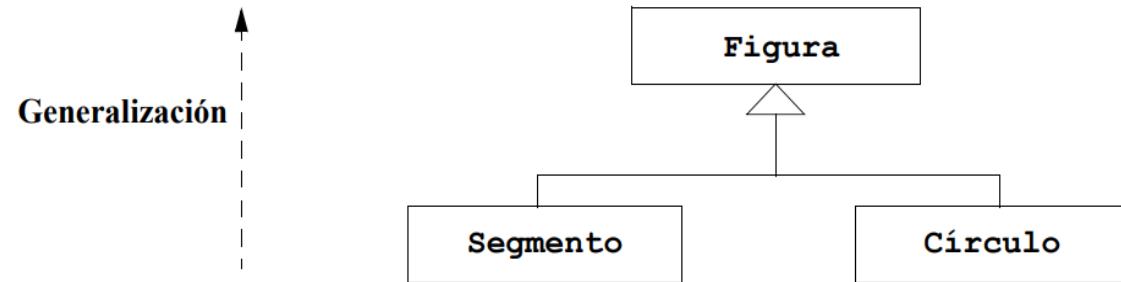
Nomenclatura

| | | | |
|-----------------|------------|-------|----------|
| clase original | superclase | padre | Vehículo |
| clase extendida | subclase | hijo | Coche |





La **Herencia** también se denomina **Extensión** o **Generalización**



La **Herencia** (y el Polimorfismo) son unos de los **conceptos más importantes y diferenciadores** de la Programación Orientada a Objetos

Al extender una clase

- se **heredan** todas las operaciones del padre
- se puede **añadir** nuevas operaciones

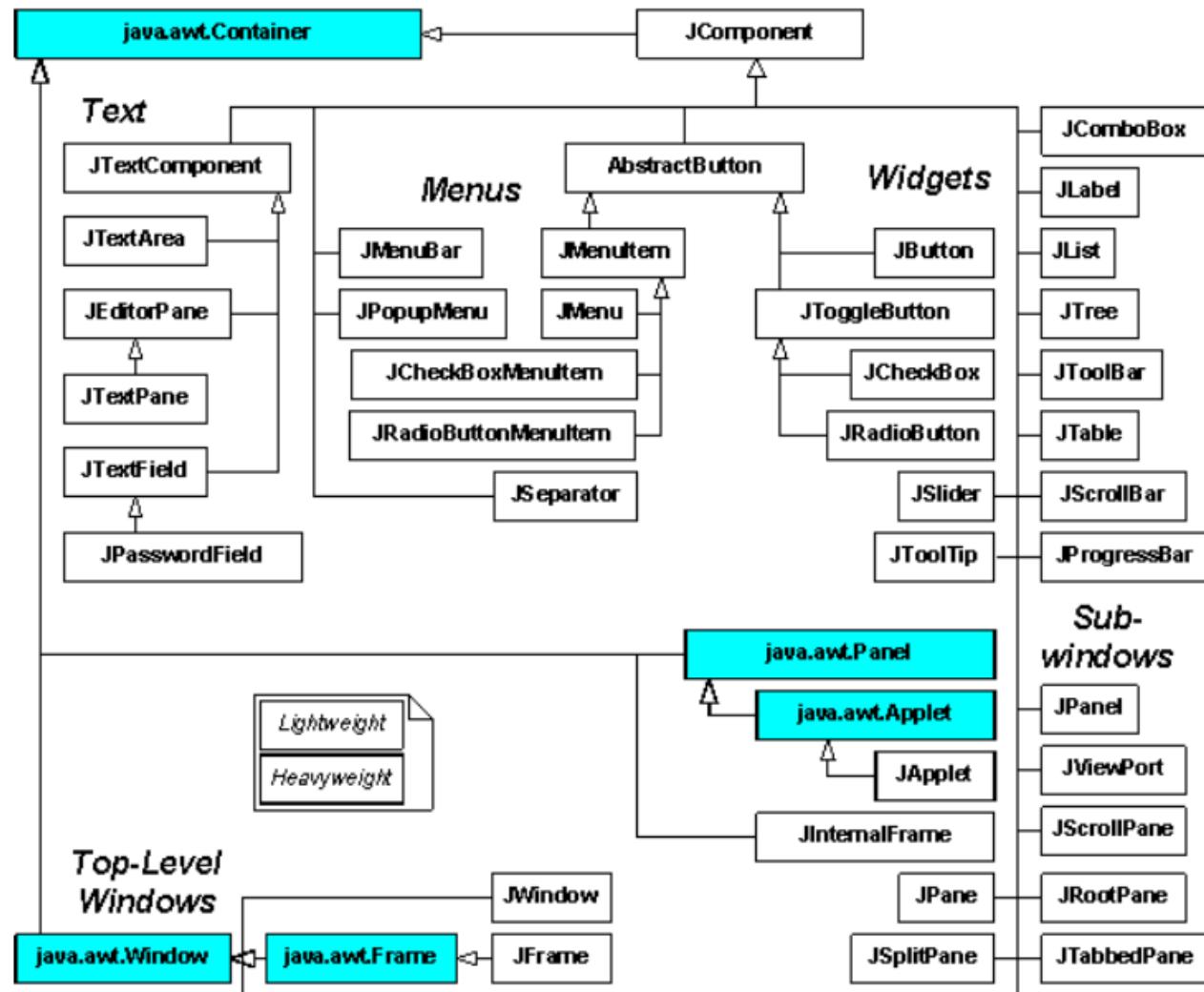
La subclase puede elegir para las operaciones heredadas:

- **redefinir** la operación: se vuelve a escribir
 - la nueva operación puede usar la del padre y hacer más cosas
 - o puede ser totalmente diferente
- no hacer nada y heredarla tal como está en el padre





La herencia puede aplicarse en sucesivos niveles, creando grandes **jerarquías de clases**



**Jerarquía de clases
de la biblioteca
gráfica Swing**



Ventajas:

+Mejora el **diseño**

Permite modelar relaciones de tipo "es un" que se dan en los problemas que se pretenden resolver

+Permite la **reutilización** del código

Los métodos de la clase padre se reutilizan en las clases hijas

+Facilita la **extensión** de las aplicaciones

Añadir una nueva subclase no requiere modificar ninguna otra clase de nuestro diseño

Principal desventaja:

- Aumenta el **acoplamiento**

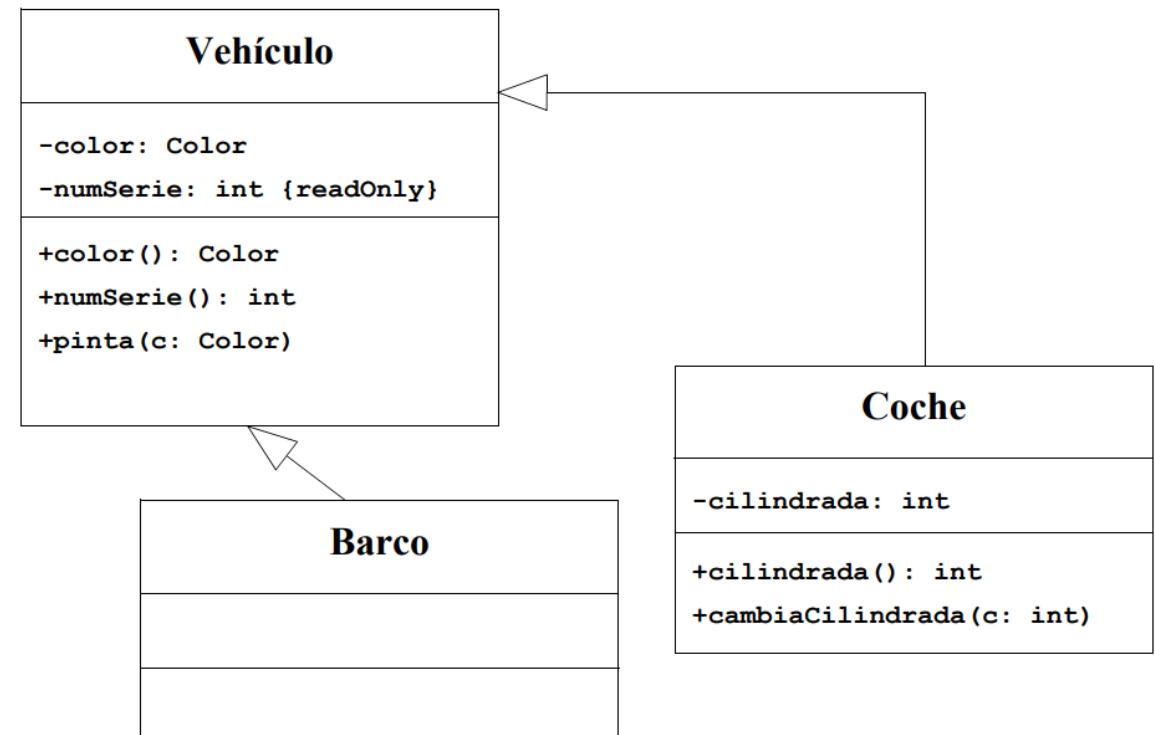
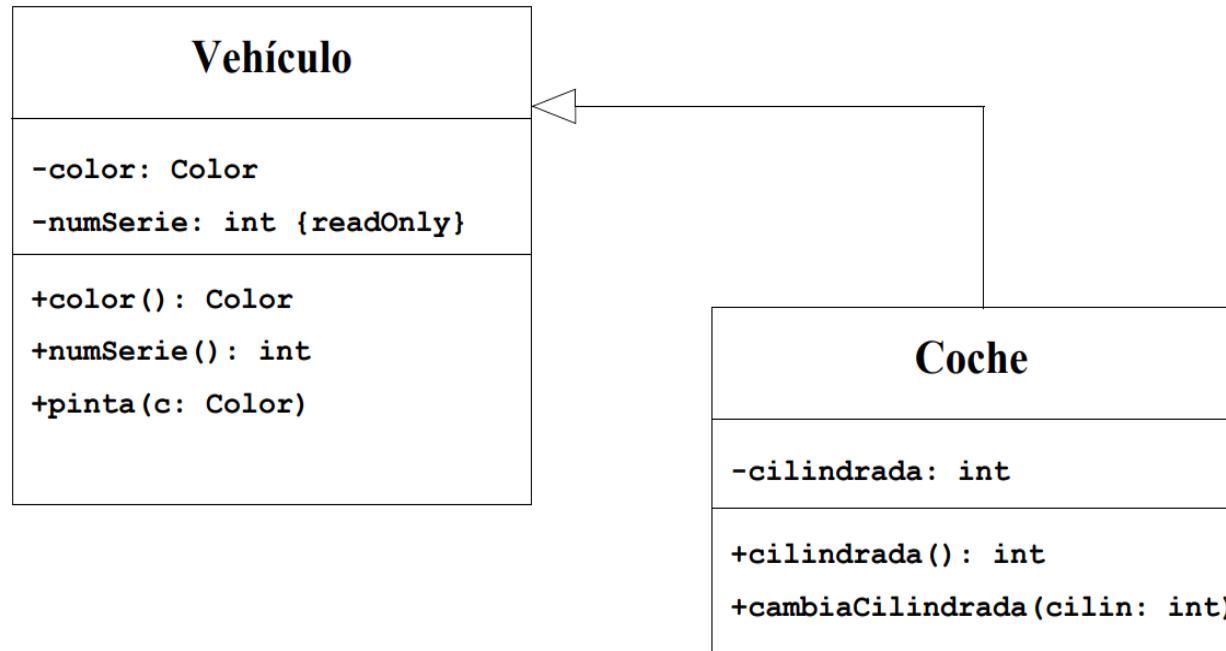
Las subclases están íntimamente acopladas con la superclase

Vehículo

```
-color: Color  
-numSerie: int {readOnly}
```

```
+color(): Color  
+numSerie(): int  
+pinta(c: Color)
```





La clase **Coche** añade el atributo cilindrada y los métodos para gestionar dicho atributo
(Observar que se puede extender una clase sin añadir atributos ni métodos)





Implementación del ejemplo

```
/**  
 * Clase que representa un vehículo cualquiera  
 */  
public class Vehículo  
{  
    // colores de los que se puede pintar un vehículo  
    public static enum Color {ROJO, VERDE, AZUL}  
  
    // atributos  
    private Color color;  
    private final int numSerie;  
  
    /**  
     * Construye un vehículo  
     * @param color color del vehículo  
     * @param numSerie número de serie del vehículo  
     */  
    public Vehículo(Color color, int numSerie)  
    {  
        this.color=color;  
        this.numSerie=numSerie;  
    }
```

```
    /**  
     * Retorna el color del vehículo  
     * @return color del vehículo  
     */  
    public Color color()  
    {  
        return color;  
    }  
    /**  
     * Retorna el numero de serie del vehículo  
     * @return numero de serie del vehículo  
     */  
    public int numSerie()  
    {  
        return numSerie;  
    }  
    /**  
     * Pinta el vehículo de un color  
     * @param nuevoColor color con el que pintar el vehículo  
     */  
    public void pinta(Color c)  
    {  
        color = c;  
    }  
}
```





```
public class Coche extends Vehículo
{
    // cilindrada del coche
    private int cilindrada;

    /** Retorna la cilindrada del coche ... */
    public int cilindrada(){
        return cilindrada;
    }

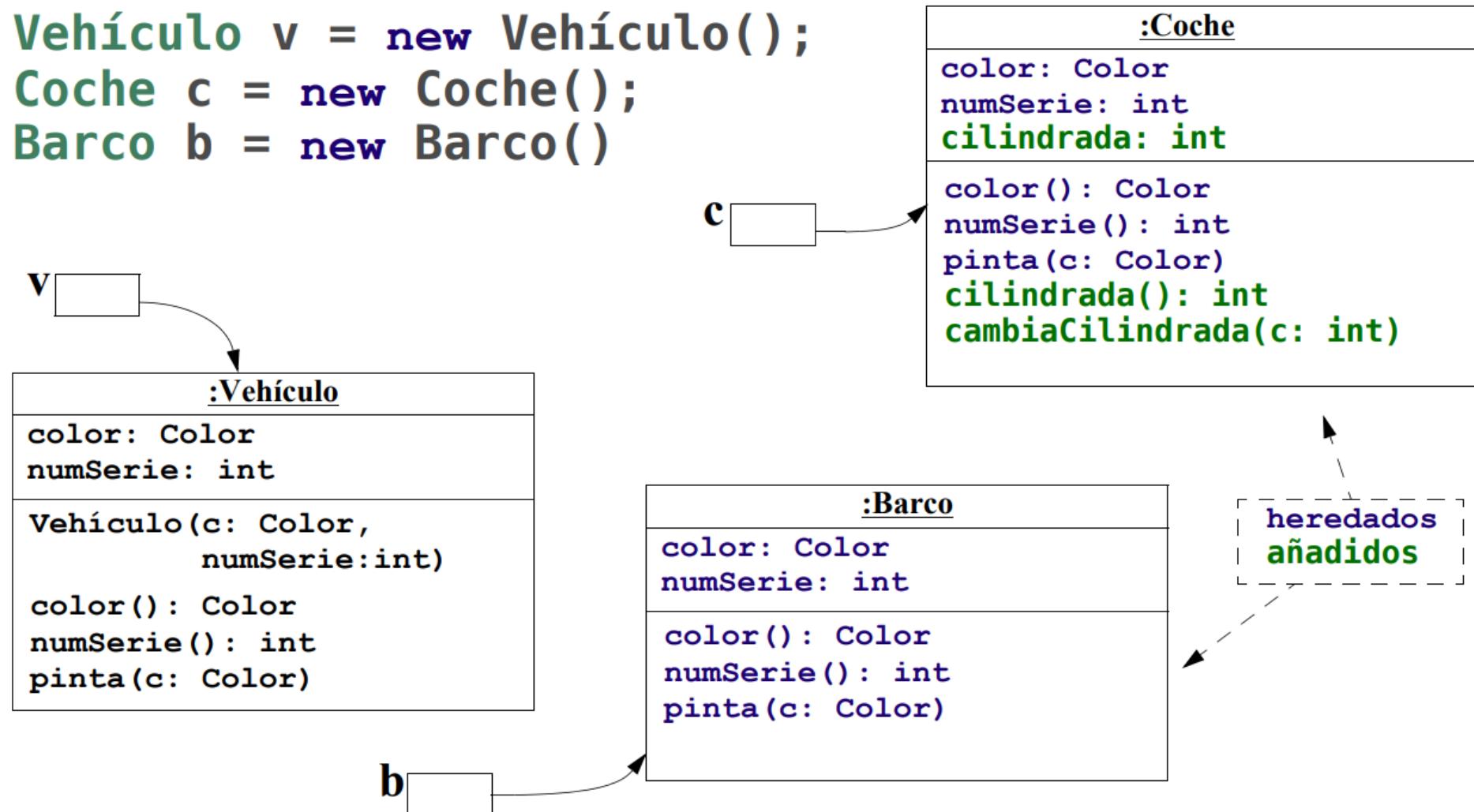
    /** Cambia la cilindrada del coche ... */
    public void cambiaCilindrada(int c) {
        this.cilindrada=c;
    }
}
```

```
public class Barco extends Vehículo
{}
```





```
Vehículo v = new Vehículo();  
Coche c = new Coche();  
Barco b = new Barco()
```





Los constructores no se heredan

- las subclases deben definir su propio constructor

Normalmente será necesario inicializar los atributos de la superclase

- para ello se llama a su constructor desde el de la subclase

```
/** constructor de una subclase */
public Subclase(parámetros...) {
    // invoca el constructor de la superclase
    super(parámetros para la superclase);
    // inicializa sus atributos
    ...
}
```

- la llamada a “super” debe ser la primera instrucción del constructor de la subclase





Si desde un constructor de una subclase no se llama expresamente al de la superclase

- el compilador añade la llamada al constructor sin parámetros

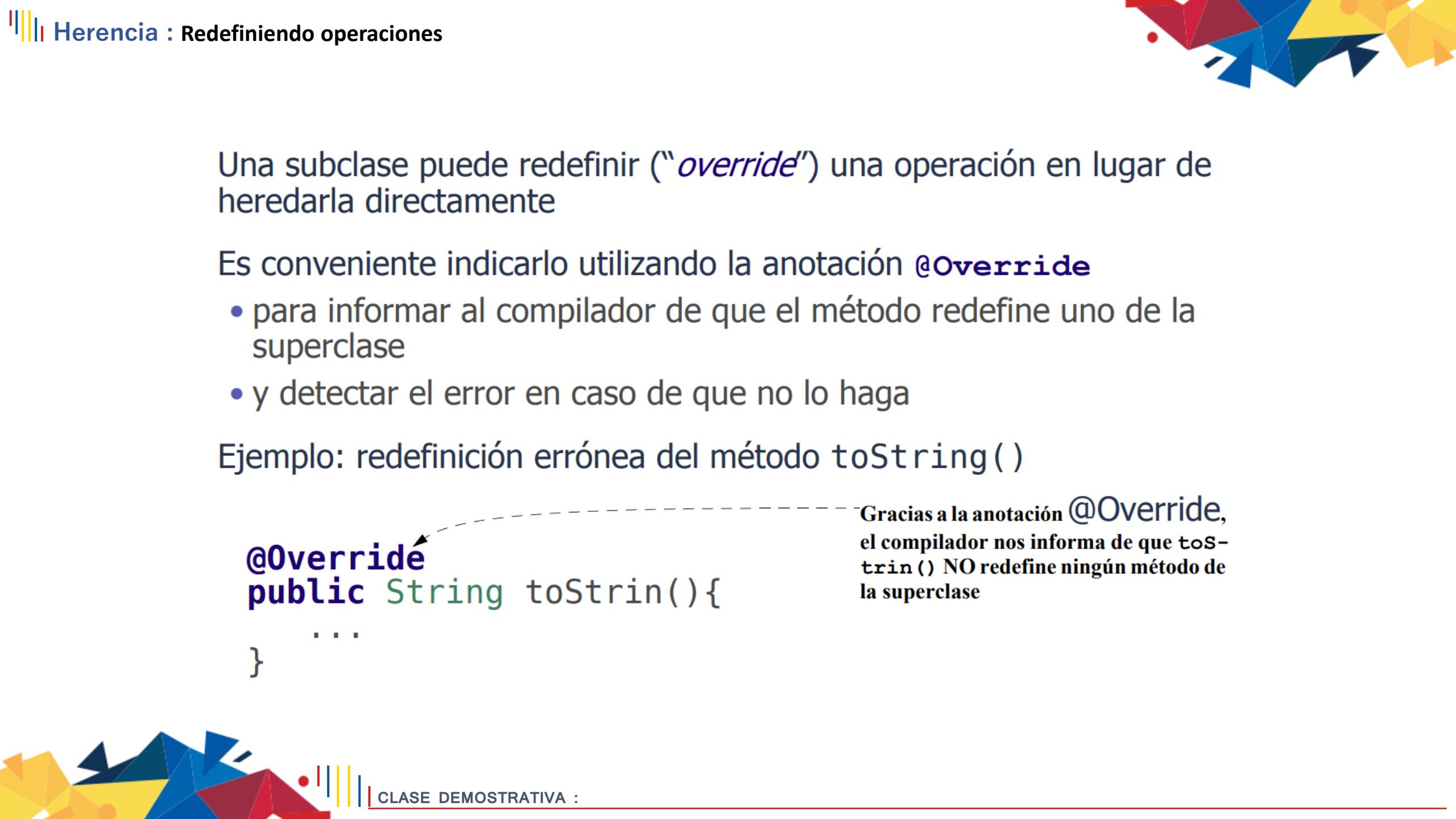
```
public Subclase(int i){  
    this.i=i;  
}
```

se convierte en

```
public Subclase(int i){  
    super();  
    this.i=i;  
}
```

- en el caso de que la superclase no tenga un constructor sin parámetros se produciría un error de compilación





Una subclase puede redefinir ("*override*") una operación en lugar de heredarla directamente

Es conveniente indicarlo utilizando la anotación **@Override**

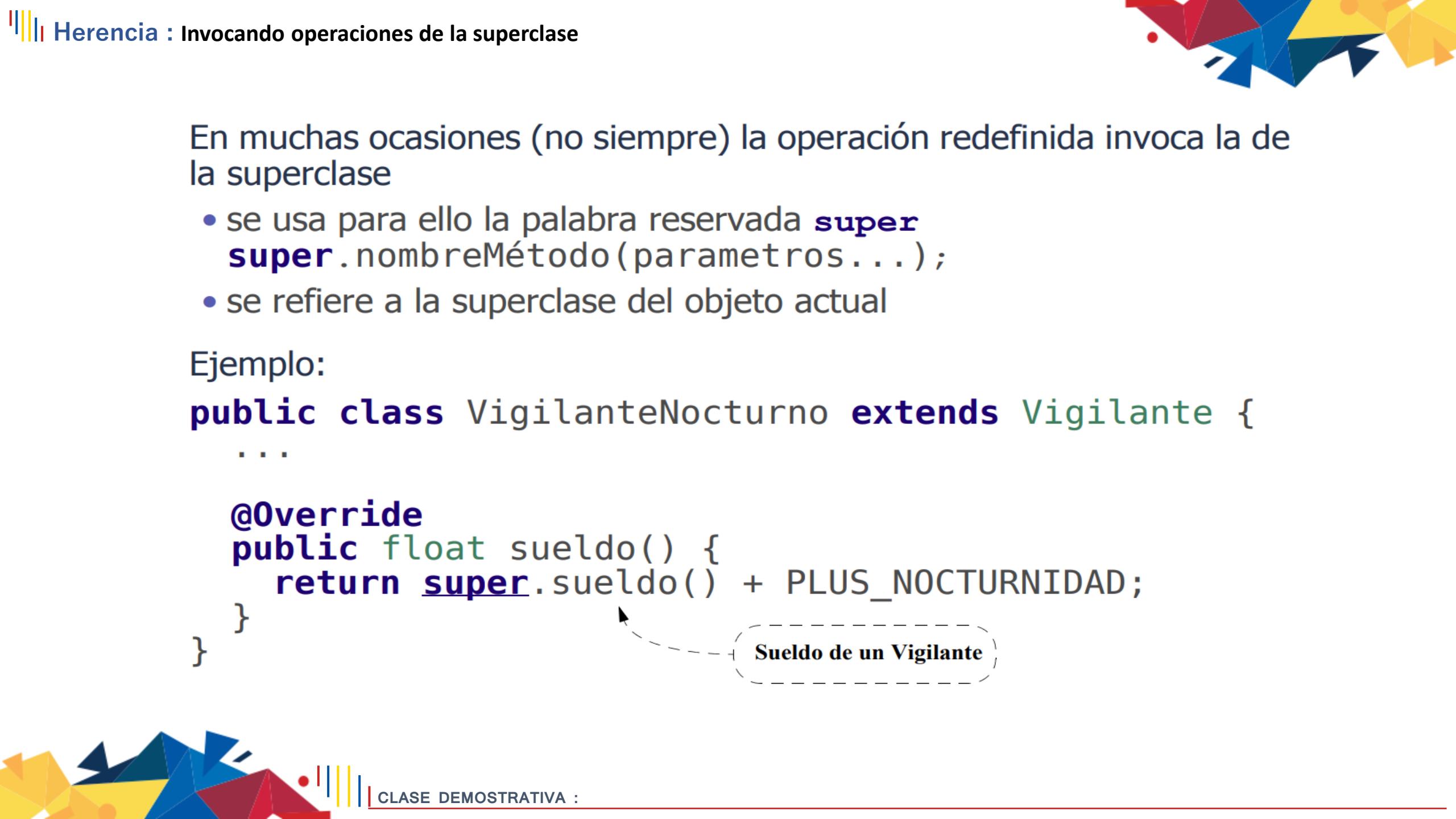
- para informar al compilador de que el método redefine uno de la superclase
- y detectar el error en caso de que no lo haga

Ejemplo: redefinición errónea del método `toString()`

```
@Override
public String toString(){  
    ...  
}
```

Gracias a la anotación **@Override**,
el compilador nos informa de que `toS-
trin()` NO redefine ningún método de
la superclase





En muchas ocasiones (no siempre) la operación redefinida invoca la de la superclase

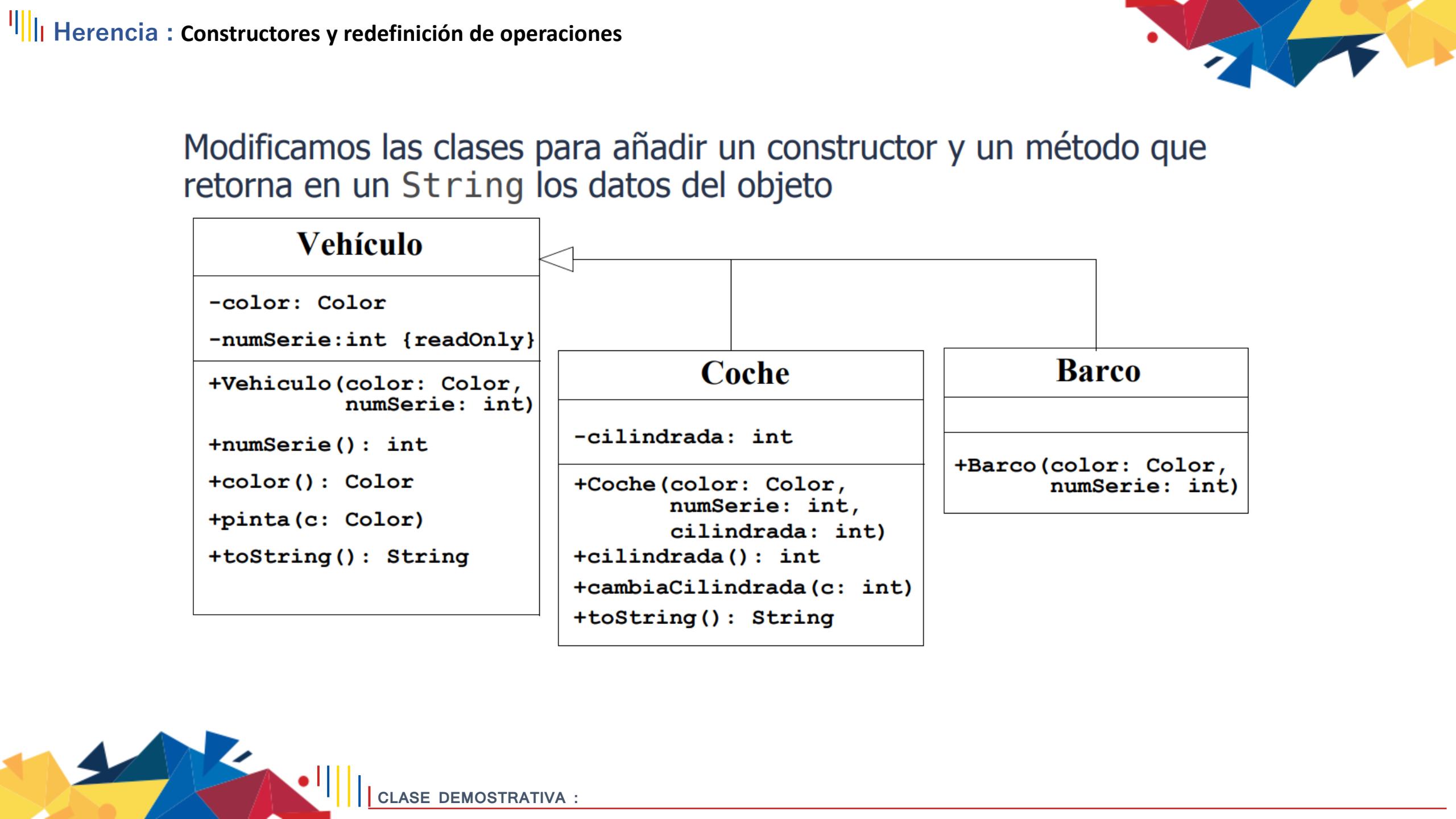
- se usa para ello la palabra reservada **super**
super.nombreMétodo(parametros...);
- se refiere a la superclase del objeto actual

Ejemplo:

```
public class VigilanteNocturno extends Vigilante {  
    ...  
  
    @Override  
    public float sueldo() {  
        return super.sueldo() + PLUS_NOCTURNIDAD;  
    }  
}
```

Sueldo de un Vigilante







```
public class Vehículo {  
    // colores de los que se puede pintar un vehículo  
    public static enum Color {ROJO, VERDE, AZUL}  
    // atributos privados  
    private Color color;  
    private final int numSerie;  
  
    /**  
     * Construye un vehículo  
     * @param color color del vehículo  
     * @param numSerie número de serie del vehículo  
     */  
    public Vehículo(Color color, int numSerie) {  
        this.color = color;  
        this.numSerie = numSerie;  
    }  
}
```





Ejemplo: clase Vehículo (cont.)

```
public int numSerie() {...} ← No repetimos el código  
public Color color() ... ← (es igual que en el ejemplo anterior)  
public void pinta(Color c) ...  
  
/**  
 * Retorna un texto con los datos del vehículo  
 * @return texto con los datos del vehículo  
 */  
@Override  
public String toString() {  
    return "Vehículo -> numSerie= " +  
        numSerie + ", color= " + color;  
}
```





```
public class Coche extends Vehículo {  
  
    // cilindrada del coche  
    private int cilindrada;  
  
    /**  
     * Construye un coche  
     * @param color color del coche  
     * @param numSerie número de serie del coche  
     * @param cilindrada cilindrada del coche  
     */  
    public Coche(Color color, int numSerie,  
                int cilindrada) {  
        super(color, numSerie);  
        this.cilindrada = cilindrada;  
    }  
}
```





Ejemplo: subclase Coche (cont.)

```
/** Obtiene la cilindrada del coche ... */
public int cilindrada() {
    return cilindrada;
}

/** Cambia la cilindrada del coche ... */
public void cambiaCilindrada(int nueva) {
    cilindrada = nueva;
}

@Override
public String toString() {
    return super.toString() + ", cilindrada= "
        + cilindrada;
}
```





```
public class Barco extends Vehículo {  
  
    /**  
     * Construye un barco  
     * @param color color del barco  
     * @param numSerie número de serie del barco  
     */  
    public Barco(Color color, int numSerie) {  
        super(color, numSerie);  
    }  
}
```

Modificadores de acceso para miembros de clases:

- <ninguno>: accesible desde el paquete
- **public**: accesible desde todo el programa
- **private**: accesible sólo desde esa clase
- **protected**: accesible desde sus subclases y, en Java, desde cualquier clase en el mismo paquete

UnaClase

| |
|-----------------------|
| + atrPúblico |
| - atrPrivado |
| ~ atrPaquete |
| # atrProtegido |

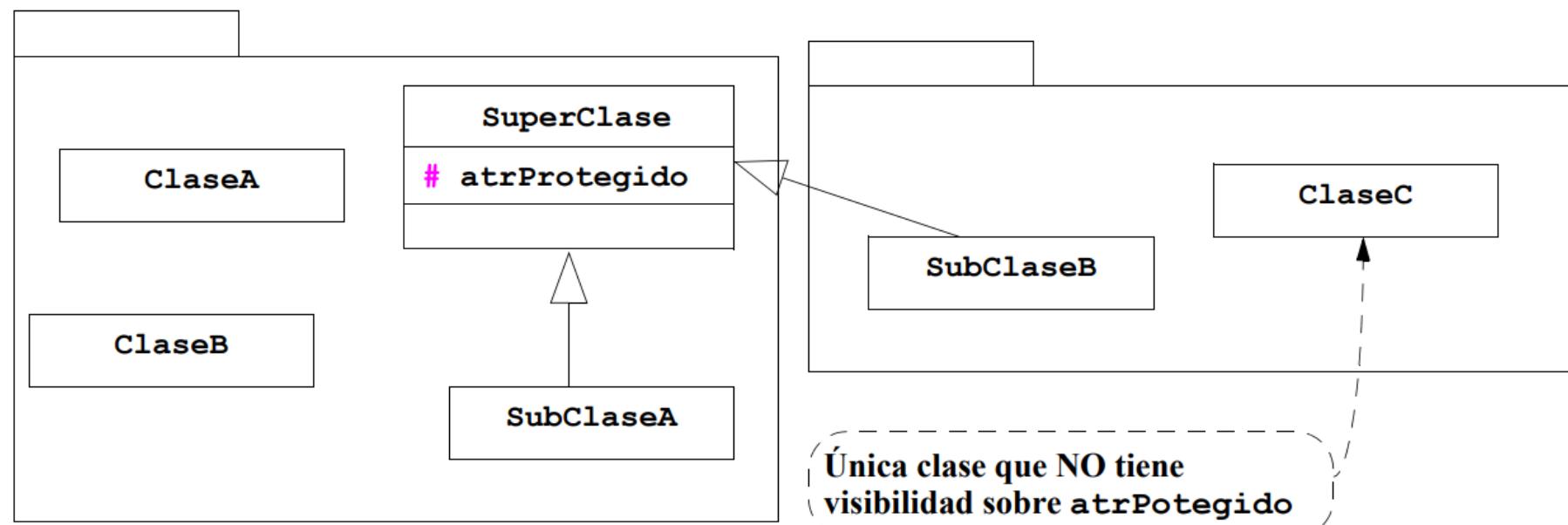
| |
|-----------------------|
| + metPúblico |
| - metPrivado |
| ~ metPaquete |
| # metProtegido |





En general, definir **atributos protected** en Java **NO es una buena práctica** de programación

- ese atributo sería accesible desde cualquier subclase
 - puede haber muchas y eso complicaría enormemente la tarea de mantenimiento
- además (en Java) el atributo es accesible desde todas las clases del paquete (subclases o no)

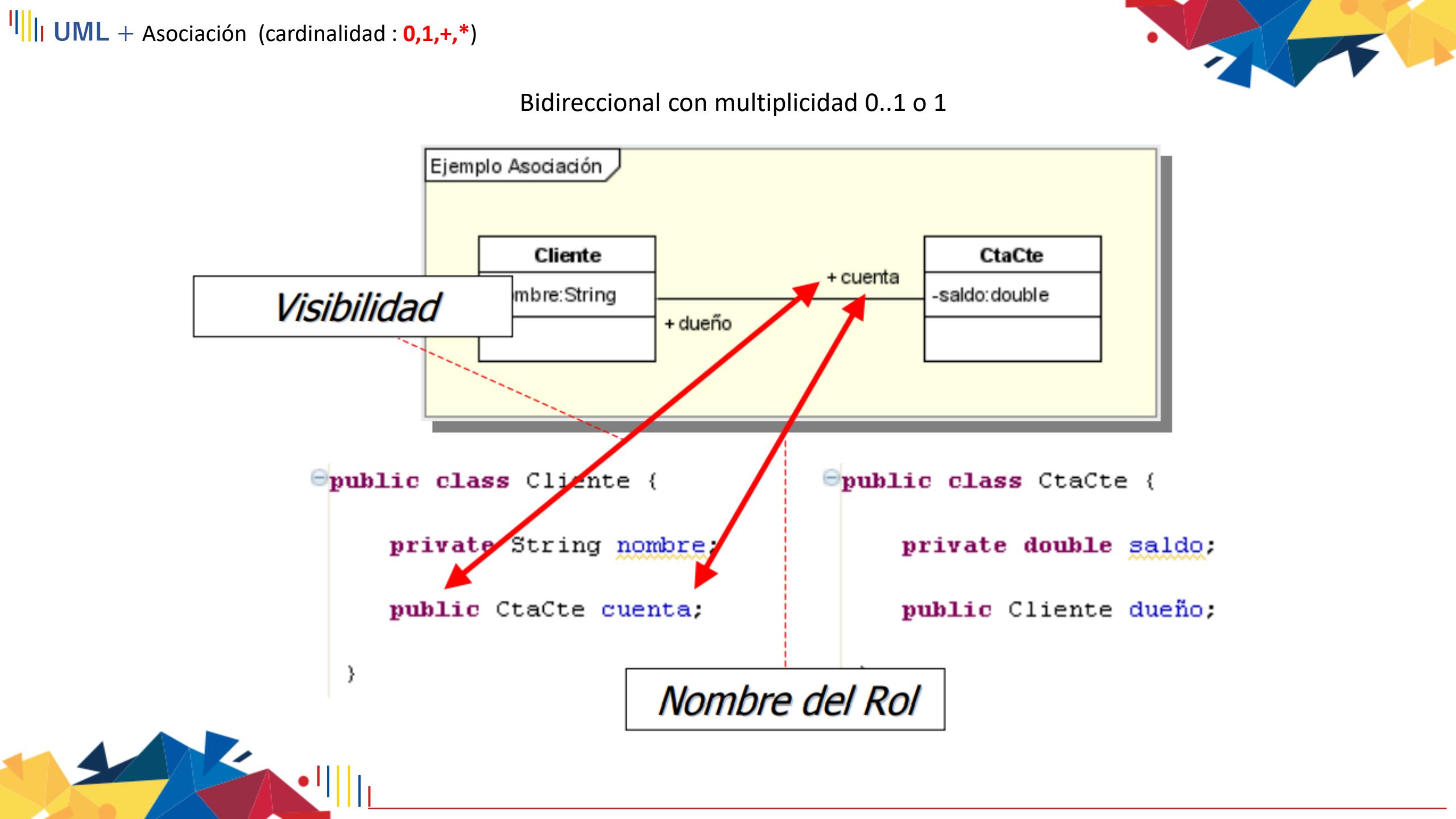




| | Misma clase | Subclase | Mismo paquete | Otro paquete |
|----------------------------------|-------------|----------|---------------|--------------|
| Sin modificador (paquete) | Sí | | Sí | |
| <code>public</code> | Sí | Sí | Sí | Sí |
| <code>private</code> | Sí | | | |
| <code>protected</code> | Sí | Sí | Sí | |

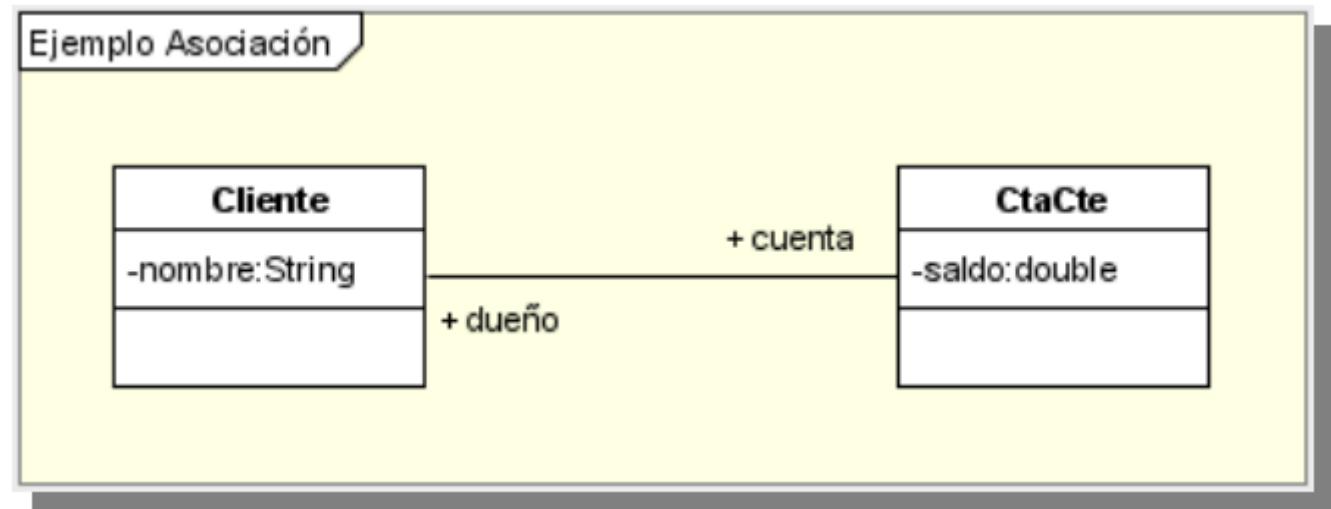
- ✓ Modificador `static`. Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, todas las clases compartirán ese mismo atributo con el mismo valor. Es un caso de miembro estático o miembro de clase: un **atributo estático** o **atributo de clase** o **variable de clase**.
- ✓ Modificador `final`. Indica que el atributo es una **constante**. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los **atributos constantes** (`final`) se escribe con **todas las letras en mayúsculas**.







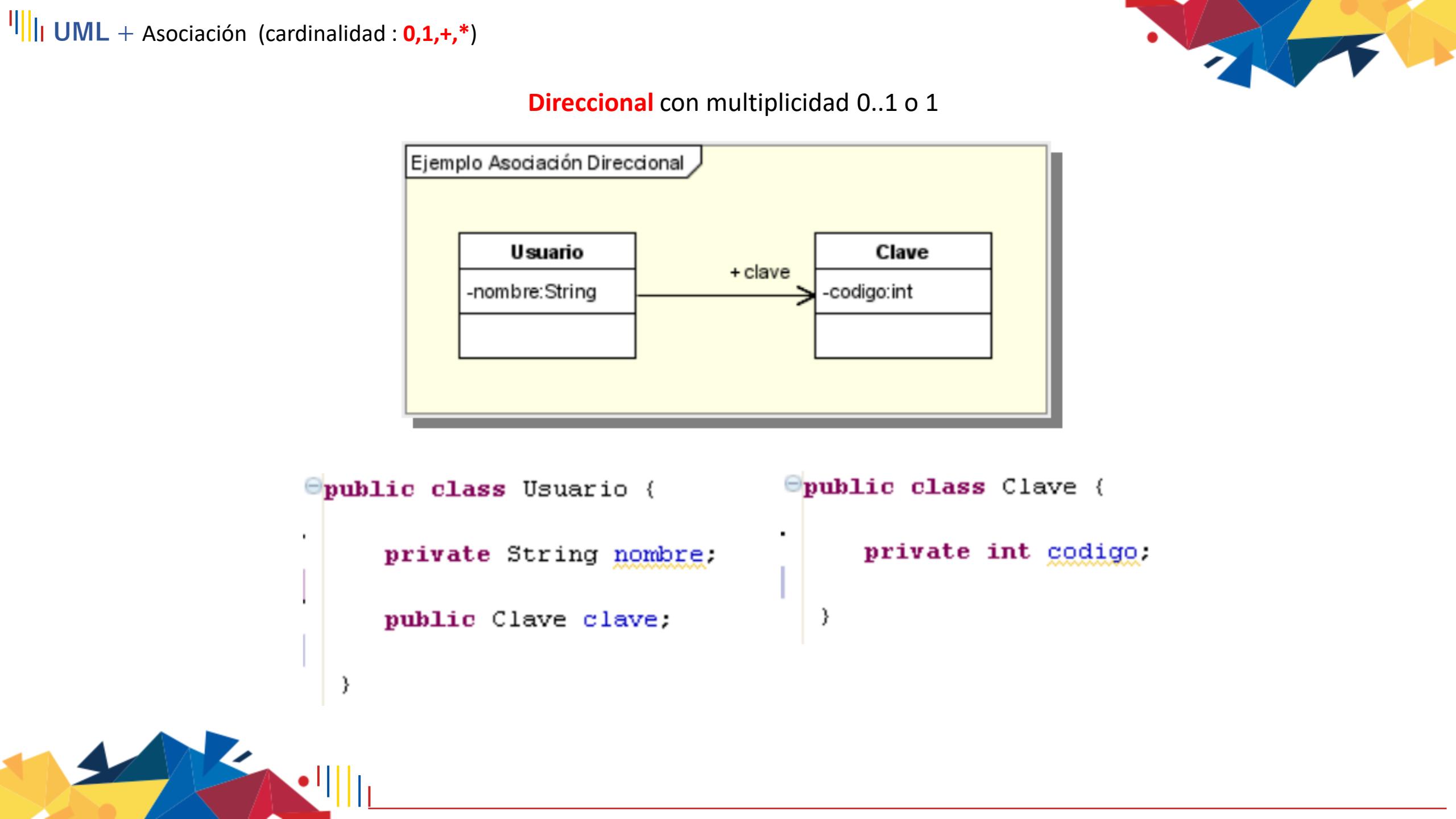
Bidireccional con multiplicidad 0..1 o 1



```
@public class Cliente {  
    private String nombre;  
    public CtaCte cuenta;  
}
```

```
@public class CtaCte {  
    private double saldo;  
    public Cliente dueño;  
}
```

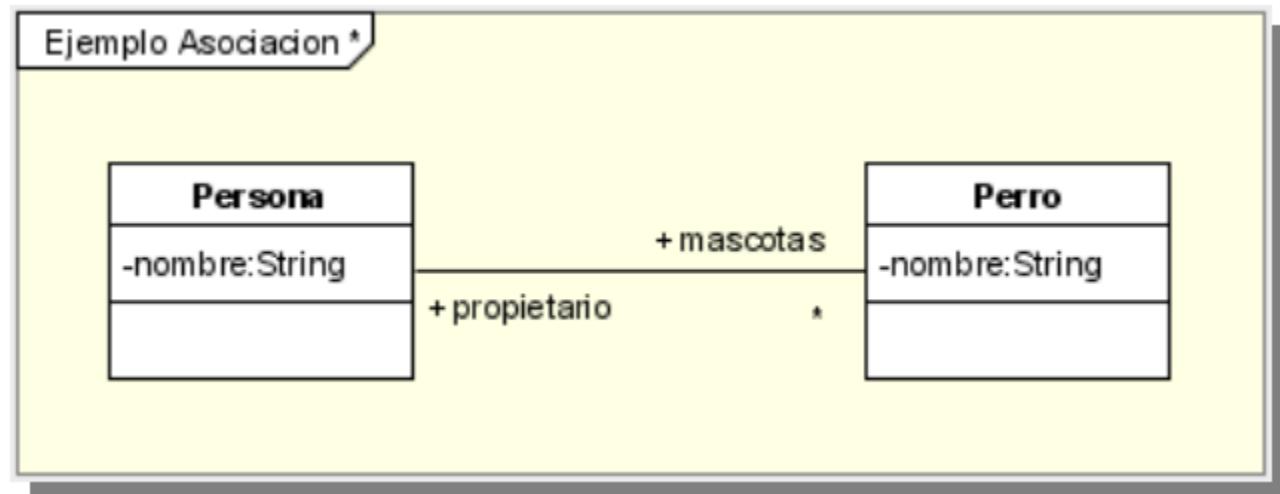




Direccional con multiplicidad 0..1 o 1



Bidireccional con multiplicidad *



```
public class Persona {

    private String nombre;
    Java.util.List<Perro> mascotas = new ArrayList<Perro>();
    public java.util.Collection mascotas = new java.util.TreeSet();

}
```

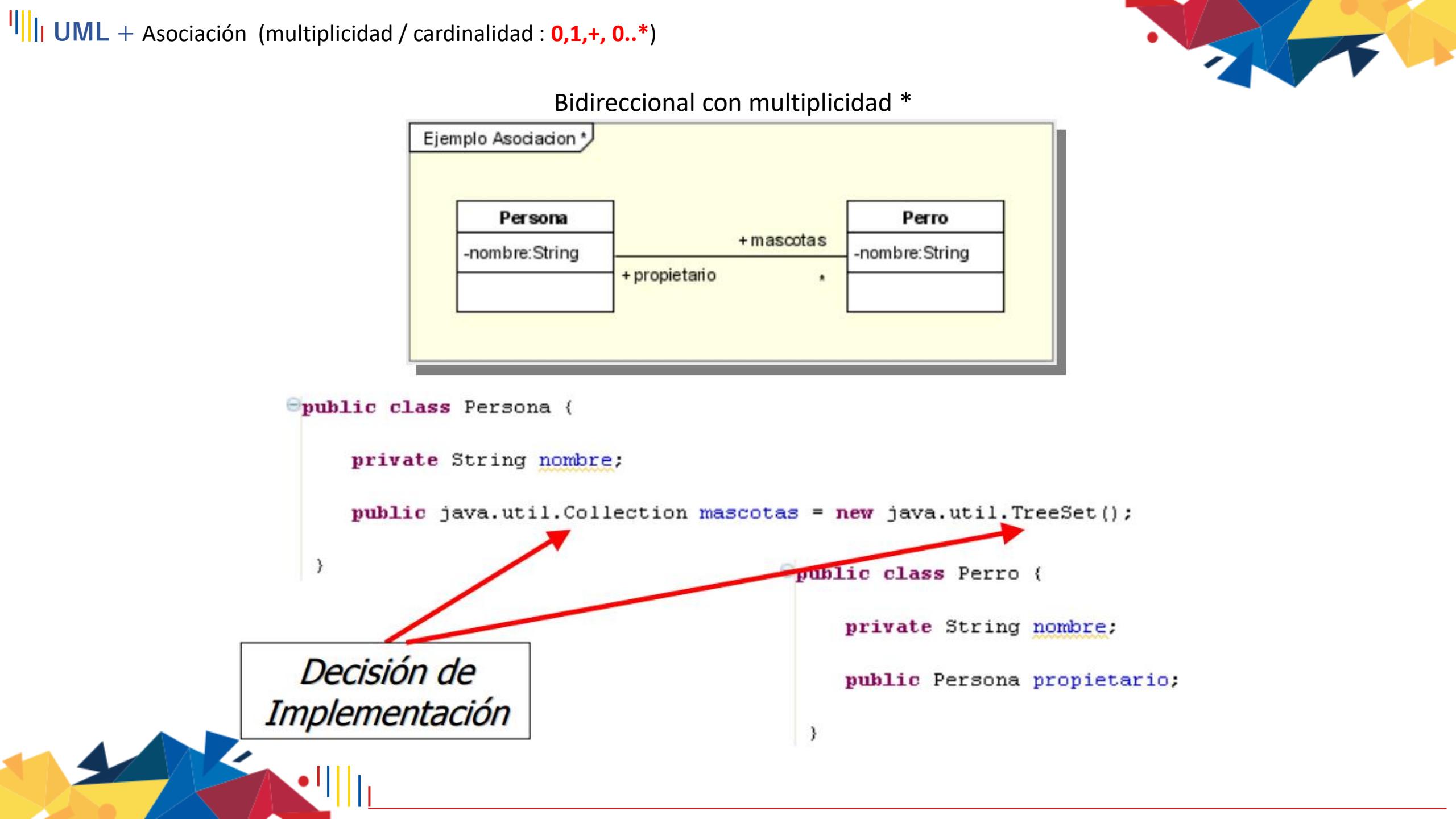
```
public class Perro {

    private String nombre;

    public Persona propietario;

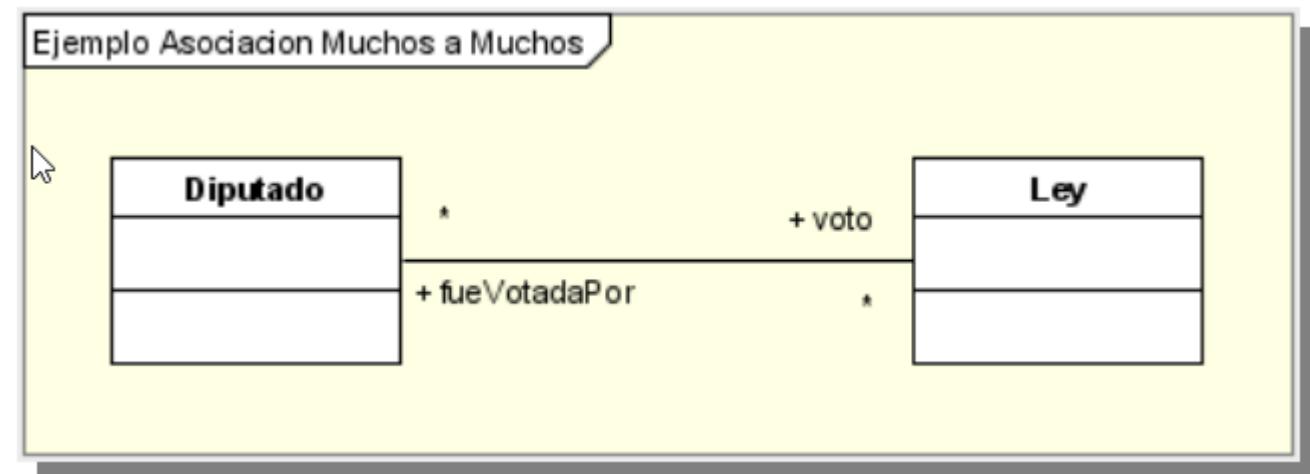
}
```







Bidireccional con multiplicidad *



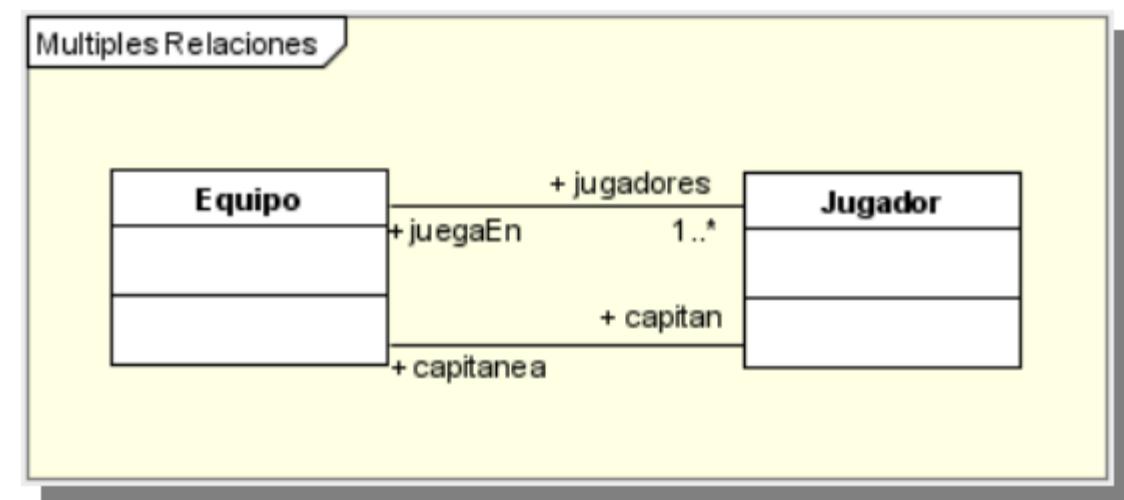
```
public class Diputado {
    public java.util.Collection voto = new java.util.TreeSet();
}

public class Ley {
    public java.util.Collection fueVotadaPor = new java.util.TreeSet();
}
```





* Relaciones

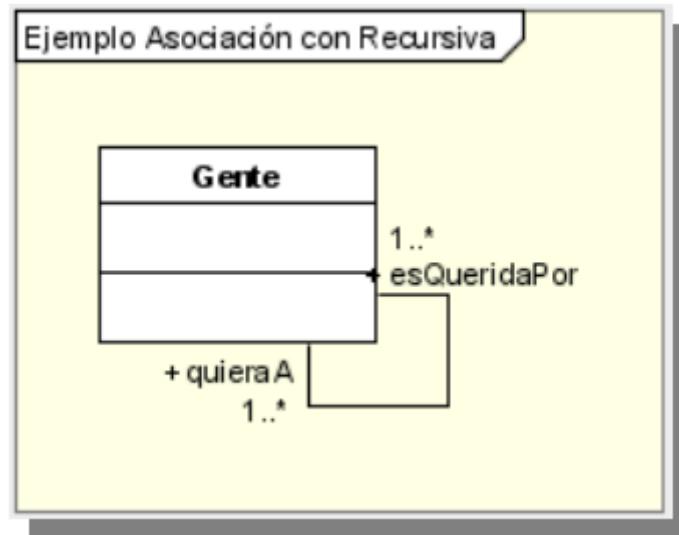


```
public class Equipo {  
  
    public Jugador capitana;  
  
    public java.util.Collection jugadores = new java.util.TreeSet();  
}  
  
public class Jugador {  
  
    public Equipo capitanaea;  
  
    public Equipo juegaEn;  
}
```





* Relación de recursividad



```
@public class Gente {

    public java.util.Collection quieraA = new java.util.TreeSet();

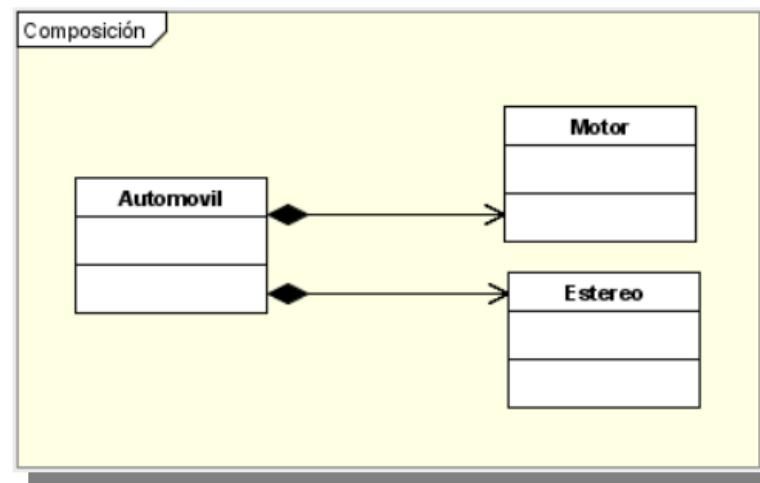
    public java.util.Collection esQueridaPor = new java.util.TreeSet();

}
```





Hay una dependencia en los ciclos de vida

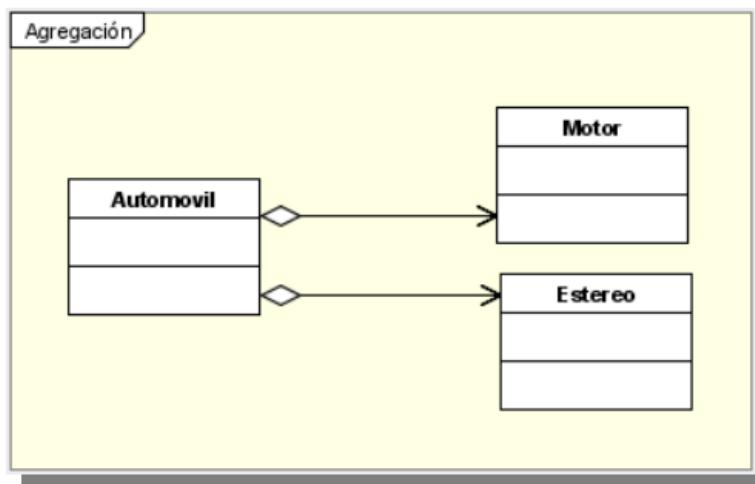


```
public class Automovil {  
    public Estereo estereo;  
    public Motor motor;  
  
    public Automovil() {  
        estereo = new Estereo();  
        motor = new Motor();  
    }  
}
```





Hay una dependencia en los ciclos de vida

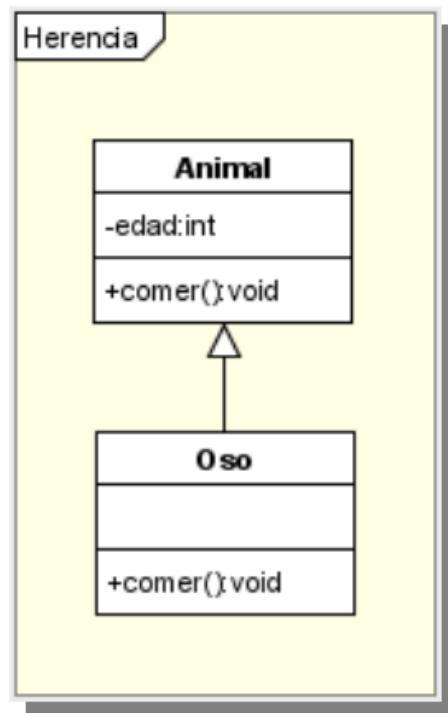


```
public class Automovil {
    public Estereo estereo;
    public Motor motor;

    public Automovil() {
    }

    public void ensamblar(Estereo e, Motor m) {
        estereo = e;
        motor = m;
    }
}
```





```
public class Animal {

    private int edad;

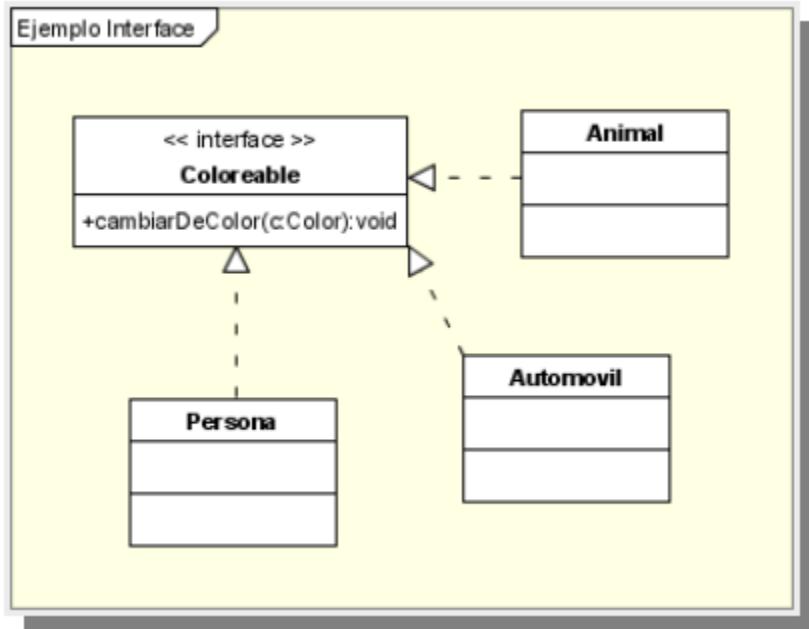
    public void comer() {
        // your code here
    }
}

public class Oso extends Animal {

    public void comer() {
        // Para el oso significa otra cosa...
    }
}
```

Según el lenguaje, puede ser necesario hacer explícito el override

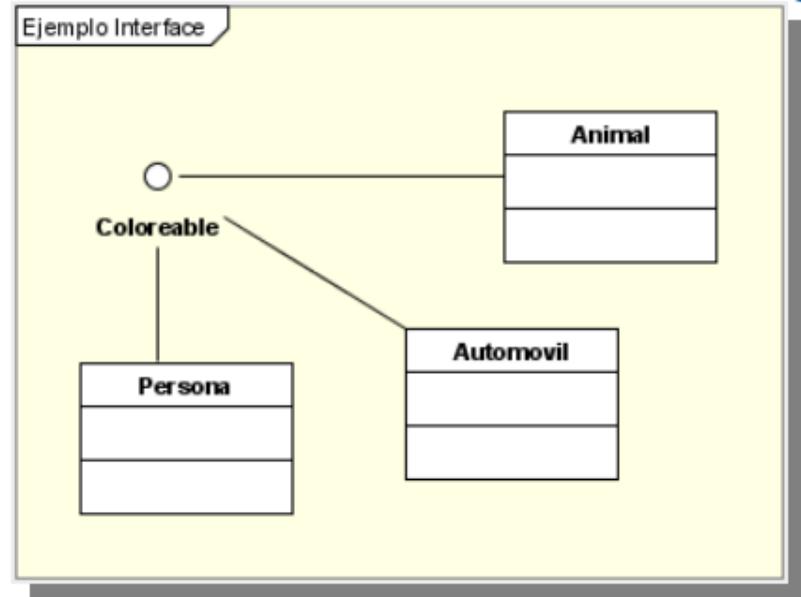




```

@public interface Coloreable {
    public void cambiarDeColor(Color c);
}

@public class Automovil implements Coloreable {
    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}
  
```



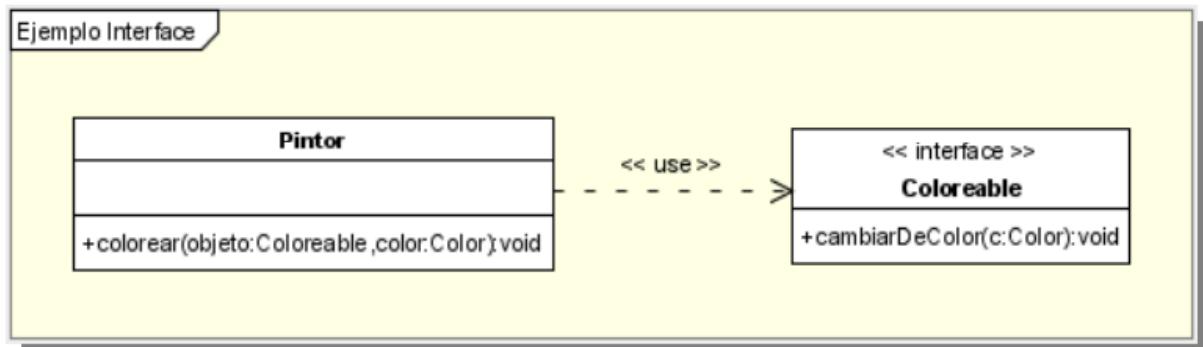
```

@public class Persona implements Coloreable {
    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}

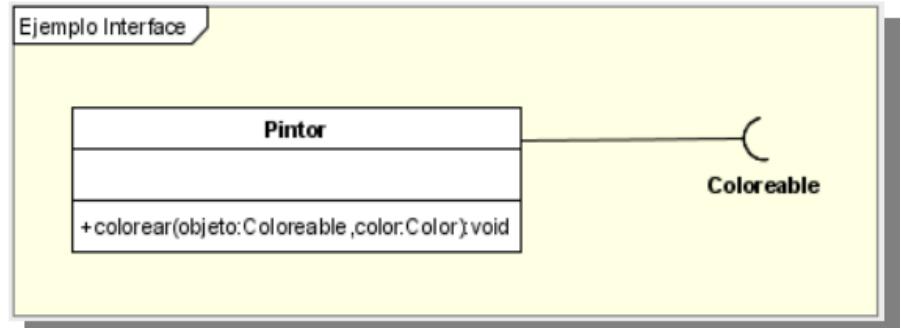
@public class Animal implements Coloreable {
    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}
  
```



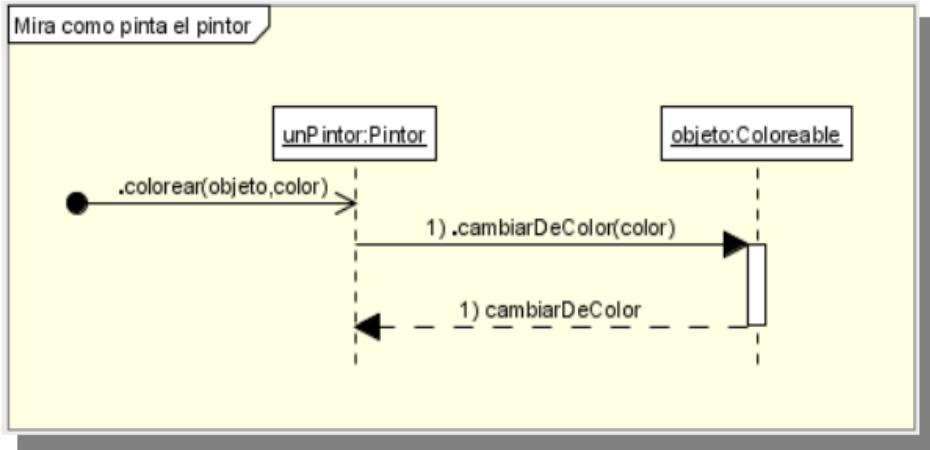
Ejemplo Interface

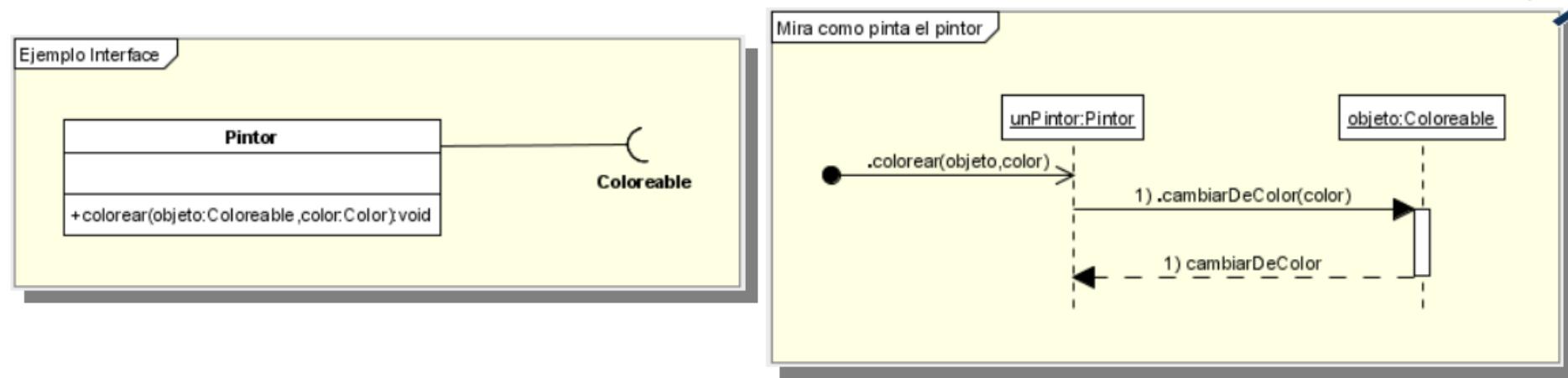


Ejemplo Interface



Mira como pinta el pintor





```

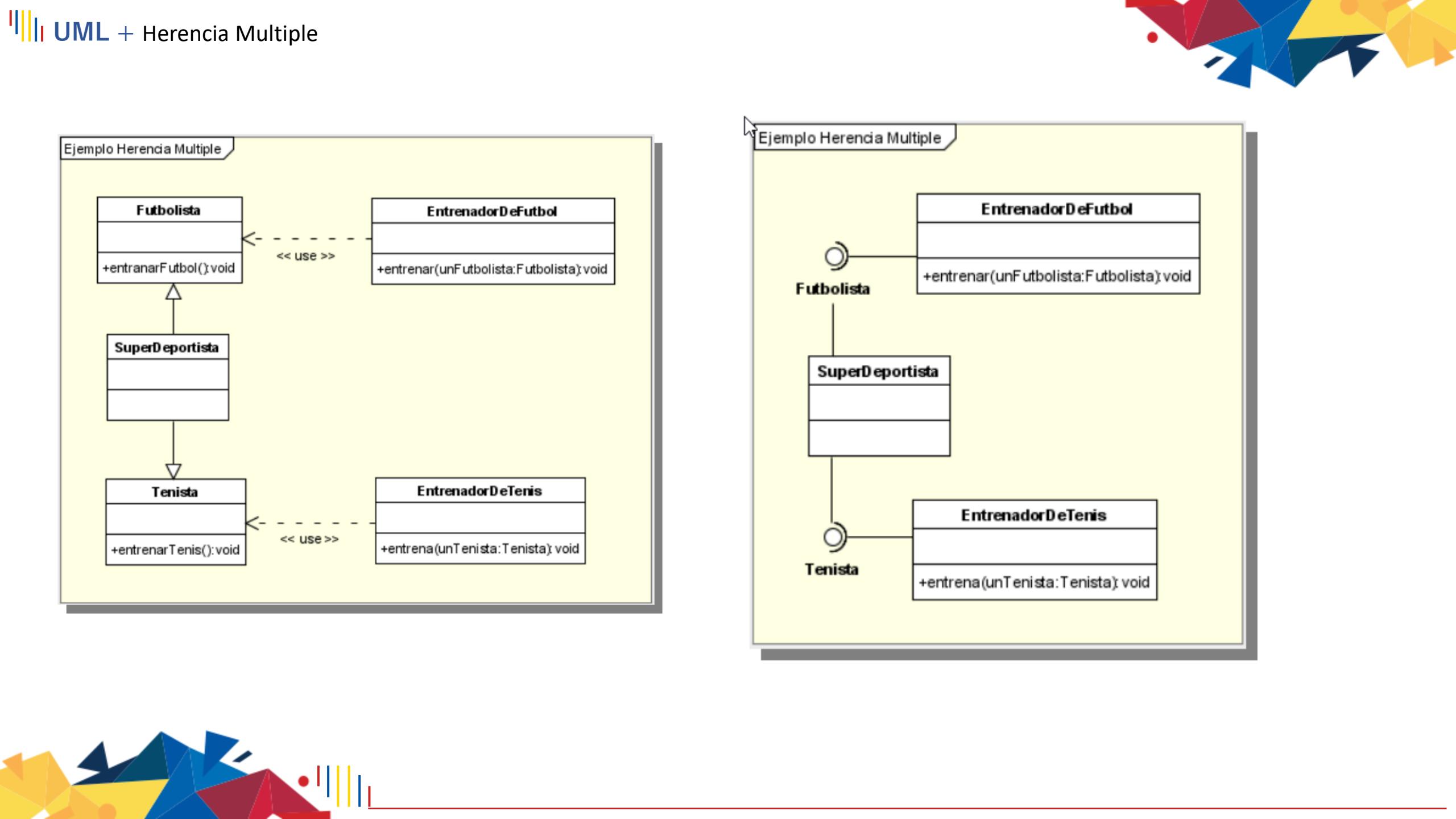
public class Pintor {

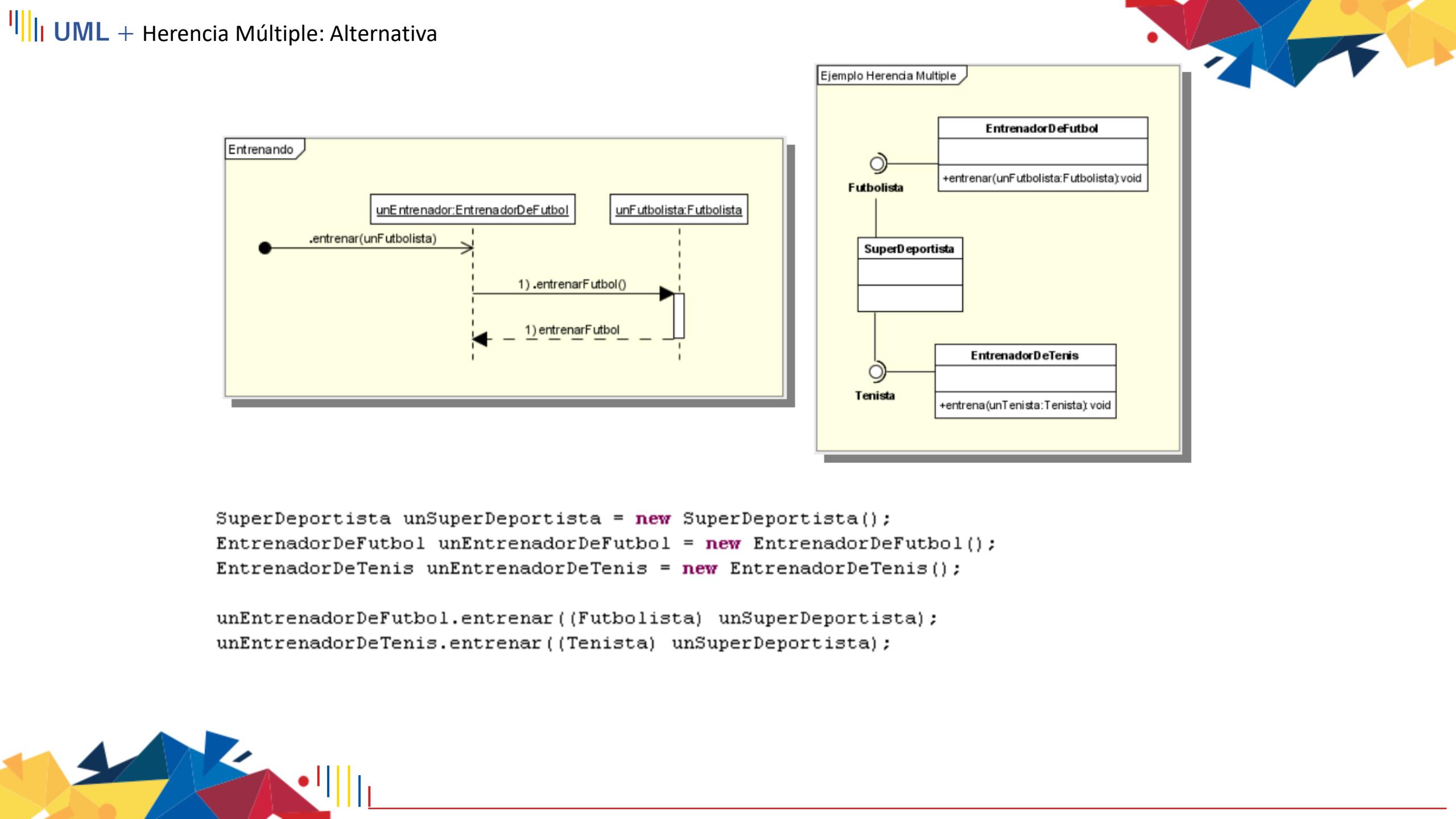
    public void colorear(Coloreable objeto, Color color) {
        objeto.cambiarDeColor(color);
    }
}

Persona unaPersona = new Persona();
Automovil unAutomovil = new Automovil();
Animal unAnimal = new Animal();
Color rojo = new Color();
Pintor unPintor = new Pintor();

unPintor.colorear((Coloreable) unaPersona, rojo);
unPintor.colorear((Coloreable) unAnimal, rojo);
unPintor.colorear((Coloreable) unAutomovil, rojo);
  
```



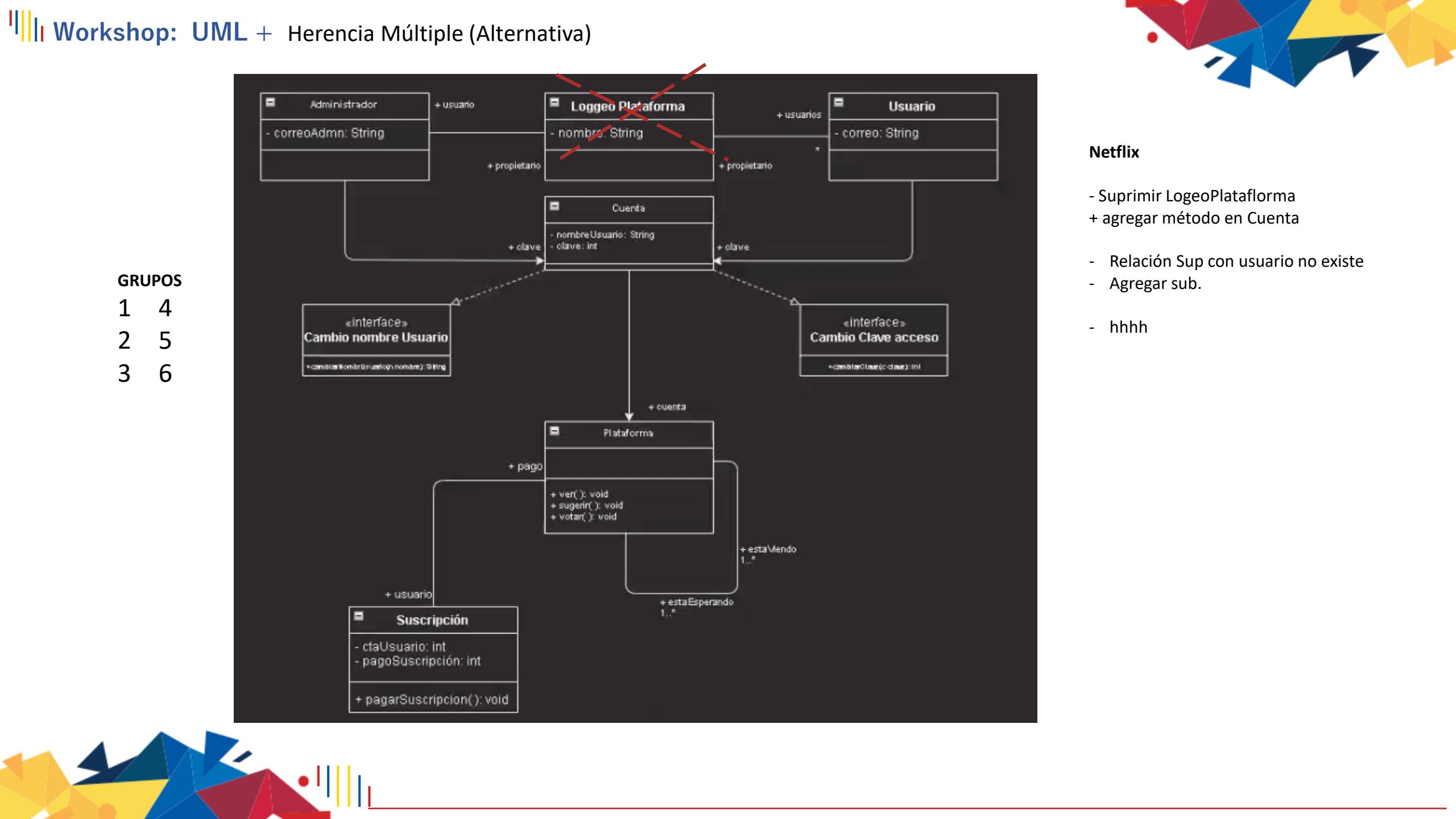




#HágalePues!



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE
ESCUELA POLITÉCNICA NACIONAL

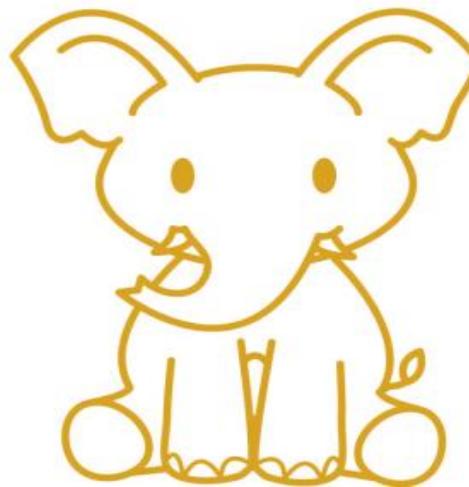


#HágalePues!



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE
ESCUELA POLITÉCNICA NACIONAL

PROHIBIDO LOS ...



TROMPUDOS



BRAVOS



< x%!.../>



#ComoTas!



SIEMPRE SONRÍE
así no quieras



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE
ESCUELA POLITÉCNICA NACIONAL



rubber duck debugging



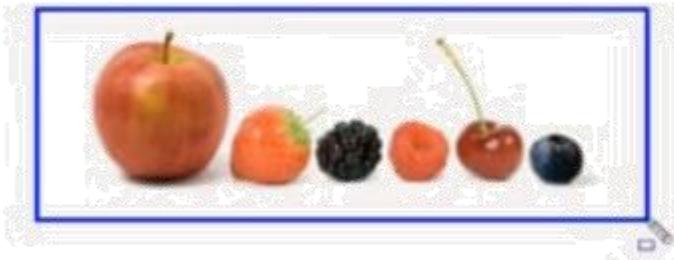


Yo
Implementando
rubber duck debugging

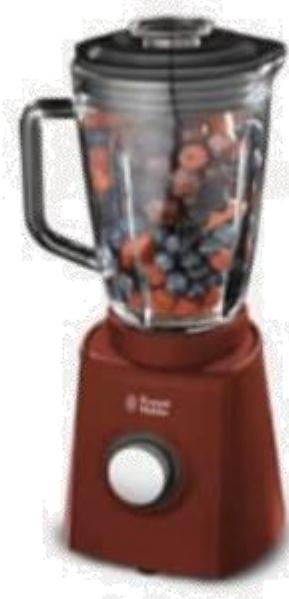




Métodos



parámetros



métodos





```
// instanciar (new): → Clase (var) → objeto : nombre(var)
```

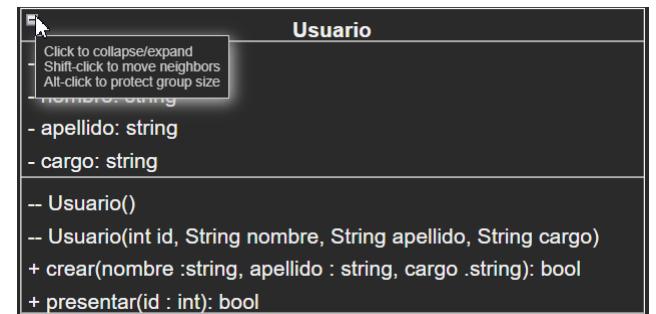
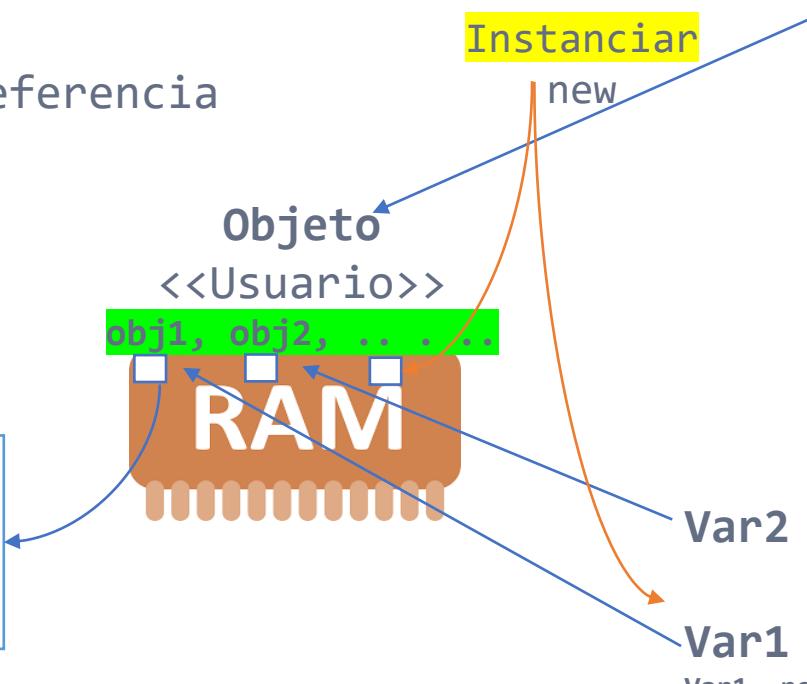
```
Usuario obj1; // declarando y referencia a un objeto
    obj1 = new Usuario(); // instancia --> constructor
```

```
Usuario obj2; // variable
obj2 = new Usuario(1, "pepe", "alimaña", "biólogo");
    // instancia --> constructor + sobrecargado
```

```
Obj2._nombre = "abc";
```

```
new Usuario(); // objeto sin referencia
```

Obj1 : Usuario(pepe)
`_id = -1
 _nombre = ""
 _apellido = ""
 _cargo = "xxxx"`



```
public class Usuario {
    // Autor: david.calahorrano@epn.edu.ec
    private int _id;
    public String _nombre;
    private String _apellido;
    private String _cargo;

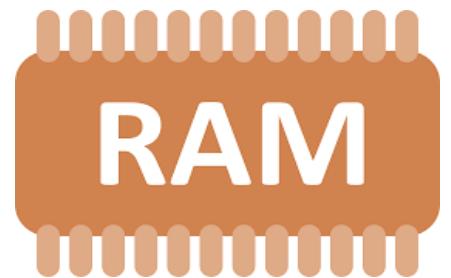
    // constructor + sobrecarga
    public Usuario() { ... }
    public Usuario(int id) { ... }
    public Usuario(String nombre) { ... }
    public Usuario(int id, String nombre, String apellido,
    cargo) { ... }

    public boolean _presentar() {
        //this. Usuario();
        System.out.println("Datos del usuario:");
        System.out.println("_id:" + this._id);
        System.out.println("_nombre :" + this._nombre);
        System.out.println("_apellido:" + this._apellido);
        System.out.println("_cargo :" + this._cargo);
        return true;
    }
}
```





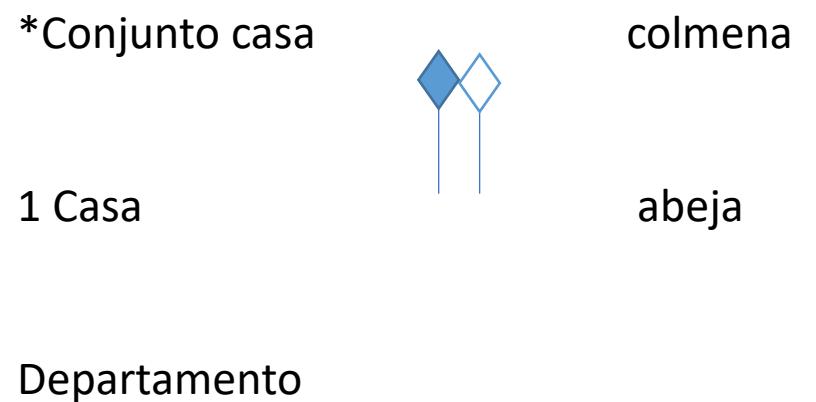
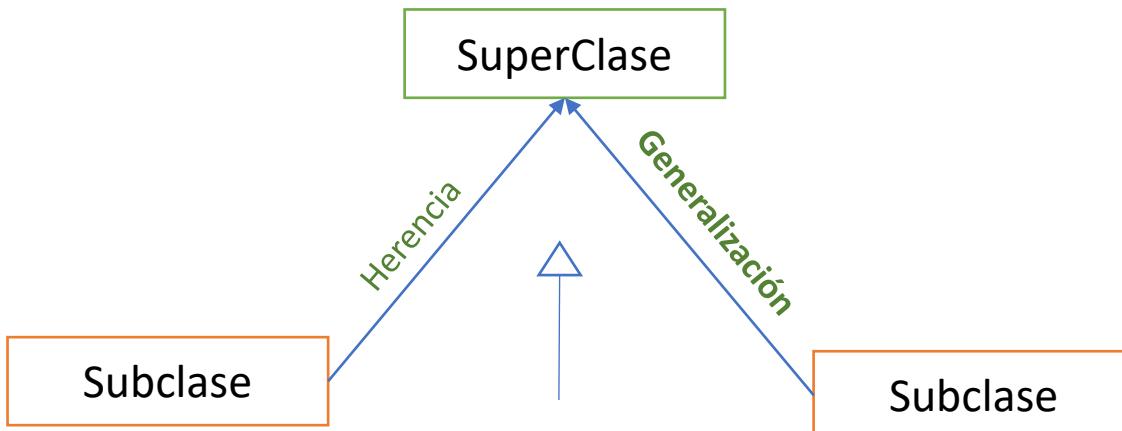
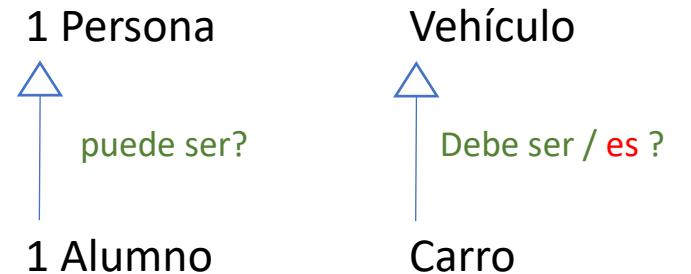
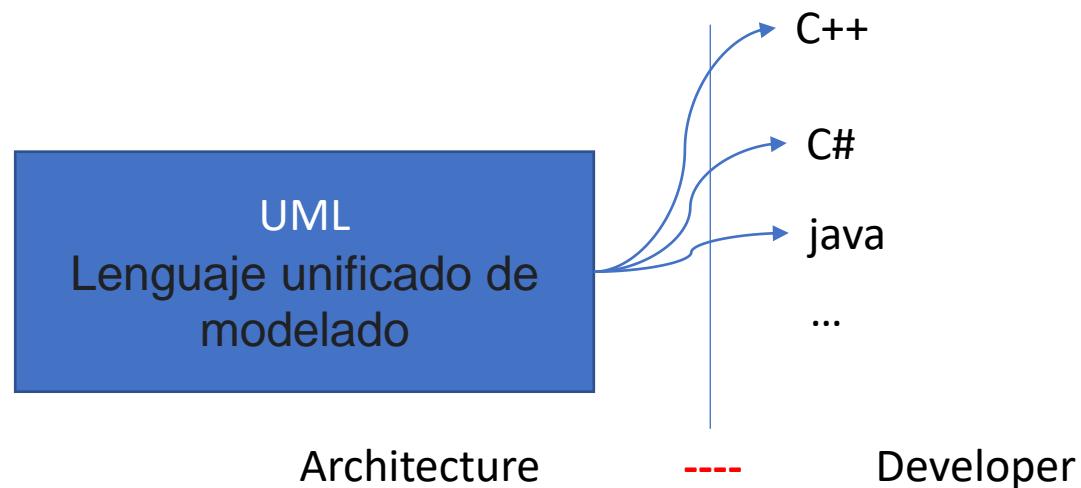
```
new Usuario("ana");  
  
Usuario obj = new Usuario("ana");  
obj = new Usuario("ana");  
obj._nombre = "maria";
```



```
Obj : Usuario(pepe)  
_id = -1  
_nombre = "ana"  
_apellido = ""  
_cargo = ""
```

```
Obj : Usuario(pepe)  
_id = -1  
_nombre = "maria"  
_apellido = ""  
_cargo = ""
```







La interfaz **nos permite especificar un conjunto de operaciones en clase que pueden ser utilizada por otras.**

La interface difiere de una clase en que no tendrá atributos debido a que solo será un conjunto de operaciones

Clase

En clase, puede crear instancias de variable y crear un objeto.

La clase puede contener métodos concretos (con implementación)

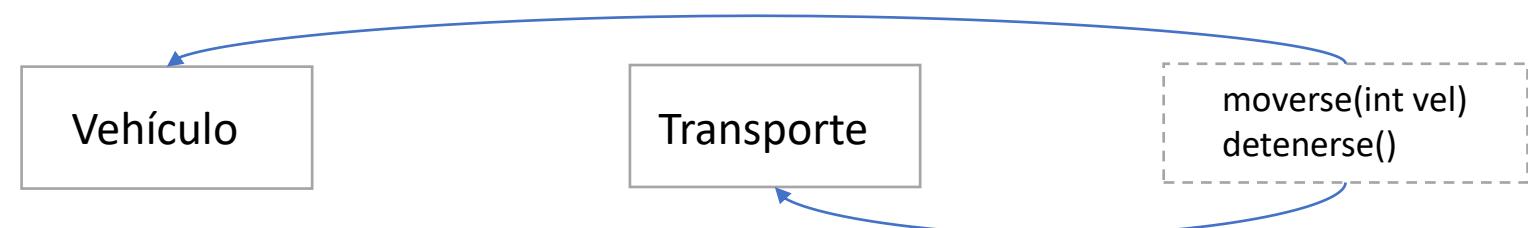
Los especificadores de acceso utilizados con las **clases** son privados, protegidos y públicos.

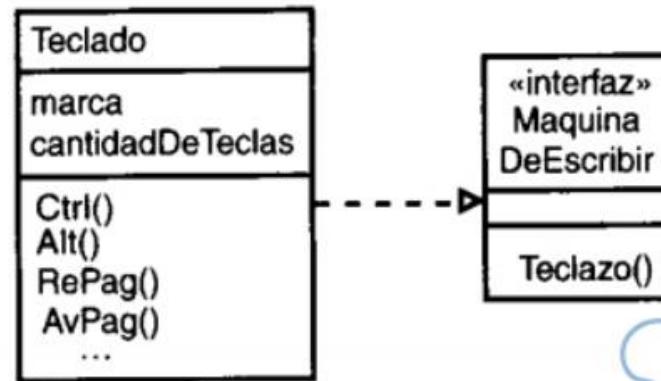
Interfaz

En una interfaz, no puede crear instancias de variables y crear un objeto.

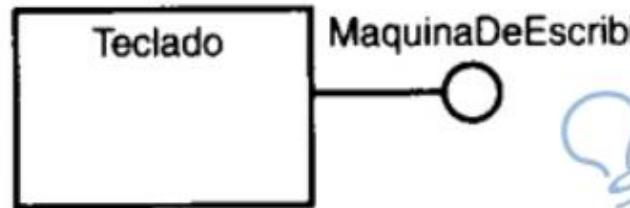
La interfaz no puede contener **métodos concretos** (con implementación)

En la interfaz solo se utiliza un especificador: público.

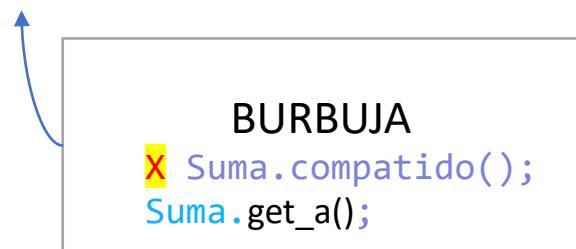
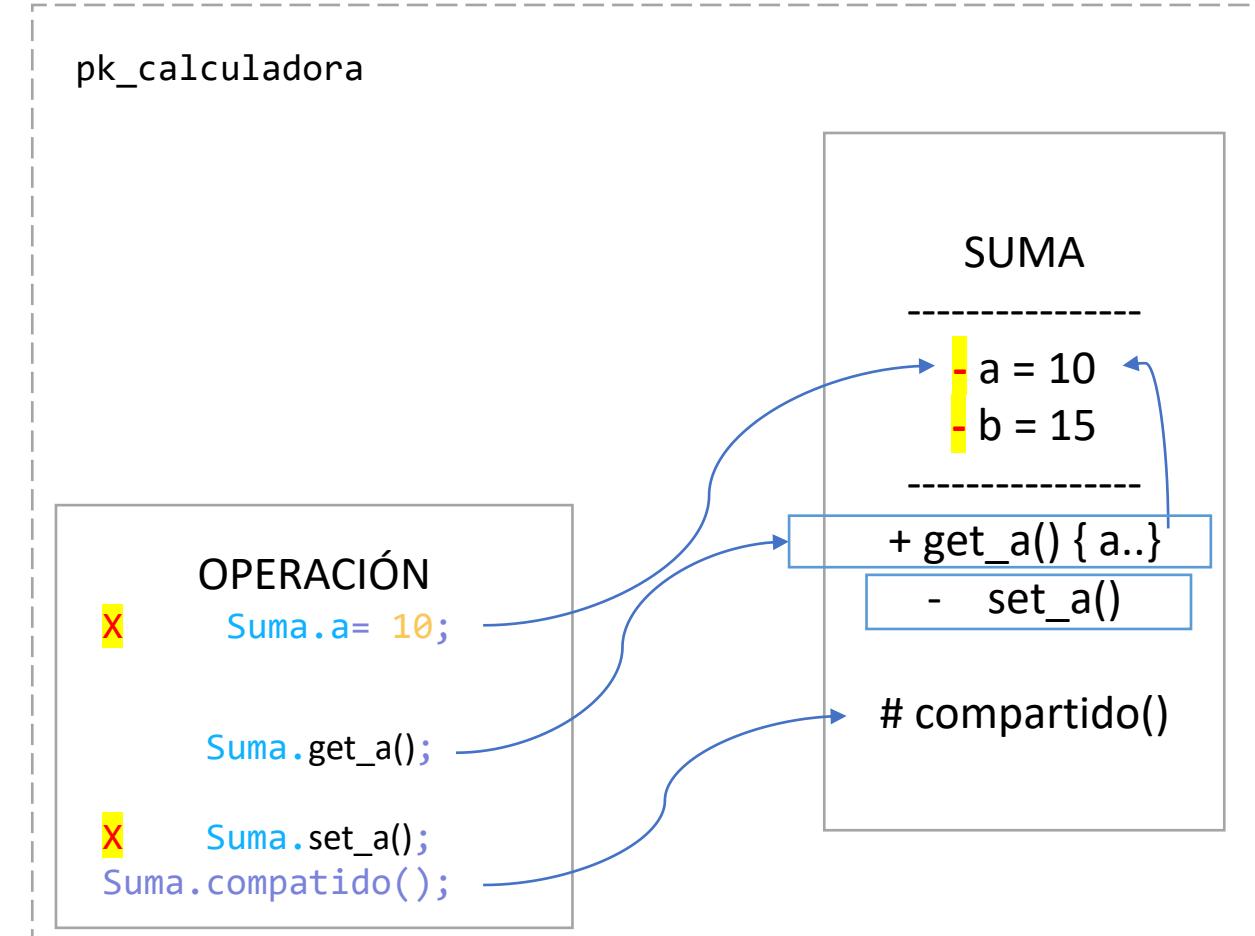




 solvetic.com



 solvetic.com



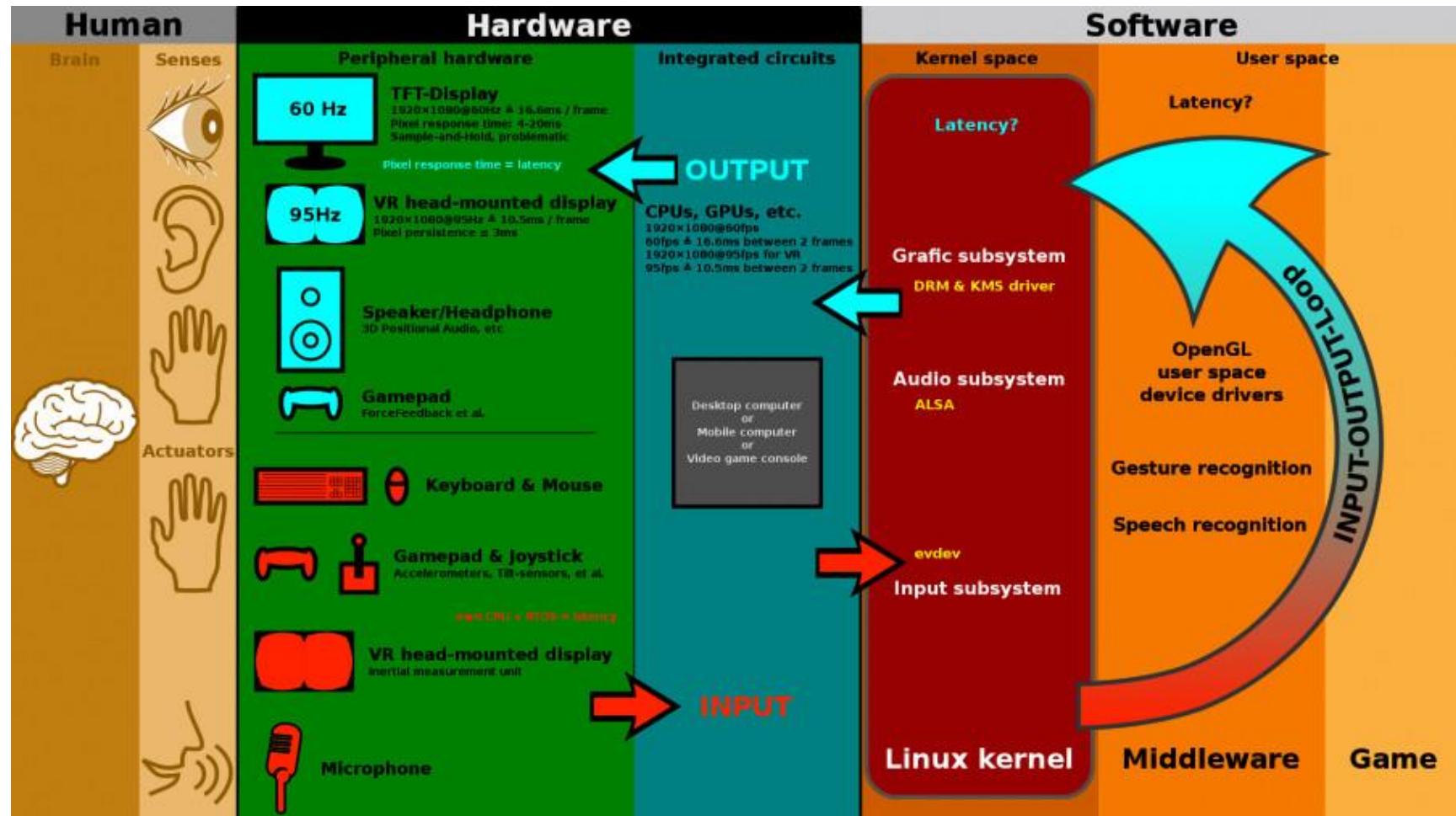


- POO
 - > paradigma de programación
 - > template (propiedad/atributo, métodos, mensajes/eventos)
 - > instancia de una clase
 - un conjunto de datos relacionados que identifican el **estado actual** del objeto
 - un conjunto de *comportamientos* = *métodos*.
- constructores -> método que se ejecuta 1 vez en la instancia de la clase
- accesibilidad -> private, protected, public
- encapsulamiento -> métodos/clase + **accesibilidad**
- instancia -> crear objetos (new)
- Sobre carga -> si 2 o + métodos en la misma clase que utilizan el mismo nombre pero con distintos parámetros o tipos de parámetros/datos
- Herencia -> relación de jerarquía
- Instancia





User interface UI es el espacio donde se producen las interacciones entre seres humanos y máquinas. El objetivo de esta interacción es permitir el funcionamiento y control más efectivo de la máquina desde la interacción con el humano.





Las interfaces de usuario pueden clasificarse según su **tipo**:

- **Interfaz de línea de comandos** (Command-Line Interface, CLI): Interfaces alfanuméricas (intérpretes de comandos) que solo presentan texto.
- **Interfaces gráficas de usuario** (Graphic User Interface, GUI): Permiten comunicarse con la computadora de forma rápida e intuitiva representando gráficamente los elementos de control y medida.
- **Interfaz natural de usuario** (Natural User Interface, NUI): Pueden ser táctiles (teléfonos o tablets), pueden funcionar mediante reconocimiento del habla (Siri o Alexa) o mediante movimientos corporales (Kinect o Wii).

o según su construcción:

- **Interfaz física o hardware**: teclados, ratones, pantallas, pulsadores, sensores, etc.
- **Interfaz lógica o software**: programas o parte de programa que permiten expresar las órdenes a la computadora o visualizar su respuesta (vistas, menús, ventanas, formularios, etc.)





La **experiencia de usuario o user experience UX** y la usabilidad, son conceptos que a menudo se confunden en uno solo pero que tienen matices distintos.

La **usabilidad** hace referencia a la calidad de la página web o del programa informático que son sencillos de usar porque facilitan la lectura de los textos, descargan rápidamente la información y presentan funciones y menús sencillos, por lo que el usuario encuentra satisfechas sus consultas y cómodo su uso.

La **experiencia de usuario o UX** hace referencia al aspecto emocional del usuario, esto es, cómo le hace sentir esa experiencia.

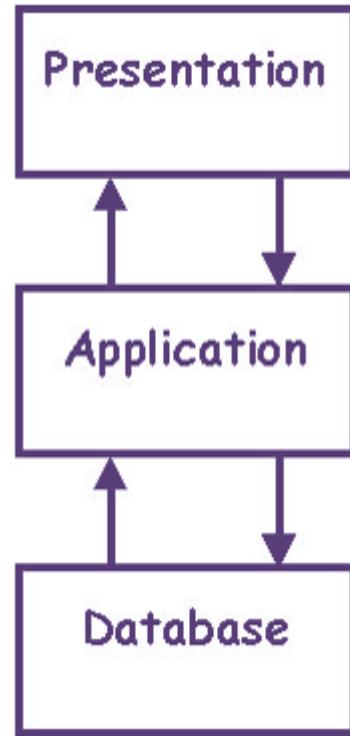
Ambas son fundamentales y deben ir de la mano a la hora de diseñar cualquier tipo de interfaz.





- User interface programming in the user's computer
- Business logic in a more centralized computer, and
- Required data in a computer that manages a database.

Advantages and Disadvantages of Multi-Tier Architectures

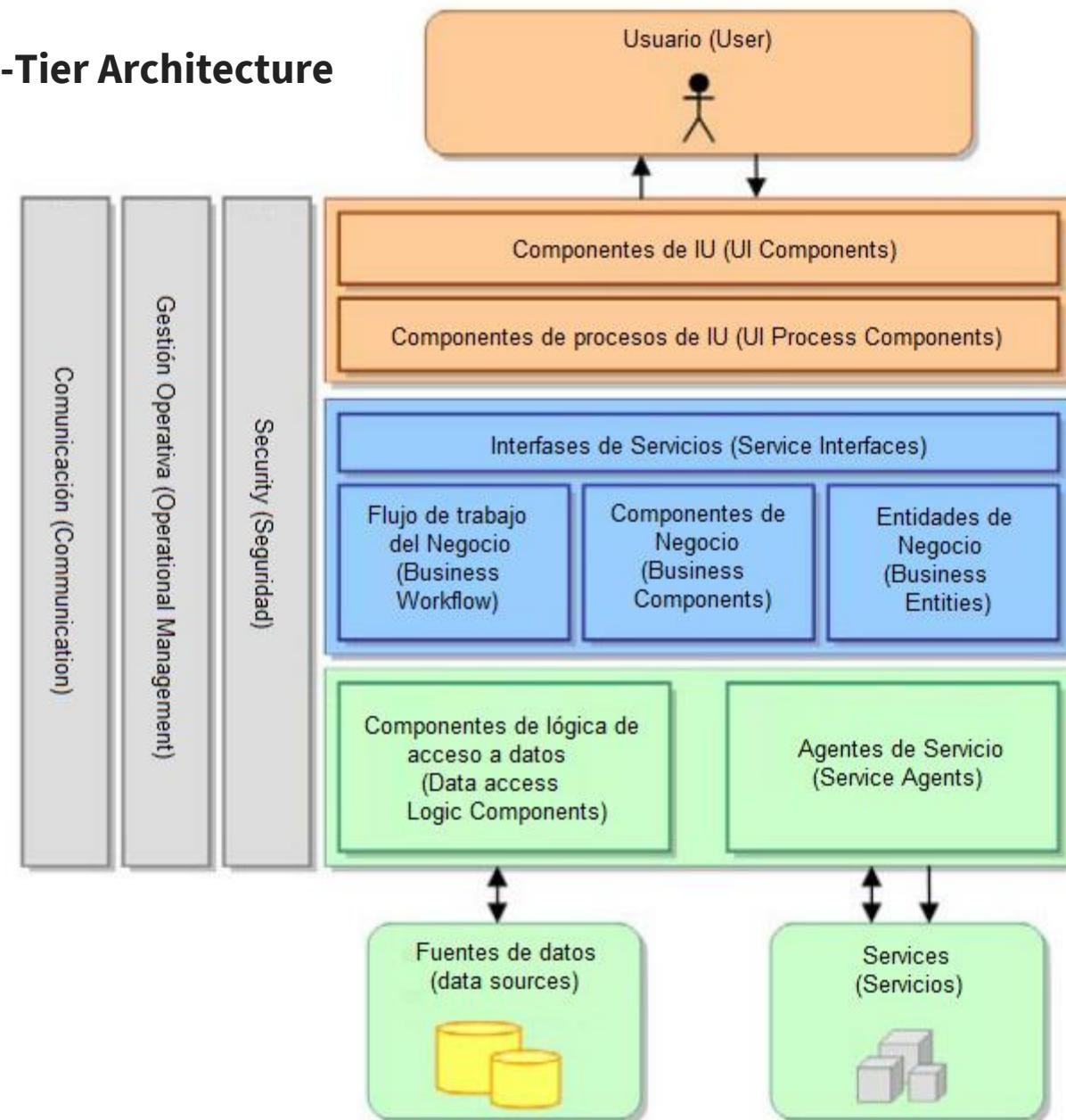


| Advantages | Disadvantages |
|-------------------------|--------------------------|
| • Scalability | • Increase in Effort |
| • Data Integrity | • Increase in Complexity |
| • Reusability | |
| • Reduced Distribution | |
| • Improved Security | |
| • Improved Availability | |



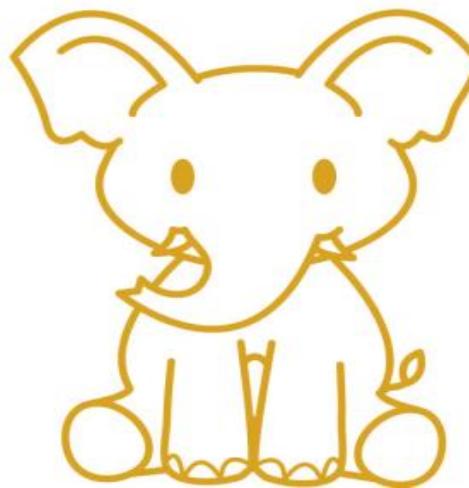


N Tier(Multi-Tier), 3-Tier, 2-Tier Architecture





PROHIBIDO LOS ...



TROMPUDOS



BRAVOS



< x%!.../>





- **ARREGLOS Y OBJETOS ARRAYLIST:** ArrayList, Vector
 - Manejo de iteradores, librerías comunes
 - Linked List
-
- **N-TIER Architecture**
 - MER: Modelado entidad relación
 - Manejo de Archivos & Base de datos









Manejo de iteradores



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE



Manejo de iteradores



Patricio Michael Paccha Angamarca
MAGISTER EN INGENIERÍA DE SOFTWARE







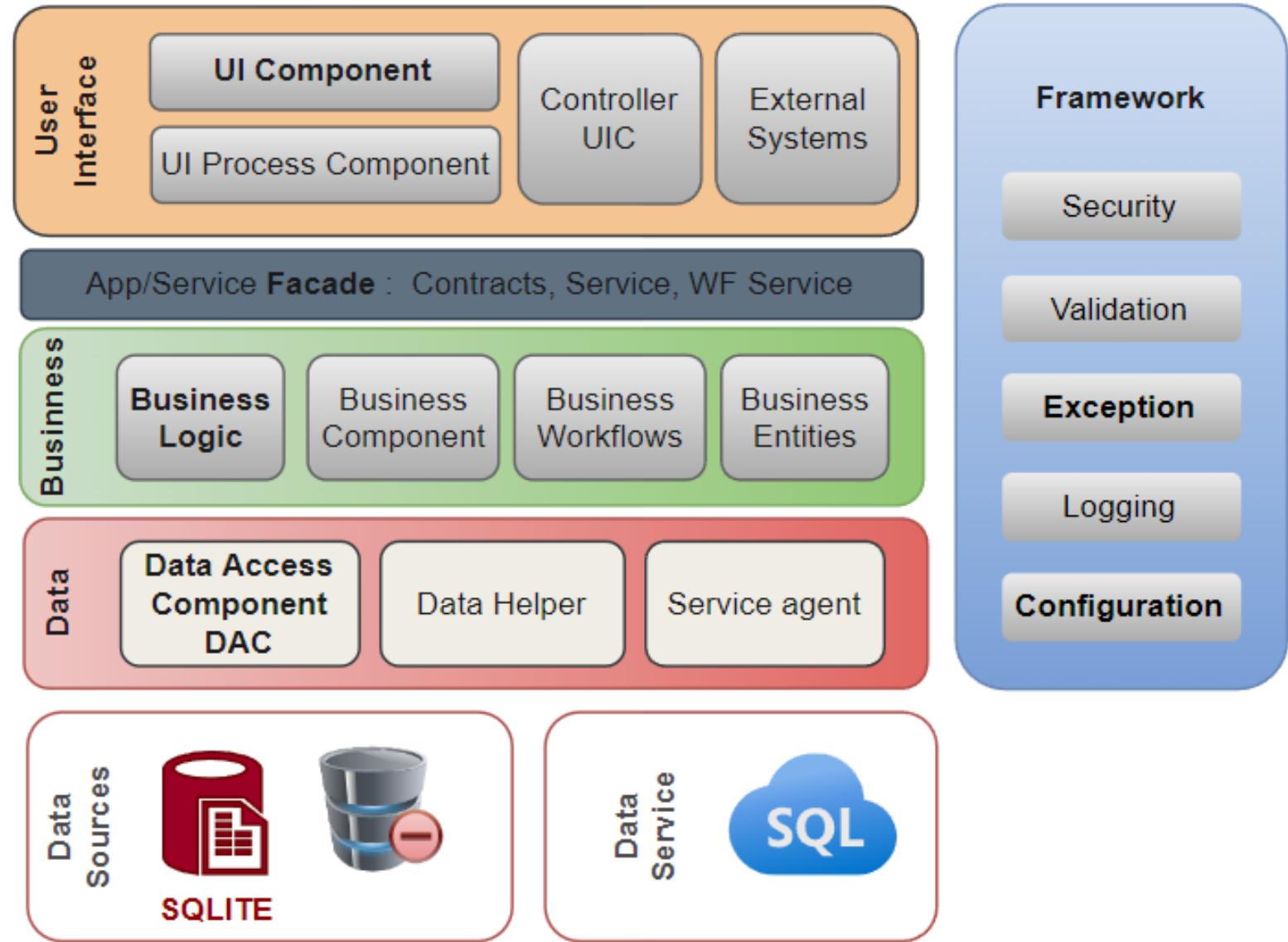


ArrayList, List, Vector



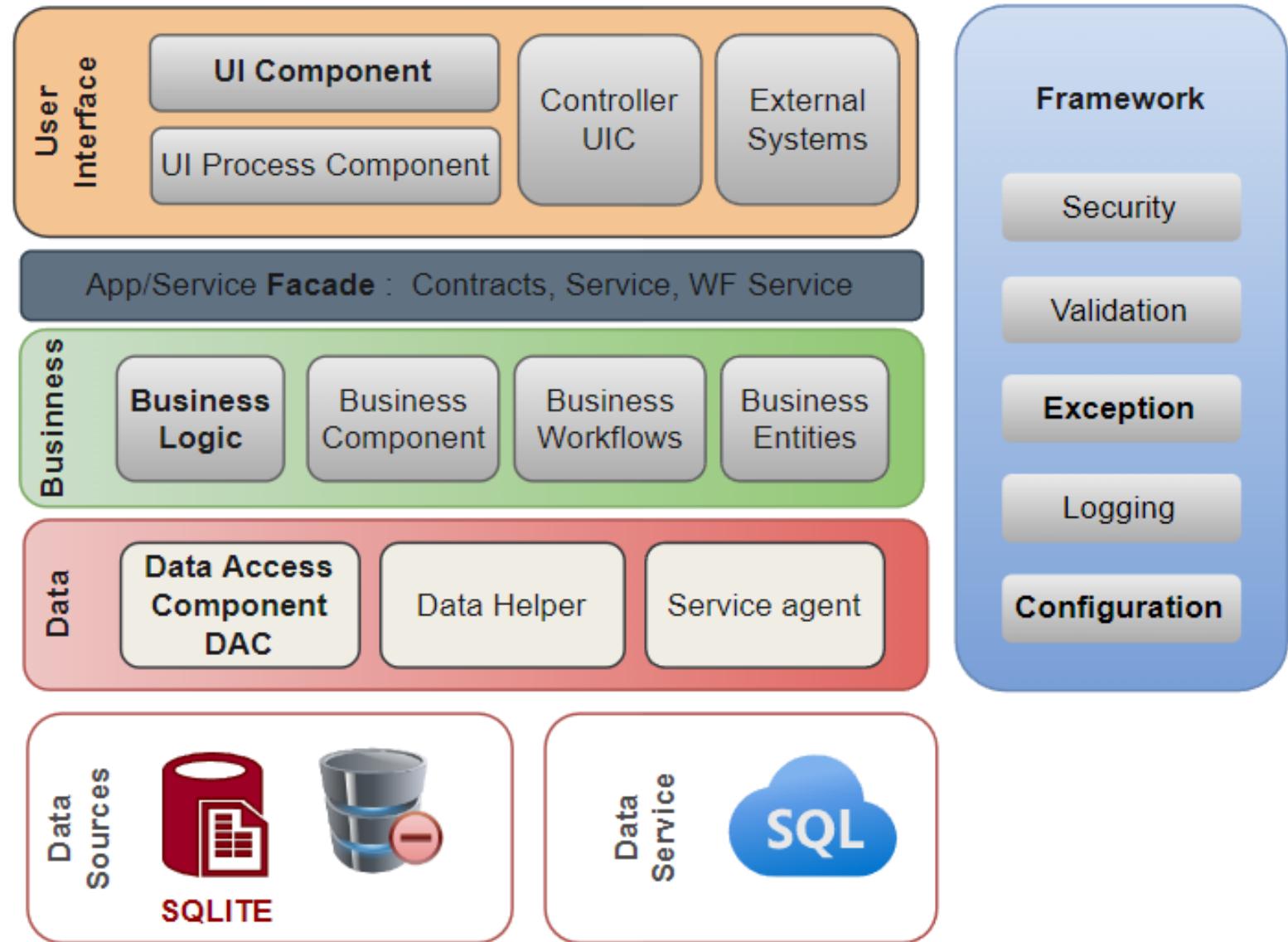


TinderPet



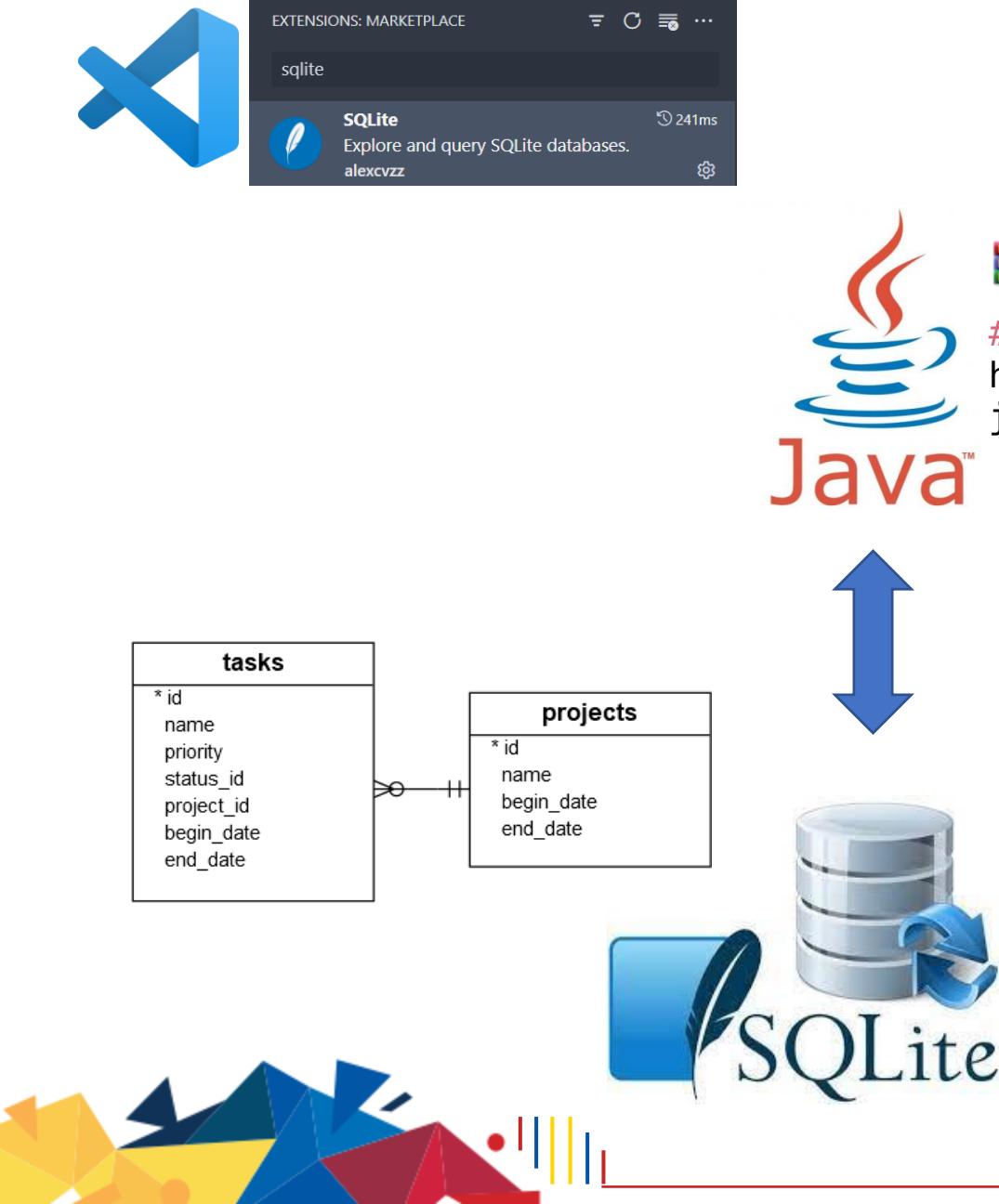


TinderPet





N-TIER Architecture : Data Source



EXTENSIONS: MARKETPLACE
sqlite
SQLite Explore and query SQLite databases.
alexcvzz

sqlite-jdbc-3.8.11.2.jar

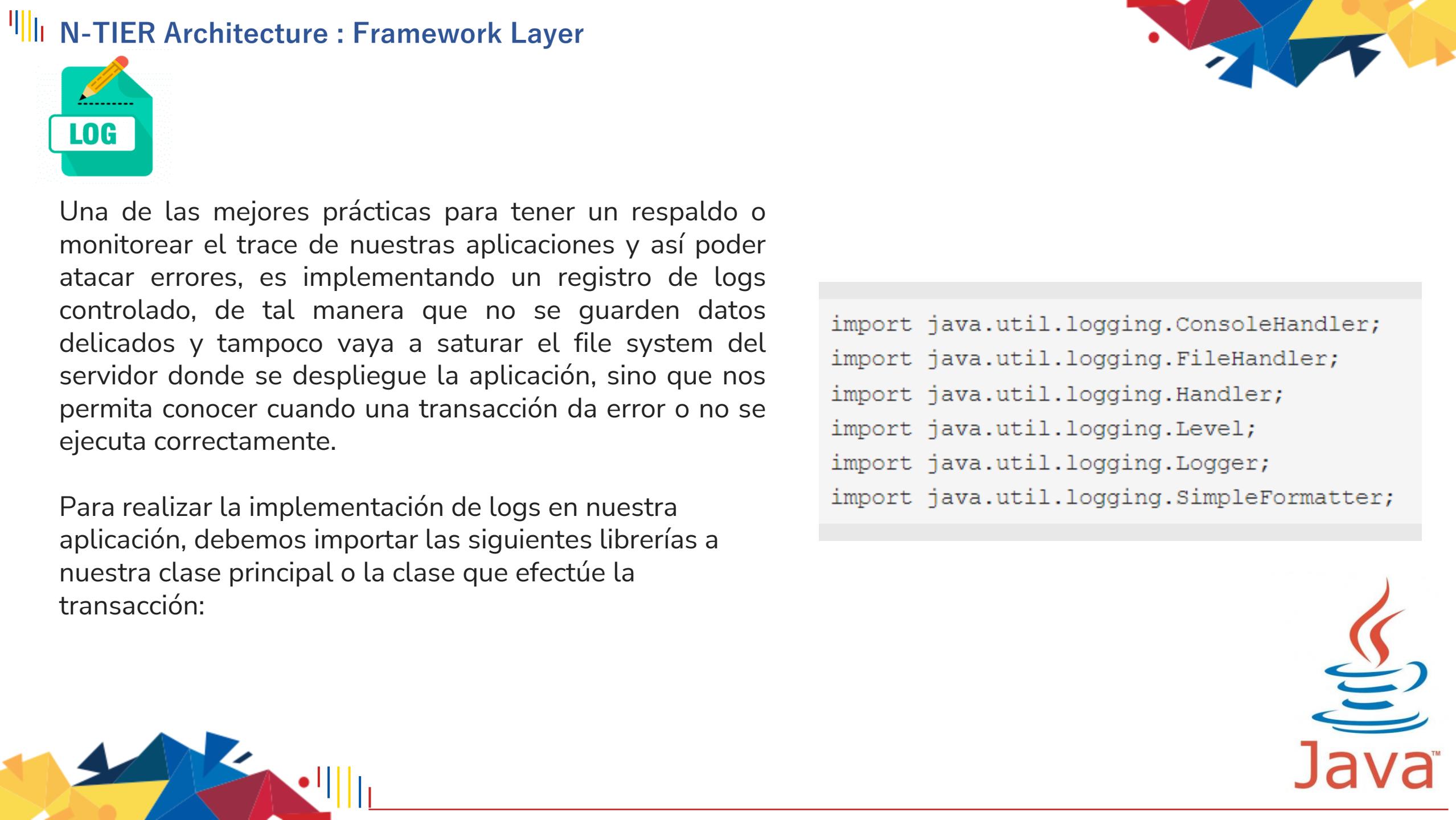
Download sqlite-jdbc JAR 3.8.11.2 with all dependencies
<https://jar-download.com/artifacts/org.xerial/sqlite-jdbc/3.8.11.2/source-code>

| STORAGE CLASS | USES |
|---------------|---|
| NULL | Value is a null value. |
| INTEGER | Value is a signed integer. Depending on the magnitude of the value, it can be stored in 1, 2, 3, 4, 6, or 8 bytes. |
| REAL | Value is a floating point value. Stored as an 8-byte IEEE floating point number. |
| text | Value is a text string. Stored using the database encoding (utf-8, utf-16be or utf-16le) |
| BLOB | Value is a blob of data. Stored exactly as it was input. |



Cuando se está preparando un programa para un entorno de producción tener un log donde se reporten los eventos y errores puede ser la diferencia entre pasar una semana tratando de replicar un error o solo leer un archivo y saber en que línea ocurrió el error y si bien hay todo un mundo de librerías y frameworks para este propósito no hay que olvidar que el propio Java ya contiene las clases para hacer esto y que nunca esta de mas ahorrarse dependencias.





N-TIER Architecture : Framework Layer



Una de las mejores prácticas para tener un respaldo o monitorear el trace de nuestras aplicaciones y así poder atacar errores, es implementando un registro de logs controlado, de tal manera que no se guarden datos delicados y tampoco vaya a saturar el file system del servidor donde se despliegue la aplicación, sino que nos permita conocer cuando una transacción da error o no se ejecuta correctamente.

Para realizar la implementación de logs en nuestra aplicación, debemos importar las siguientes librerías a nuestra clase principal o la clase que efectúe la transacción:

```
import java.util.logging.ConsoleHandler;  
import java.util.logging.FileHandler;  
import java.util.logging.Handler;  
import java.util.logging.Level;  
import java.util.logging.Logger;  
import java.util.logging.SimpleFormatter;
```





- JFrame
- JPanel
- JButton
- JLabel

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

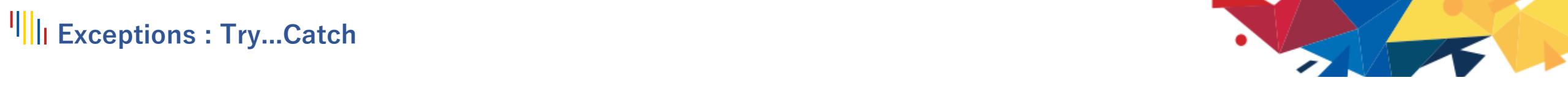
try{} catch{} finally{}

GUI

- Agrega *.jar a tu aplicación
<https://www.formdev.com/flatlaf/>
<https://www.formdev.com/flatlaf/themes>
<https://www.youtube.com/watch?v=Gxf4T-4lx-w&t=260s>

- Errores Personalizados





Exceptions : Try...Catch

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

Finally / throw

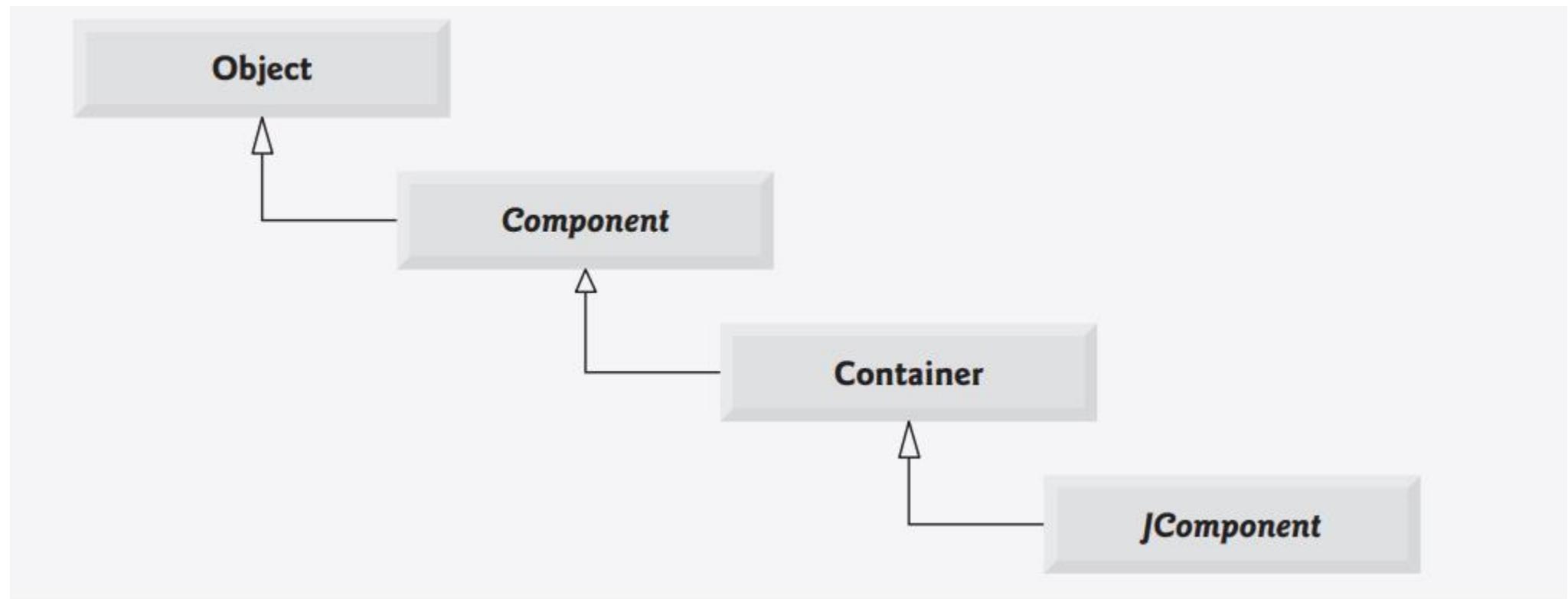
```
try {  
    /* do code */  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```







javax.swing.JOptionPane





Ingresa tu usuario > Ingresa tu contraseña

Ingresa tu contraseña

Tuvimos un problema

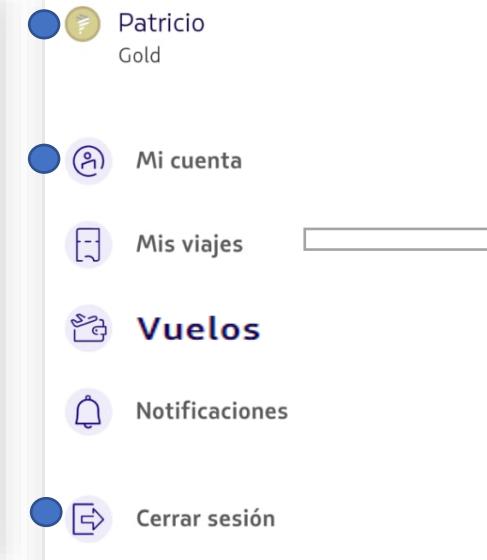
No pudimos iniciar sesión. Espera unos minutos e inténtalo nuevamente.

Usuario: 911036354456

Contraseña:

.....

Reintentar



Grupo

Grupo 3

Vuelos más buscados

Ida 01/02/23

Desde Quito a

Manta

Solo ida

Economy

Precio final desde

USD 38,25

Tasas incluidas - Vuelo directo



Ida 01/02/23

Desde Guayaquil a

Quito

Solo ida

Economy

Precio final desde

USD 36,24

Tasas incluidas - Vuelo directo



Itinerario de vuelo

Salida

GYE 6:00
Jj De Olmedo
Intl.

Duración

50 min

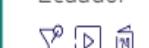
Llegada

UIO 6:50
Mariscal Sucre
Intl.



Airbus A319

Ecuador



Grupo 5 , 6



Patricio
Gold

Mi cuenta

Mis viajes

Vuelos

Notificaciones

Cerrar sesión

Grupo 4, 2

Ida y Vuelta ▾ Economy ▾ 1 pasajero ▾ Usar Millas LATAM Pass

Guayaquil, GYE - Ec Quito, UIO - Ecuad. Ida jue. 2 mar Vuelta jue. 13 abr **Buscar**

Elige un vuelo de ida

Ordenado por: Recomendado ▾
El orden aplicará para tu vuelo de ida y vuelta.

| Recomendado | Más rápido |
|---|---------------------------------------|
| 6:00 GYE Duración 0 h 50 min Directo | 6:50 UIO Adulto desde USD 85,52 |
| 7:05 GYE Duración 0 h 50 min Directo | 7:55 UIO Adulto desde USD 50,80 |



Patricio
Gold

Mi cuenta

Mis viajes

Vuelos

Notificaciones

Cerrar sesión



6:00 10/02/23 20:25 Guayaquil GYE
Aeropuerto Jj De Olmedo Intl. | ↗ 1

Lima LIM · Escala de 1 h 30 min

6:50 11/02/23 6:05 Buenos Aires EZE
Aeropuerto Ezeiza Intl.

En este vuelo puedes llevar:

- 1 bolso o mochila pequeña
- 1 equipaje de mano (10 kg)
- 1 equipaje de bodega (23 kg)

Grupo 1





GRACIAS...!

PROGRAMACIÓN II

Patricio Michael Paccha Angamarca
Magister en ingeniería de software

