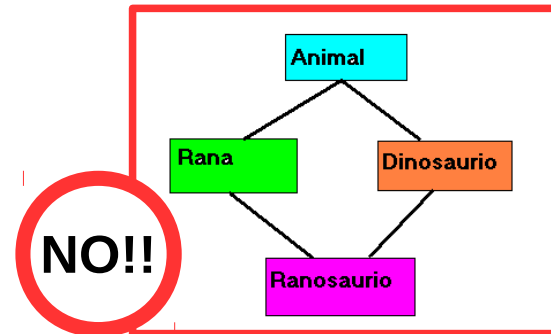
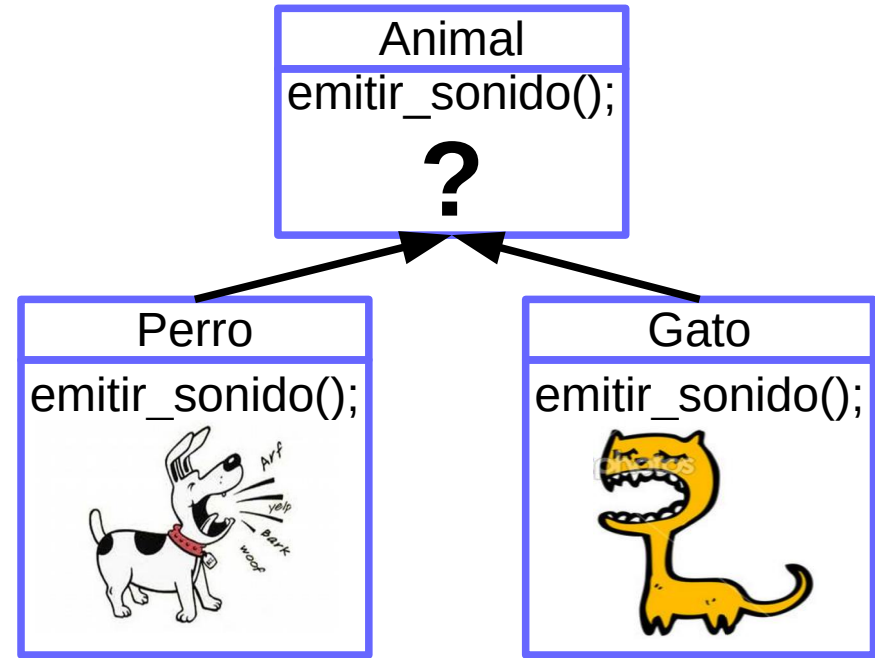


# CLASE 3 – HERENCIA. CLASES ABSTRACTAS

- Concepto de Herencia.
- Sobreescritura y sobrecarga de métodos.
- Ocultamiento de variables
- La palabra "super".
- Constructores en Clases Extendidas
- Herencia y redefinición de miembros
- Especificadores de acceso.
- Métodos y clases "final".
- Concepto de Delegación
- Herencia vs. Delegación.
- Clases y métodos abstractos.
- Diseñando una clase para que sea extensible.
- La clase Object: método toString.



El concepto de herencia refiere al hecho de transmitir “algo” desde un organismo a otro.

Supongamos que María tiene un hijo entonces esperamos que el hijo de María, herede “cosas” de ella.

¿Qué cosas esperamos que herede?

Rasgos Físicos

Color de Piel

Nacionalidad

Color de Ojos

Apellido

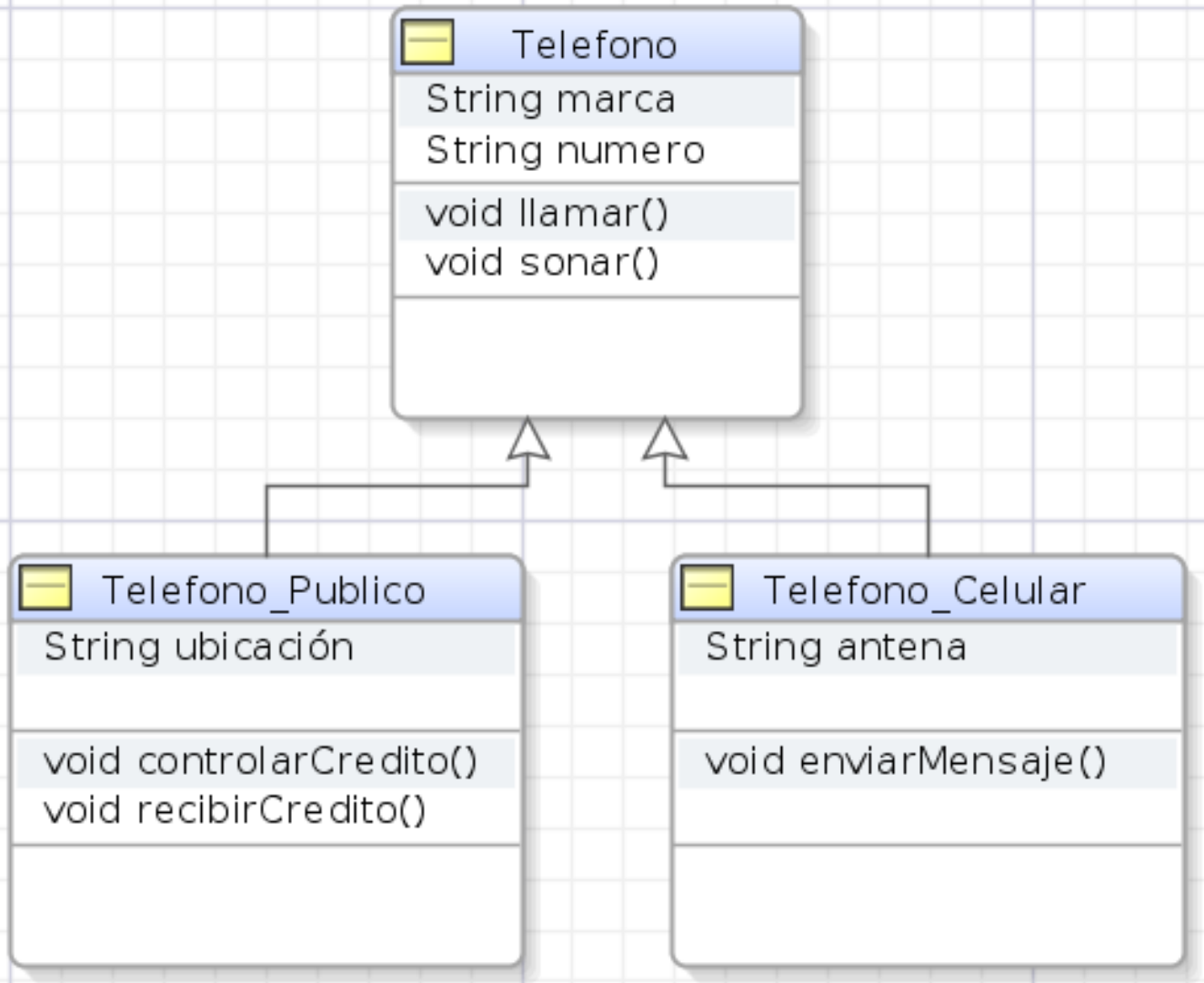


# Concepto de Herencia

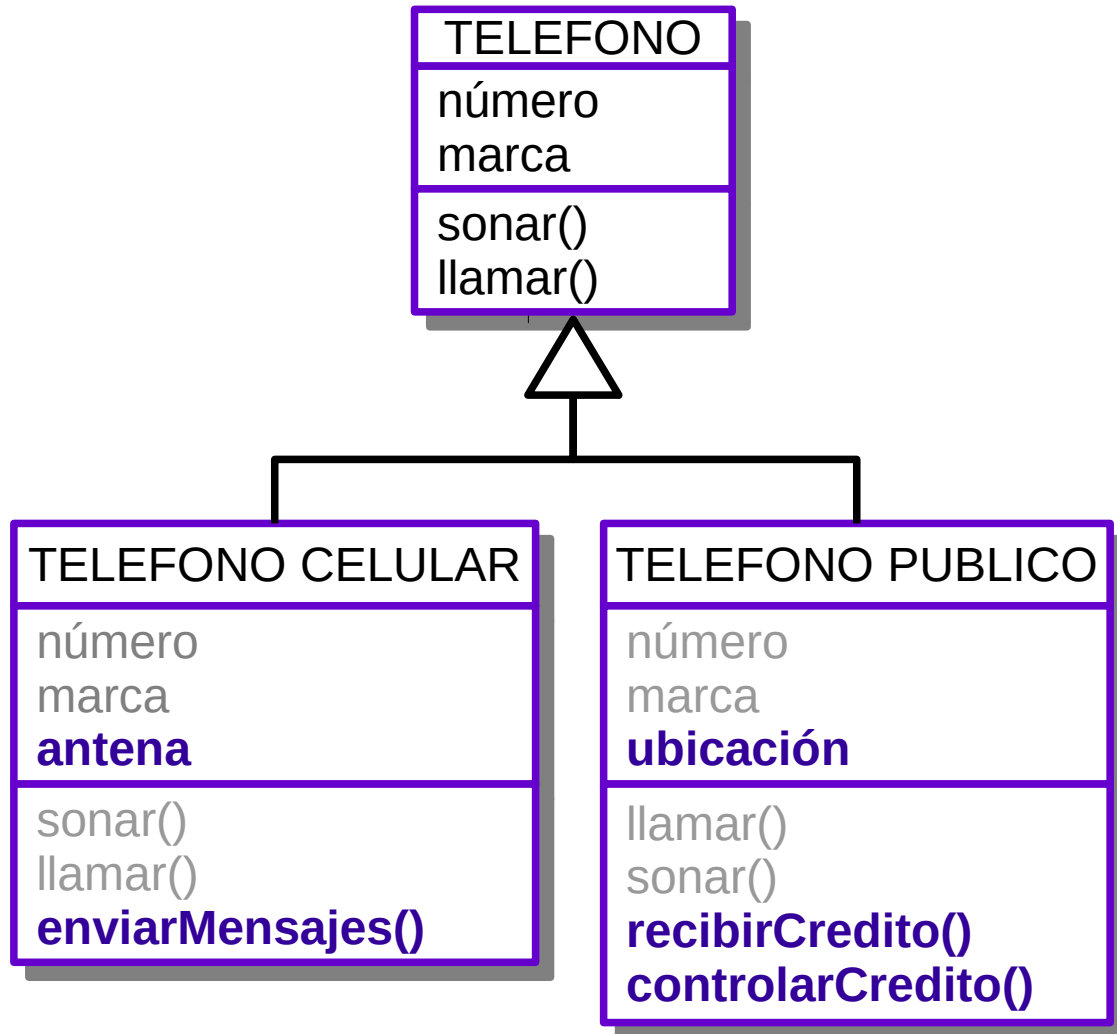
La programación orientada a objetos permite a las clases expresar similitudes entre objetos que tienen algunas características y comportamiento común. Estas **similitudes pueden expresarse usando herencia**. El término **herencia** se refiere al hecho de que una clase **hereda** las características (atributos, variables) y el comportamiento (métodos) de otra clase.



# Concepto de Herencia: Ejemplo Clase Telefono



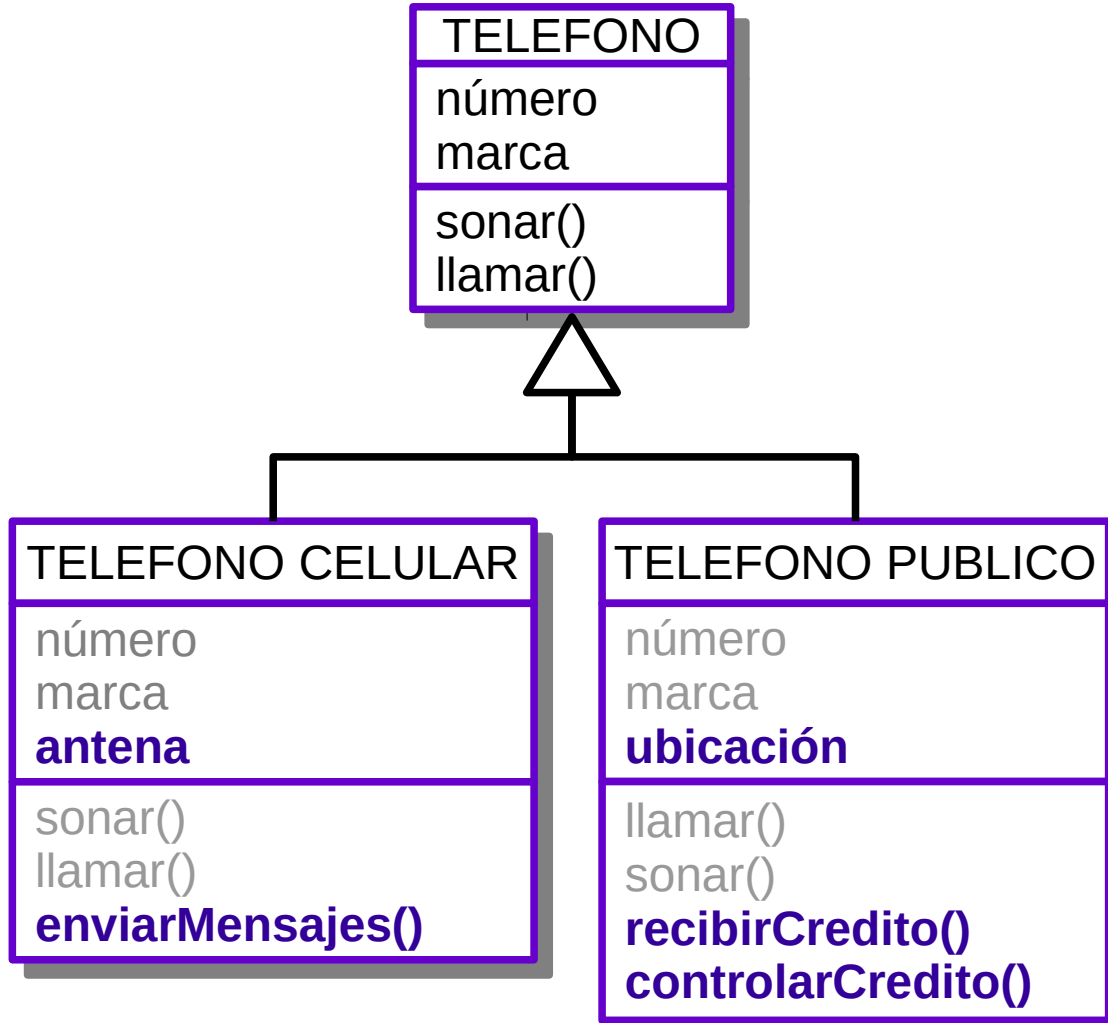
# Concepto de Herencia



Cada uno de estos tipos de teléfonos que hemos enumerados tienen “características comunes” (atributos y métodos).

Podemos observar claramente que tanto Teléfono Celular como Teléfono Público heredan atributos y comportamientos del objeto Teléfono.

# Concepto de Herencia



Los atributos heredados **no** deben volver a escribirse en la clase hija, excepto que se quiera **ocultar(\*)** al atributo de la clase padre

Los métodos heredados no deben volver a escribirse en la clase hija, excepto que se quieran **sobreescibir(\*)**.

(\*) después se analizarán estos conceptos

# Concepto de Herencia



La herencia es, después de la agregación o composición, el mecanismo más utilizado para alcanzar algunos de los objetivos más preciados en el desarrollo de software como lo son la reutilización y la extensibilidad. A través de ella los diseñadores pueden **crear nuevas clases partiendo de una clase o de una jerarquía de clases preexistente** (ya comprobadas y verificadas) evitando con ello el rediseño, la modificación y verificación de la parte ya implementada. La herencia facilita la creación de objetos a partir de otros ya existentes e implica que una subclase obtiene todo el comportamiento (métodos) y eventualmente los atributos (variables) de su superclase.

# Concepto de Herencia - Aspectos

## Aspectos que se observan en la Herencia:

### **Herencia de contrato o tipo:**

la subclase adquiere el tipo de la superclase y se puede utilizar polimórficamente allí donde se pudiera utilizar la superclase.

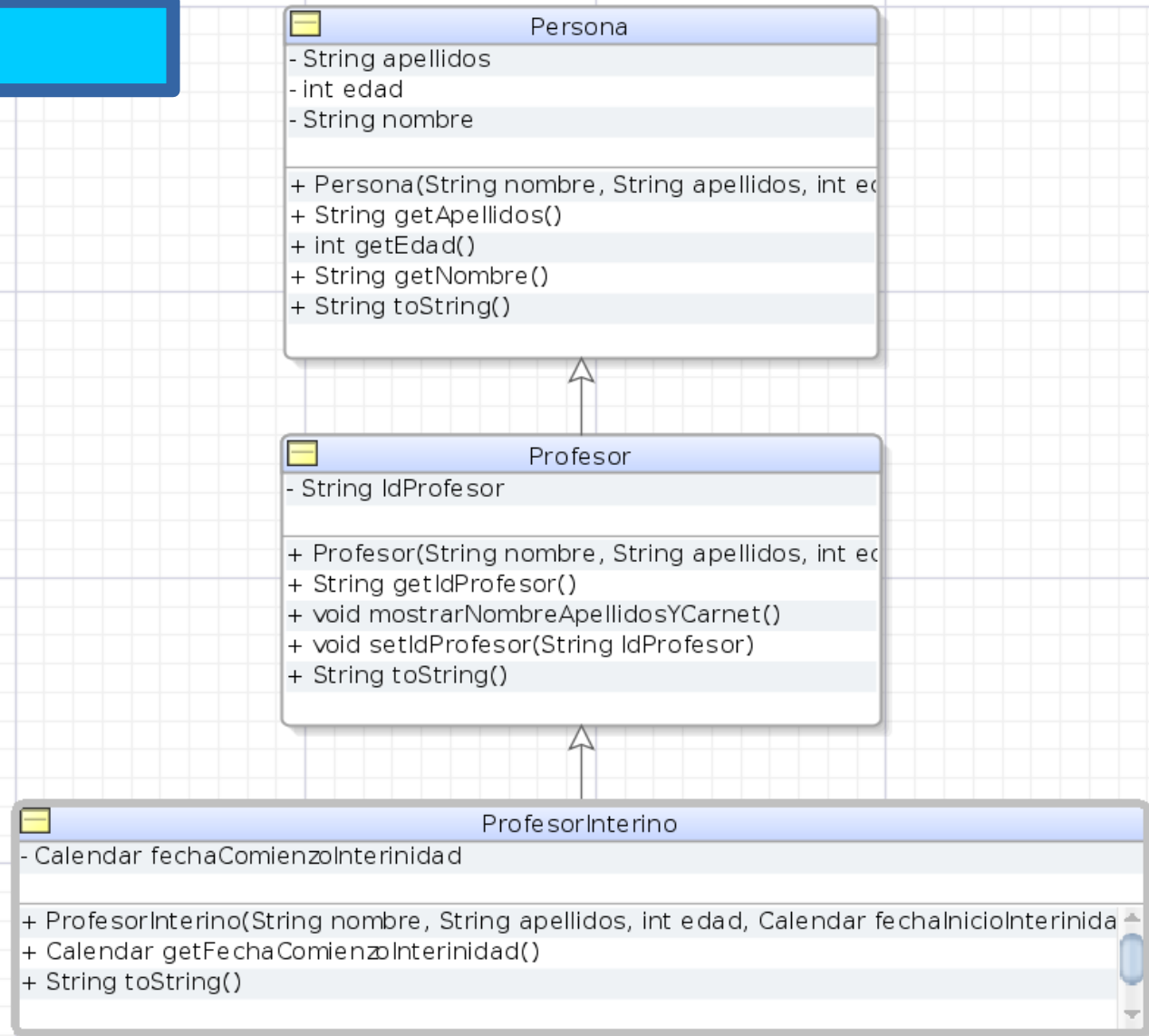
### **Herencia de implementación:**

la subclase adquiere la implementación de la superclase en términos de sus campos y métodos accesibles

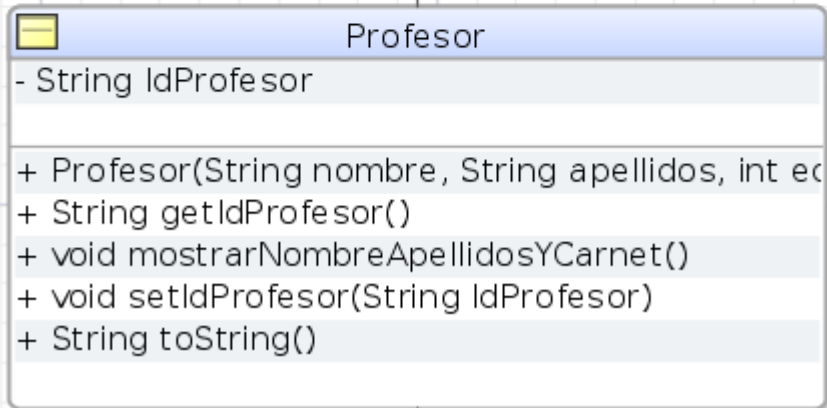
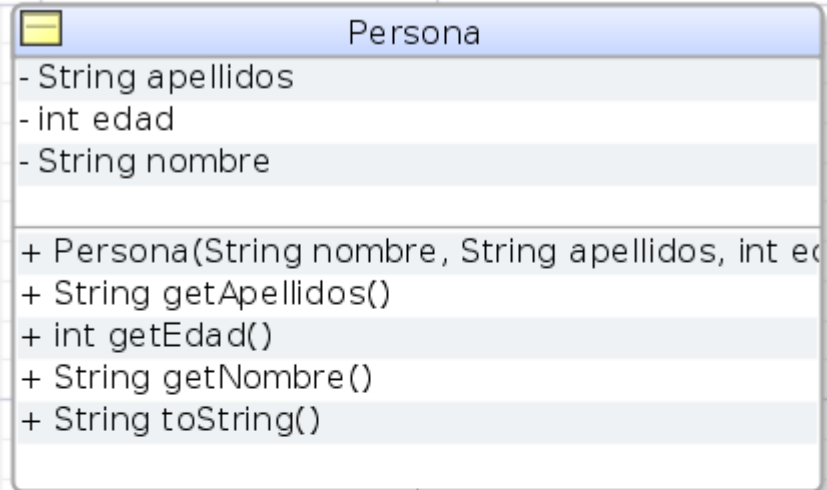


# Ejemplo

Una subclase puede agregar características y comportamiento a su superclase y **reemplazar** o modificar el **comportamiento** heredado.



# Ejemplo



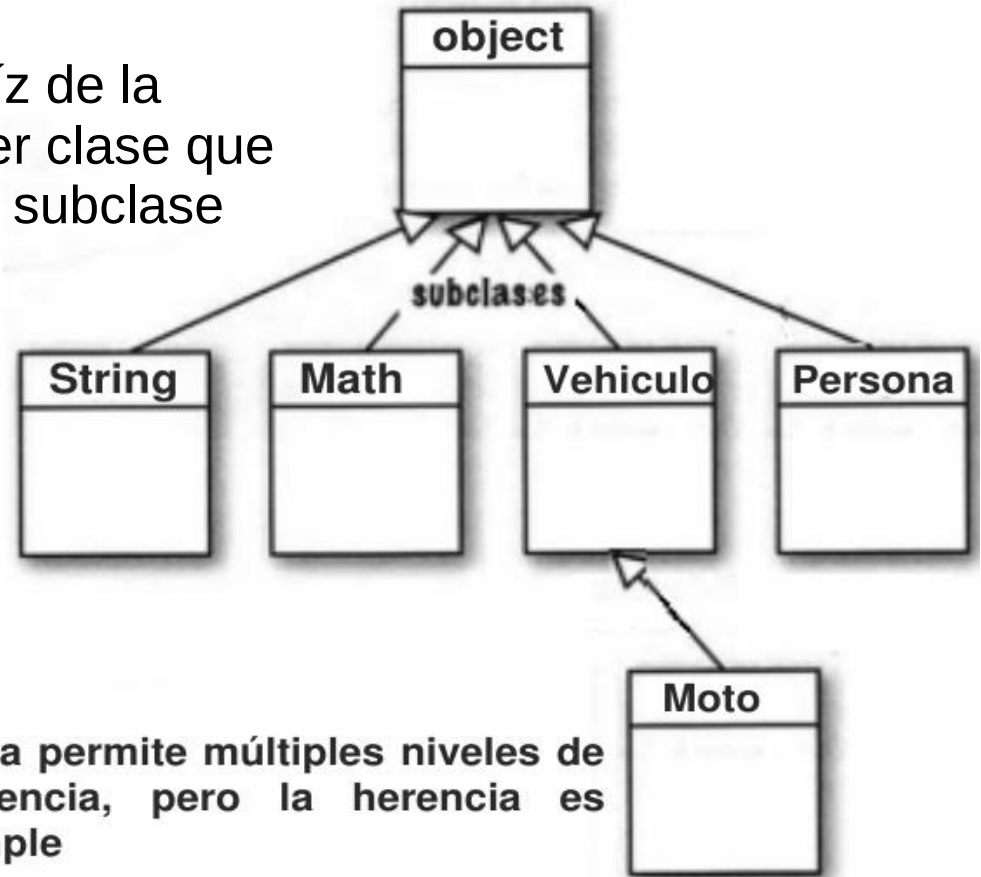
¿Cómo indicamos la relación de herencia en el código fuente Java? Con la palabra clave **extends**

```
//Código de la clase profesor, subclase de Persona
public class Profesor extends Persona
{
    //Campos específicos de la subclase
    private String IdProfesor;
    //Constructor de la subclase: incluye el constructor de la clase base
    public Profesor(String nombre, String apellidos, int edad)
    {
        super(nombre, apellidos, edad);
        IdProfesor = "Unknown";
    } //Cierre del constructor
}
```

# Concepto de Herencia

La clase **Java.lang.Object** es la raíz de la jerarquía de clases en Java. Cualquier clase que no especifique un padre directo, será subclase directa de Object.

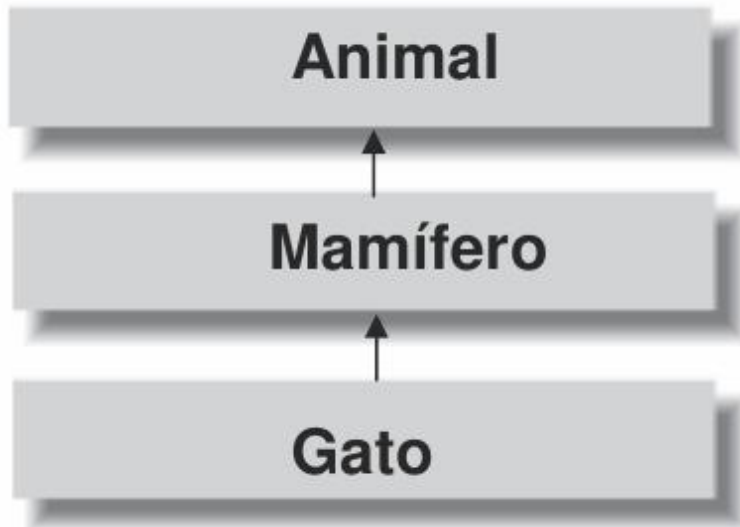
```
public class Persona {  
    private String apellido;  
    private String nombres;  
    ...  
}
```



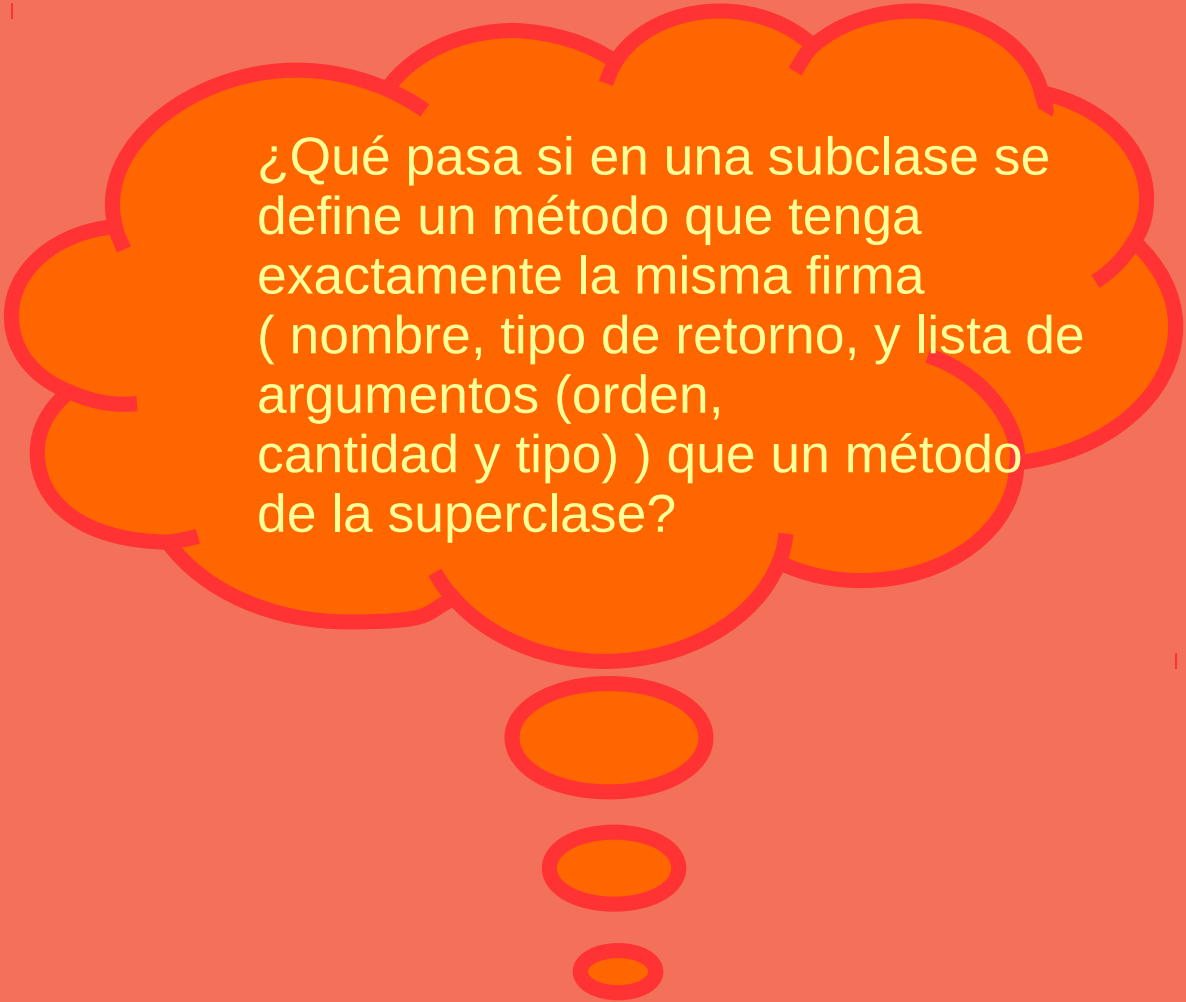
Java permite múltiples niveles de herencia, pero la herencia es simple

# Concepto de Herencia

La herencia nos permite concebir una nueva clase de objetos como un **refinamiento** de otra clase de objetos ya existente, conservando las similitudes entre las clases de objetos y especificando solamente las diferencias de la nueva clase de objetos.

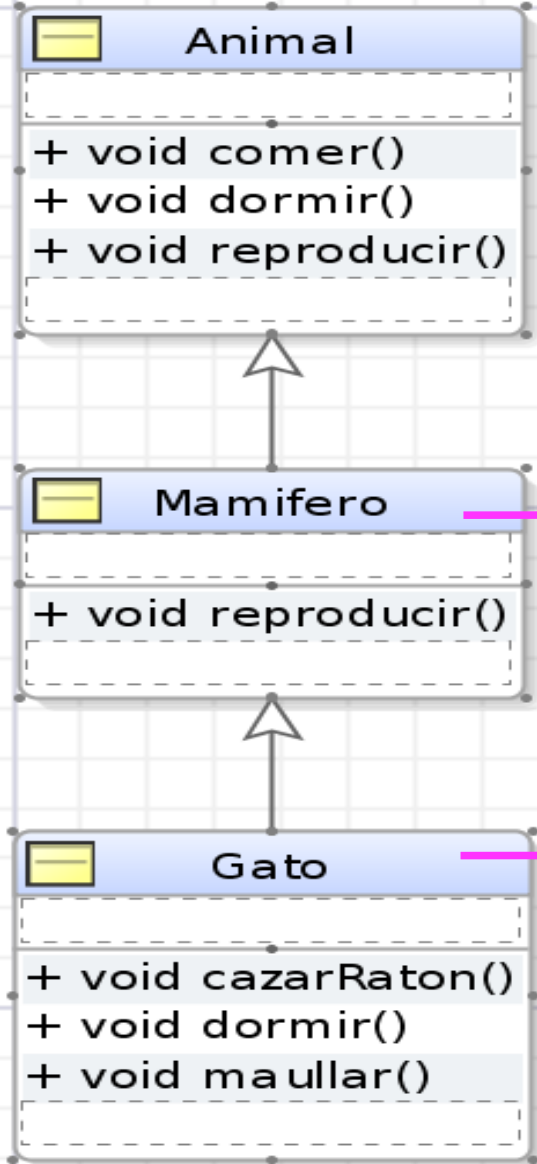


La herencia es el mecanismo de los lenguajes orientados a objetos que implementa la relación “**es un**” entre clases. Permite establecer que *una clase es como otra, con la excepción de que la nueva clase incluye “cosas” extras.*



¿Qué pasa si en una subclase se define un método que tenga exactamente la misma firma ( nombre, tipo de retorno, y lista de argumentos (orden, cantidad y tipo) ) que un método de la superclase?

# Sobreescribiendo Métodos



- Hereda todos los métodos de Animal,
- Puede acceder a ellos porque son públicos
- Sobreescribe reproducir()

- Hereda todos los métodos de Animal y Mamifero
- Puede acceder a ellos porque son públicos
- Sobreescribe dormir()
- Agrega maullar() y cazarRaton()

# Sobreescribiendo Métodos

La sobreescritura permite definir un método en una subclase, que tenga exactamente la misma firma que un método de la superclase.

Objeto:

Gaturro es una instancia de la clase Gato

**gaturro (un Gato)**

comer()

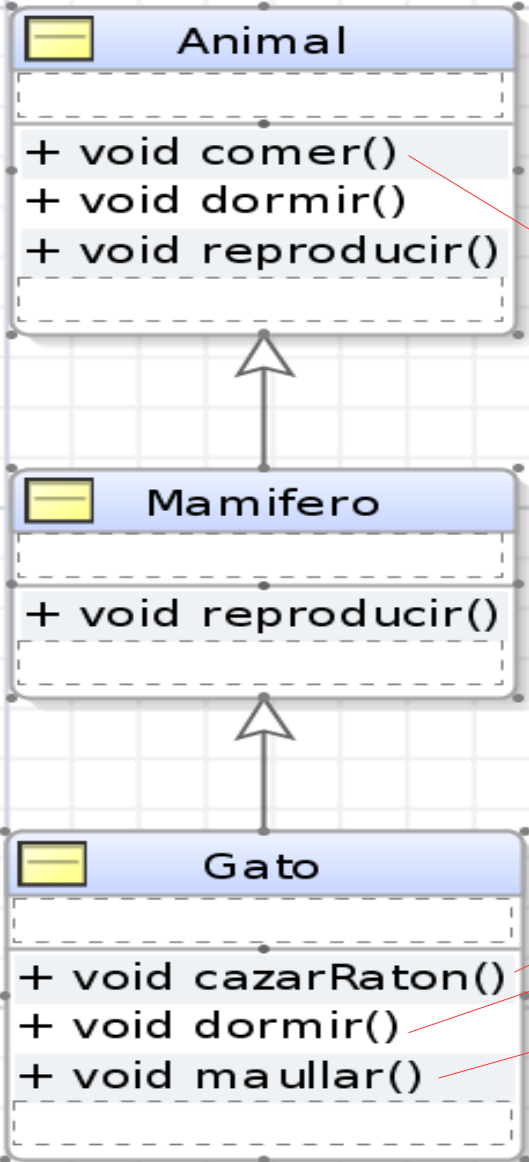
reproducir()

cazarRatón()

dormir()

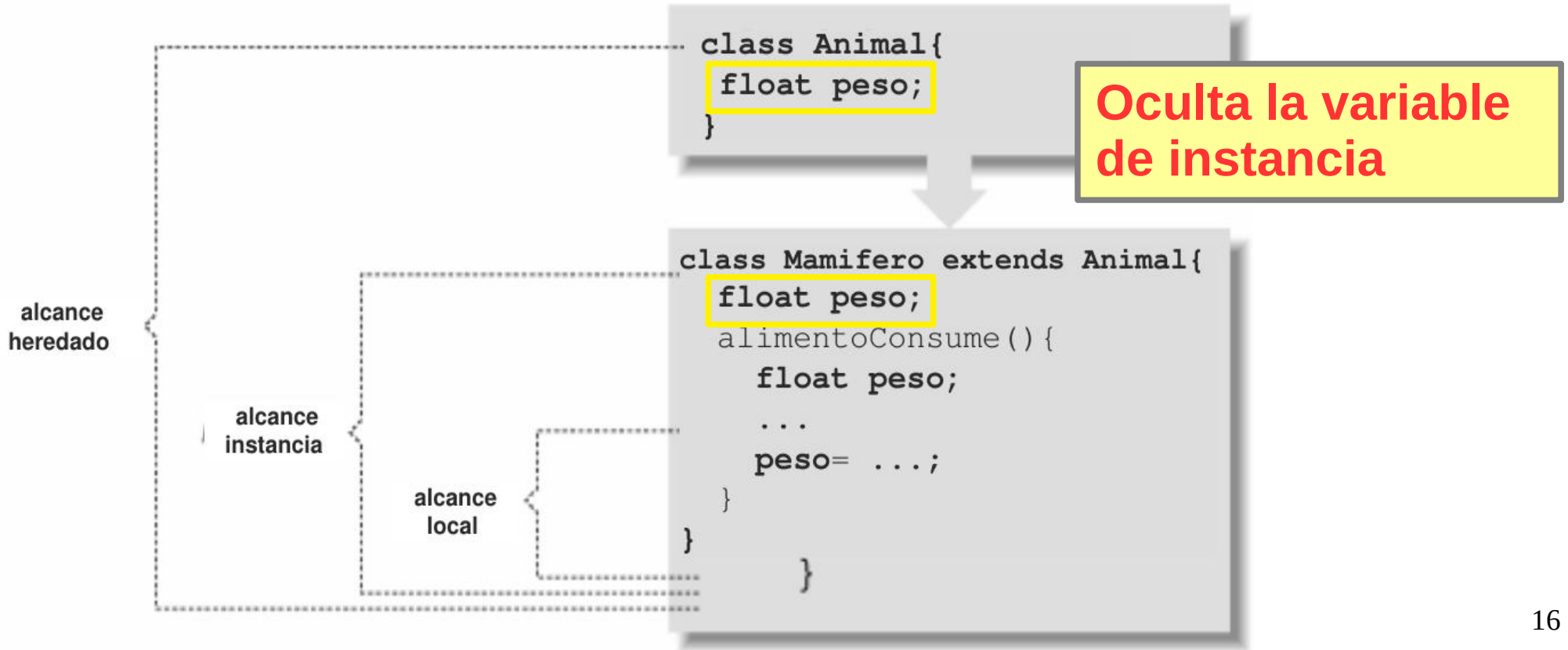
maullar()

El objetivo de sobreescribir un método es cambiar su implementación.



# Ocultamiento de variables

¿Qué pasa si en una subclase se define una variable de instancia del mismo tipo y nombre que la definida en la superclase?





# La palabra reservada super

La palabra `super` puede utilizarse para invocar a un miembro de la superclase oculto (atributo) o sobrescrito (método o constructor) por otro de la propia clase.

Se puede invocar al método `detalle()` de la superclase, usando la palabra clave `super`

```
Test.java  Auto.java  Vehiculo.java
package teoria3;
public class Auto extends Vehiculo {
    int cantPuertas=4;
    String nroPatente="EXQ056";

    public String detalles() {
        return super.detalles() + "\n" +
            "Cantidad de puertas:" + cantPuertas;
    }

    public void nro() {
        System.out.println(super.nroPatente);
    }
}
```

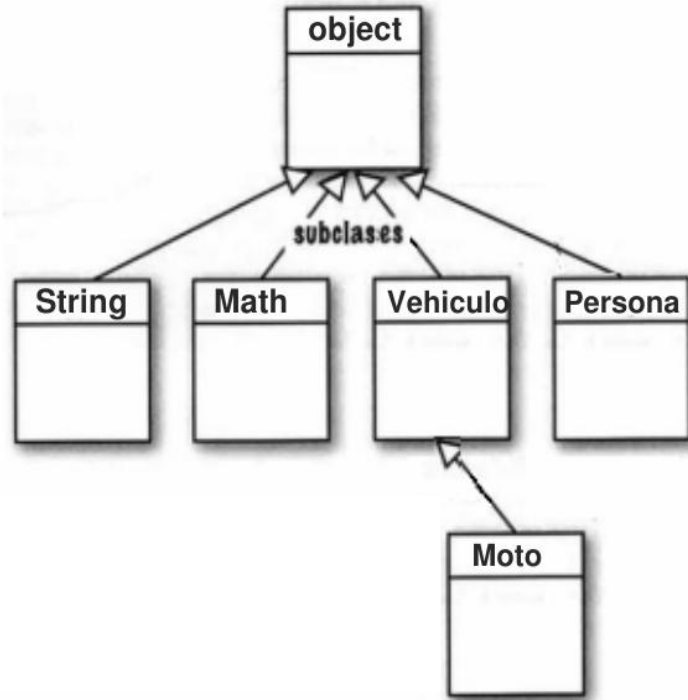
```
Test.java  Auto.java  Vehiculo.java
package teoria3;
public class Vehiculo {
    String nroPatente="AAA000";
    String propietario="AGENCIA";

    public String detalles() {
        return "Patente: " + nroPatente +
            "\n" + "Propietario: " + propietario;
    }
}
```

La palabra clave `super` también puede utilizarse para hacer referencia a un atributo de la superclase que esta oculto por la definición de un atributo con el mismo tipo y nombre

# El Constructor

El constructor de una clase extendida puede tratar con variables de estado heredadas, pero sólo su superclase sabe cómo inicializar correctamente su estado, de forma que cumpla con su contrato.



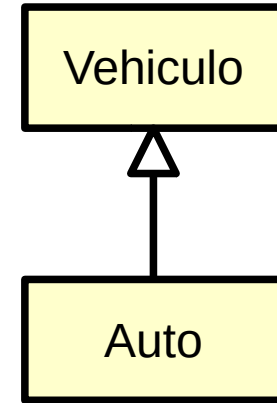
Los constructores de clases extendidas deben delegar la construcción del estado heredado invocando a un constructor de la superclase. Se utiliza **super(..)**



**SUPER !**

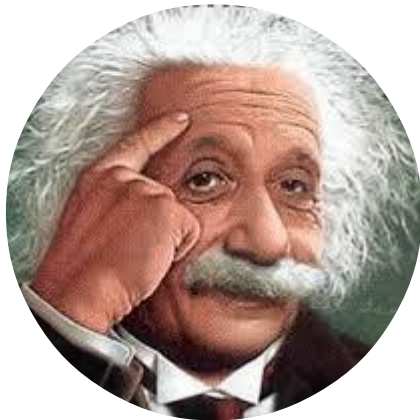
# El Constructor

```
String nroPatente;  
String propietario;  
  
public Vehículo(String nroPatente, String propietario)  
{  
    super();  
    this.nroPatente = nroPatente;  
    this.propietario = propietario;  
}
```



```
private int cantPuertas;  
  
public Auto(String nroPatente, String propietario, int cantPuertas)  
{  
    super(nroPatente, propietario);  
    this.setCantPuertas(cantPuertas);  
}
```

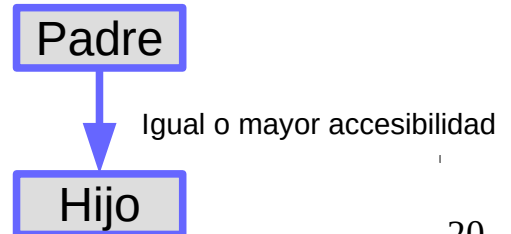
# Especificadores de acceso



Los métodos que redefinen a otros poseen sus propios especificadores de acceso. Una subclase puede **modificar el acceso** de los métodos de la superclase, pero sólo para hacerlos **más accesibles**.

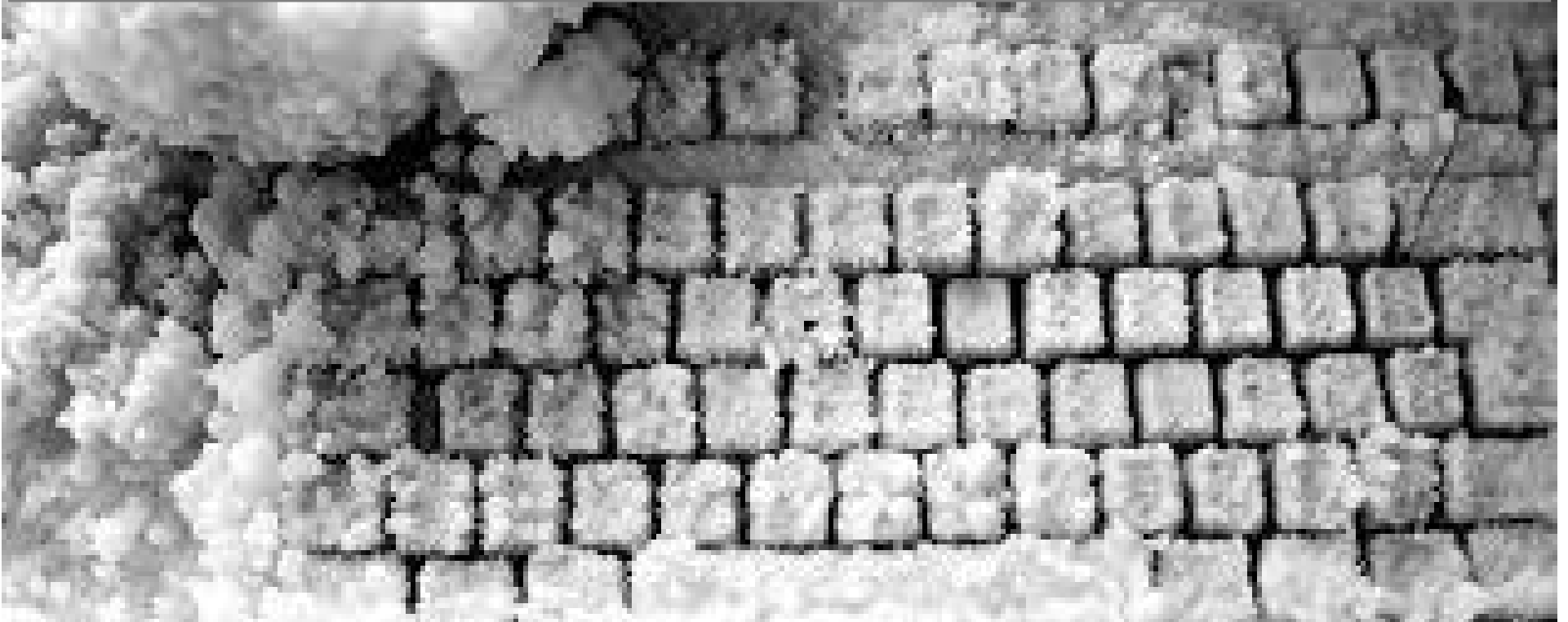
*Al hacer un método menos accesible que en la superclase se violaría el contrato de la superclase, ya que una instancia de la subclase no se podría utilizar en lugar de una instancia de la superclase.*

Un método privado puede redefinirse como protegido o público. No al revés



# Clase FINAL

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa **no pueda ser extendida**. Para esto está la cláusula final, que tiene significados ligeramente distintos según se aplique a un dato miembro, a un método o a una clase.

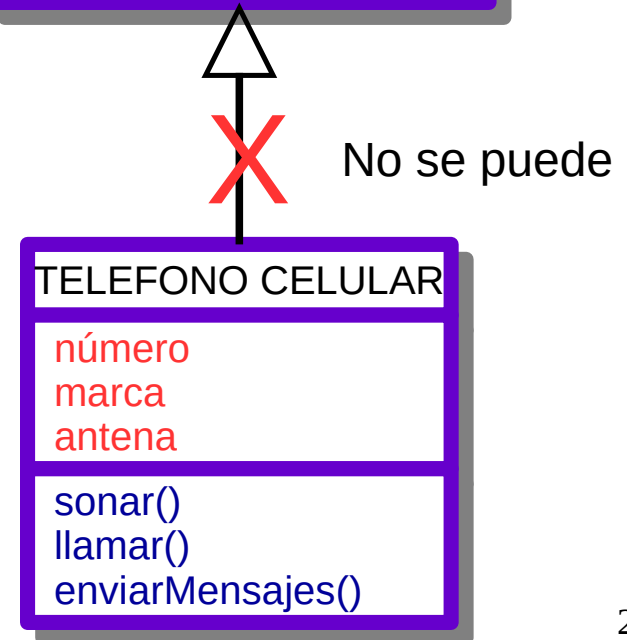


# Clase FINAL

Para una clase, final significa que la clase no puede extenderse.

Es, por tanto el punto final de la cadena de clases derivadas.

```
final class Telefono
{
    int número;
    String marca;
    ...
}
```



# Método FINAL

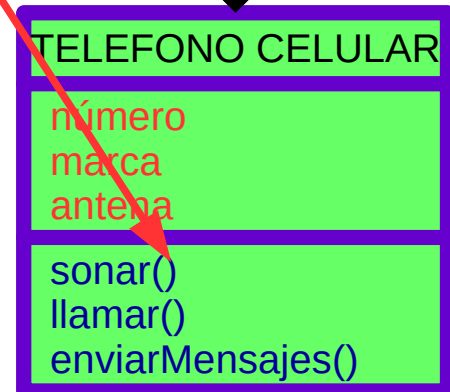
Para un método, final significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Telefono
{
    . . .
    public final void sonar()
    {...}
}
```

```
class TelefonoCelular extends Telefono
{
    . . .
    public final void sonar()
    {...}
}
```



No se puede



# Variable FINAL

Para un dato miembro, final significa también que no puede ser redefinido en una clase derivada, y que su valor no puede ser cambiado en ningún sitio; es decir el modificador final sirve también para definir valores constantes. Por ejemplo:

```
class Circulo
{
    public final static float PI = 3.141592;
}
```

Al declarar una variable como final, se previene que su contenido sea modificado. No obstante, Java permite separar la definición de la inicialización. Esta última se puede hacer más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos.

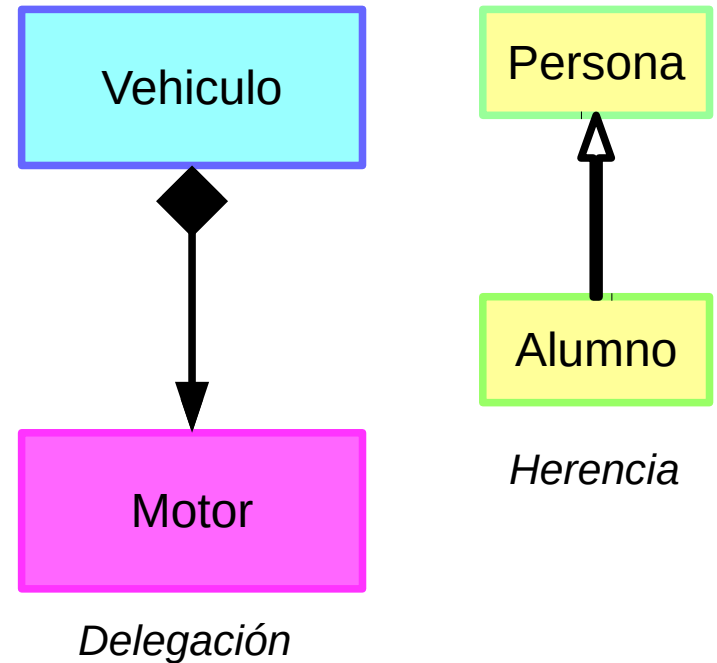
La variable final así definida, es constante pero no tiene porqué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada. Esto es lo más parecido a las constantes de otros lenguajes de programación.



# Delegación

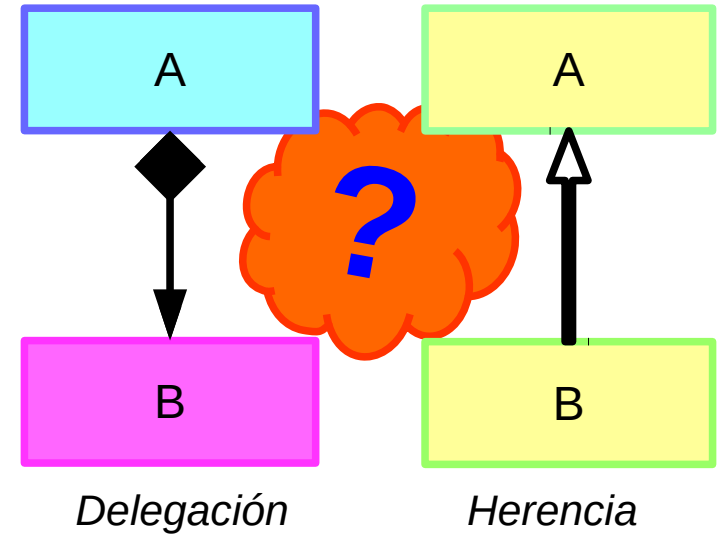
Se llama delegación a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que delegará parte de sus funcionalidades. Esta relación entre clases suele ser la más indicada cuando es necesaria una asociación entre las clases pero el principio de Liskov no se cumple. También puede verse como la relación entre clases "S contiene a T". Por ejemplo, Vehículo contiene un Motor, pero Alumno no contiene a Persona, sino que es una Persona.

**Ver Clase 2 – La relación de asociación:  
agregación y composición**



# Delegación

- La delegación se caracteriza por **"reutilización selectiva"**, en cambio en herencia es un "todo o nada". Es cierto que en Composición y delegación se escribe mucho más que en herencia, ya que en herencia se hereda de forma declarativa, y esto simplifica al programador en ciertos casos. Se habla de la herencia como Caja Blanca y de Composición y delegación como Caja Negra. Es conveniente usar herencia cuando la relación de "Es Un" es clara, es obvia.



- Es una buena costumbre usar la delegación en lugar de la herencia si no es necesario hacer conversiones hacia arriba.** En general, es mejor cualquier tipo de composición, si es factible, en lugar de la herencia. Sin embargo solo es una buena elección si simplifica más de lo que complica.

# Pregunta

En algunas circunstancias es posible definir clases que representan un concepto abstracto y como tal **no** deberían ser instanciadas.

Pensemos en un ejemplo del mundo real: “la comida”.



Definimos una clase llamada Comida

¿Es posible crear diferentes instancias de la clase Comida?

# Supongamos que la respuesta es **SI**.



No es posible  
determinarlo!!!



# Supongamos que la respuesta es **NO**.



*Esta es la respuesta correctaaaaa !!!*

La comida como concepto es abstracto, no tiene forma, no tiene color, no tiene sabor, no tiene temperatura solo representa un conjunto de sustancias alimenticias que se consumen en diferentes momentos del día pero de la que no podemos determinar otras características.

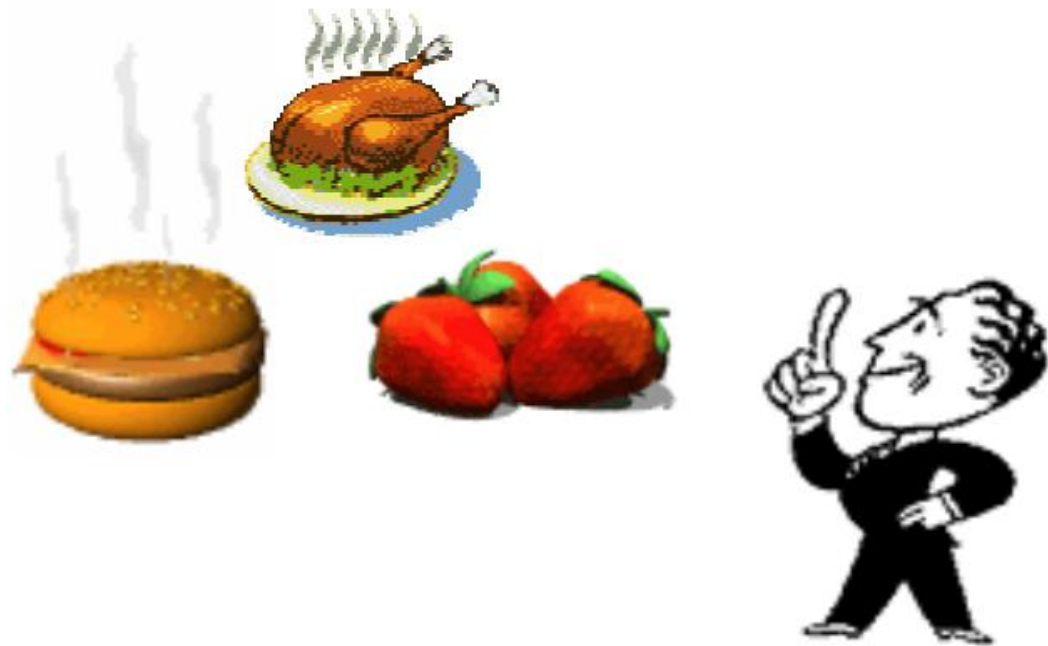


**la comida, es  
una clase  
abstracta !**



# Clases Concretas

Sin embargo, si pensamos en determinadas comidas: **las frutillas, el pollo, las hamburguesas**, etc.. **Si** pensamos en conceptos concretos pues se trata de tipos de comidas que tienen características propias como sabor, color, temperatura, forma, etc.

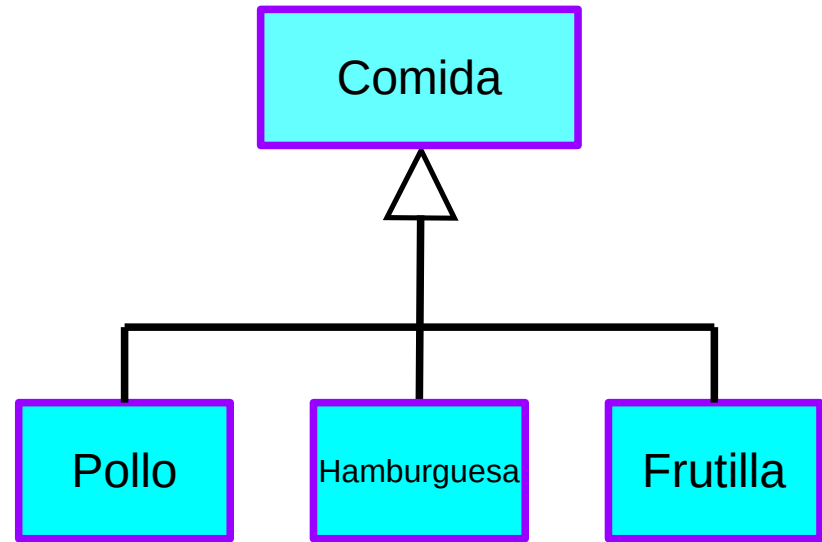


En este caso, podemos pensar en objetos concretos con características propias: los pollos son objetos concretos, cada uno tiene su propio peso, sabor y temperatura de acuerdo a la preparación; las hamburguesas también son objetos concretos, cada una tiene su propio sabor de acuerdo a si es casera o no, al tipo de carne con la que se prepare, pueden ser de diferente tamaño y forma, etc.

**Son conceptos concretos.**

# Clases Abstractas

El objetivo de una clase abstracta es *ser extendida* por clases concretas y definir una interface de comportamiento común de los objetos de sus subclases.



En JAVA para definir clases abstractas se usa la palabra clave **abstract**

# Clases Abstractas – Ejemplo Cuenta Bancaria

**En un banco existen tres tipos de cuenta:**

Caja de Ahorro  
Cuenta Corriente  
Cuentas Universitarias

**Si revisamos su comportamiento nos encontraremos con las siguientes características en común:**

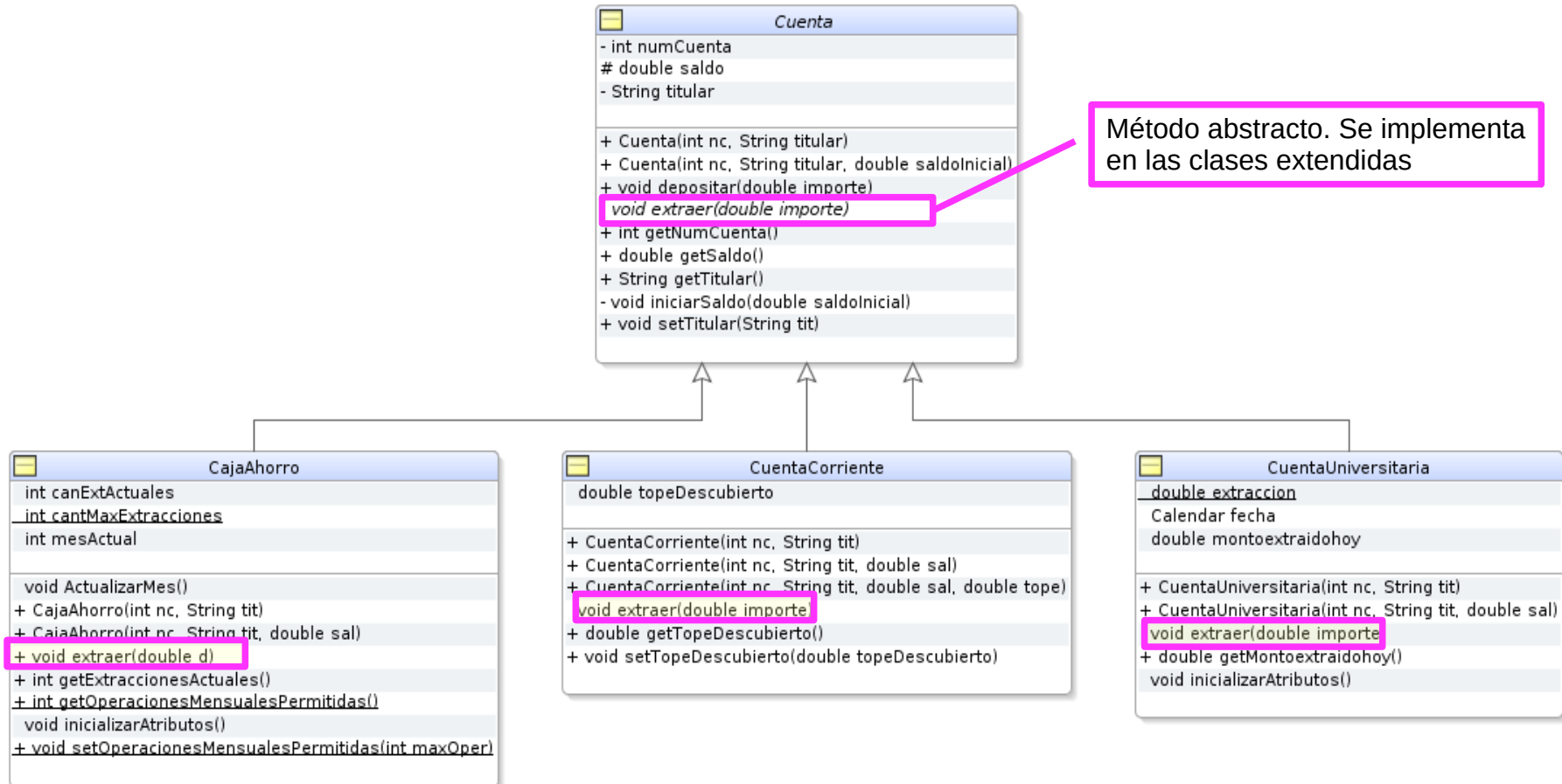
- Todas llevan cuenta de su saldo.
- Todas permiten realizar depósitos.
- Todas permiten realizar extracciones.

**Pero cada una tiene un tipo de restricción distinto en cuanto a las extracciones:**

- Cuentas corrientes: permiten que el cliente gire en descubierto (con un tope pactado con cada cliente).
- Cajas de ahorro: poseen una cantidad máxima de extracciones mensuales (para todos los clientes). No se permite girar en descubierto.
- Cuenta universitarias: permite extracciones de no más de 100\$ diarios.



# Clases Abstractas – Ejemplo Cuenta Bancaria



# Clases Abstractas – Ejemplo Cuenta Bancaria

```
public abstract class Cuenta
{
    protected double saldo;
    private String titular;
    private int numCuenta;

    public Cuenta(int nc, String titular)
    {
        numCuenta = nc;
        setTitular(titular);
    }

    public Cuenta(int nc, String titular, double saldoInicial)
    {
        numCuenta = nc;
        setTitular(titular);
        iniciarSaldo(saldoInicial);
    }

    abstract void extraer(double importe);

    public int getNumCuenta()
    { ... }

    private void iniciarSaldo(double saldoInicial)
    { ... }

    public double getSaldo()
    { ... }

    public String getTitular()
    { ... }

    public void setTitular(String tit)
    { ... }

    public void depositar(double importe)
    { ... }
}
```

```
public class CajaAhorro extends Cuenta
{
    static int cantMaxExtracciones = 30;
    int mesActual;
    int canExtActuales;

    void inicializarAtributos()
    { ... }

    CajaAhorro(int nc, String tit)
    { ... }

    public CajaAhorro(int nc, String tit, double sal)
    { ... }

    public static int getOperacionesMensualesPermitidas()
    { ... }

    public static void setOperacionesMensualesPermitidas(int maxOper)
    { ... }

    @Override
    public void extraer(double d)
    {
        if (d <= this.getSaldo())
        {
            if (canExtActuales <= cantMaxExtracciones)
            {
                saldo = saldo - d;
                //ojo aca saldo esta protected..
            }
            else
            {
                //@TODO
            }
        }
        else
        {
            //@TODO
        }
    }

    public int getExtra...
    { ... }

    void ActualizarMes(
    { ... }
}
```

# Clases Abstractas – Ejemplo Cuenta Bancaria

```
public abstract class Cuenta
{
    protected double saldo;
    private String titular;
    private int numCuenta;

    public Cuenta(int nc, String titular)
    {
        numCuenta = nc;
        setTitular(titular);
    }

    public Cuenta(int nc, String titular, double saldoInicial)
    {
        numCuenta = nc;
        setTitular(titular);
        iniciarSaldo(saldoInicial);
    }

    abstract void extraer(double importe);

    public int getNumCuenta()
    { ... }

    private void iniciarSaldo(double saldoInicial)
    { ... }

    public double getSaldo()
    { ... }

    public String getTitular()
    { ... }

    public void setTitular(String tit)
    { ... }

    public void depositar(double importe)
    { ... }
}
```

```
public class CuentaCorriente extends Cuenta
{
    double topeDescubierto = 0;

    public CuentaCorriente(int nc, String tit, double sal, double tope)
    {
        super(nc, tit, sal);
        setTopeDescubierto(tope);
    }

    public CuentaCorriente(int nc, String tit, double sal)
    {
        super(nc, tit, sal);
    }

    public CuentaCorriente(int nc, String tit)
    {
        super(nc, tit);
    }

    public double getTopeDescubierto()
    {
        return topeDescubierto;
    }

    public void setTopeDescubierto(double topeDescubierto)
    {
        this.topeDescubierto = topeDescubierto;
    }

    @Override
    void extraer(double importe)
    {
        double rta = getTopeDescubierto() - Math.abs(getSaldo() - importe);
        if (rta >= 0.01d)
        {
            saldo = saldo - importe;
        }
        else
        {
            // @TODO
        }
    }
}
```

# Clases Abstractas – Ejemplo Cuenta Bancaria

```
public abstract class Cuenta
{
    protected double saldo;
    private String titular;
    private int numCuenta;

    public Cuenta(int nc, String titular)
    {
        numCuenta = nc;
        setTitular(titular);
    }

    public Cuenta(int nc, String titular, double saldoInicial)
    {
        numCuenta = nc;
        setTitular(titular);
        iniciarSaldo(saldoInicial);
    }

    abstract void extraer(double importe);

    public int getNumCuenta()
    {...}

    private void iniciarSaldo(double saldoInicial)
    {...}

    public double getSaldo()
    {...}

    public String getTitular()
    {...}

    public void setTitular(String tit)
    {...}

    public void depositar(double importe)
    {...}
}
```

```
public class CuentaUniversitaria extends Cuenta
{
    final static double extraccion = 100;
    Calendar fecha = Calendar.getInstance();
    double montoextraidohoy = 0;

    public CuentaUniversitaria(int nc, String tit, double sal)
    {...}

    public CuentaUniversitaria(int nc, String tit)
    {...}

    public double getMontoextraidohoy()
    {...}

    void inicializarAtributos()
    {...}

    @Override
    void extraer(double importe)
    {
        if (this.getSaldo() >= importe)
        {
            inicializarAtributos();
            if (importe + montoextraidohoy < extraccion)
            {
                montoextraidohoy = montoextraidohoy + importe;
                saldo = saldo - importe;
            }
            else
            {
                //@TODO
            }
        }
        else
        {
            //@TODO
        }
    }
}
```

# Clases Abstractas – Ejemplo Cuenta Bancaria

## CajaAhorro

```
@Override
public void extraer(double d)
{
    if (d <= this.getSaldo())
    {
        if (canExtActuales <= cantMaxExtracciones)
        {
            saldo = saldo - d;
            //ojo aca saldo esta protected..
        }
        else
        {
            //@TODO
        }
    }
    else
    {
        //@T
    }
}
```

## CuentaCorriente

```
@Override
void extraer(double importe)
{
    double rta = getTopeDescubierto() - Math.abs(getSaldo() - importe);
    if (rta >= 0.01d)
    {
        saldo = saldo - importe;
    }
    else
    {
        //@TODO
    }
}
```

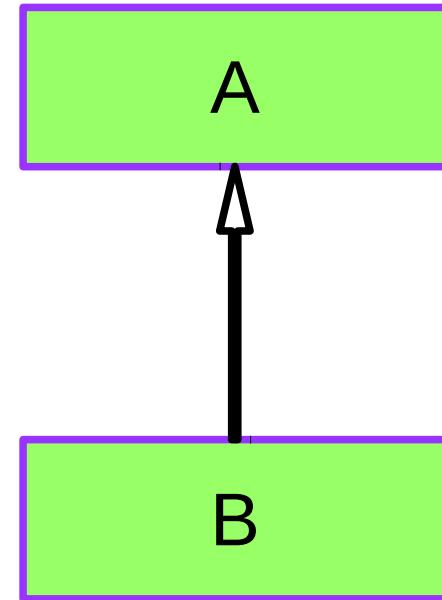
## CuentaUniversitaria

```
@Override
void extraer(double importe)
{
    if (this.getSaldo() >= importe)
    {
        inicializarAtributos();
        if (importe + montoextraido hoy < extraccion)
        {
            montoextraido hoy = montoextraido hoy + importe;
            saldo = saldo - importe;
        }
        else
        {
            //@TODO
        }
    }
    else
    {
        //@TODO
    }
}
```

# Diseñando una clase para que sea extensible

## Especialización vs. Generalización

- Una jerarquía de clases puede surgir de dos maneras:
  - Generalización
  - Especialización



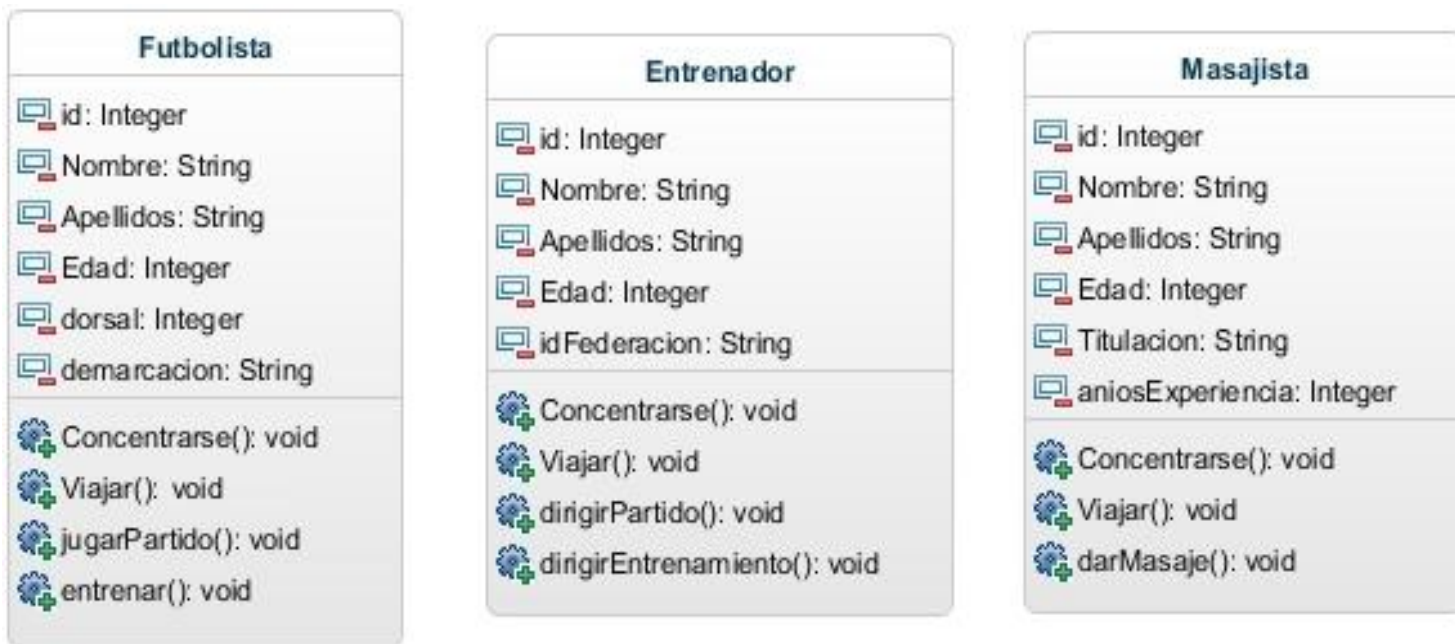
# Diseñando una clase para que sea extensible

## Generalización

- Diseñar clases por separado.
- Descubrir características comunes y específicas.
- Abstraer características comunes en una nueva clase.
- Generar la jerarquía de clases usando la nueva clase como superclase de las clases previamente generadas.
- Eliminar las características comunes de las subclases.
- Verificar que la jerarquía respete el principio “es un”.

# Diseñando una clase para que sea extensible

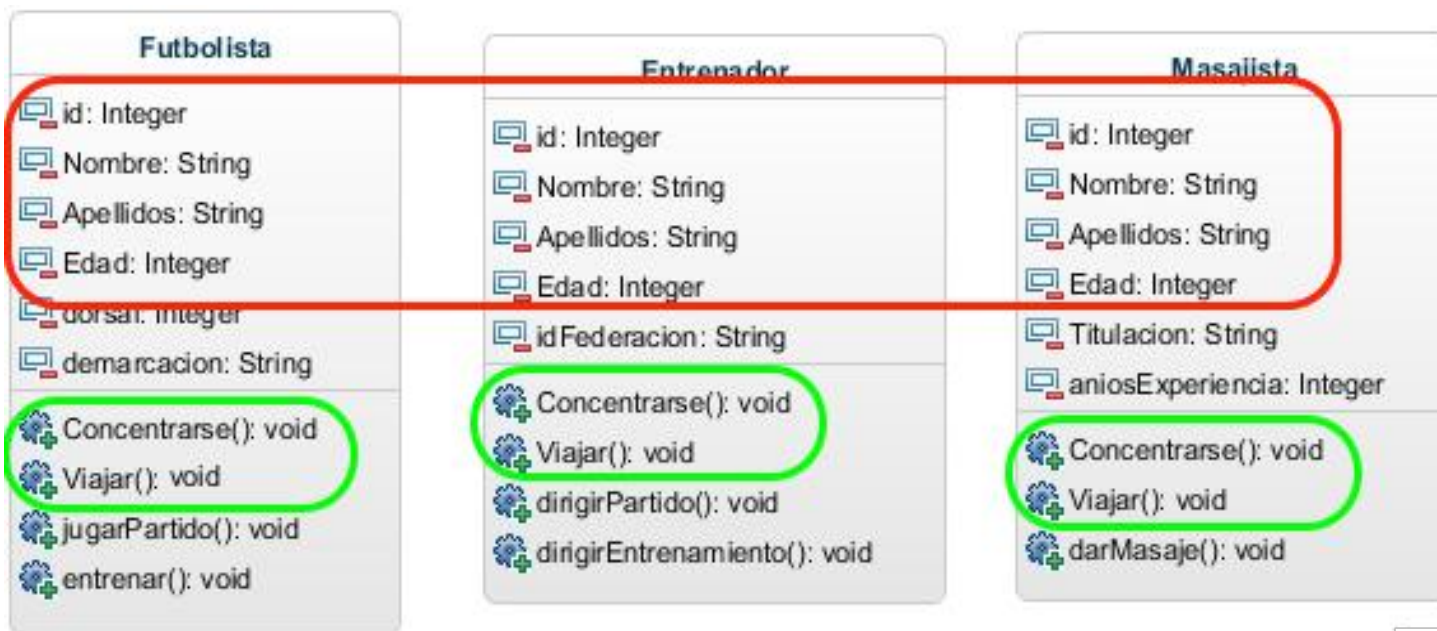
## Generalización





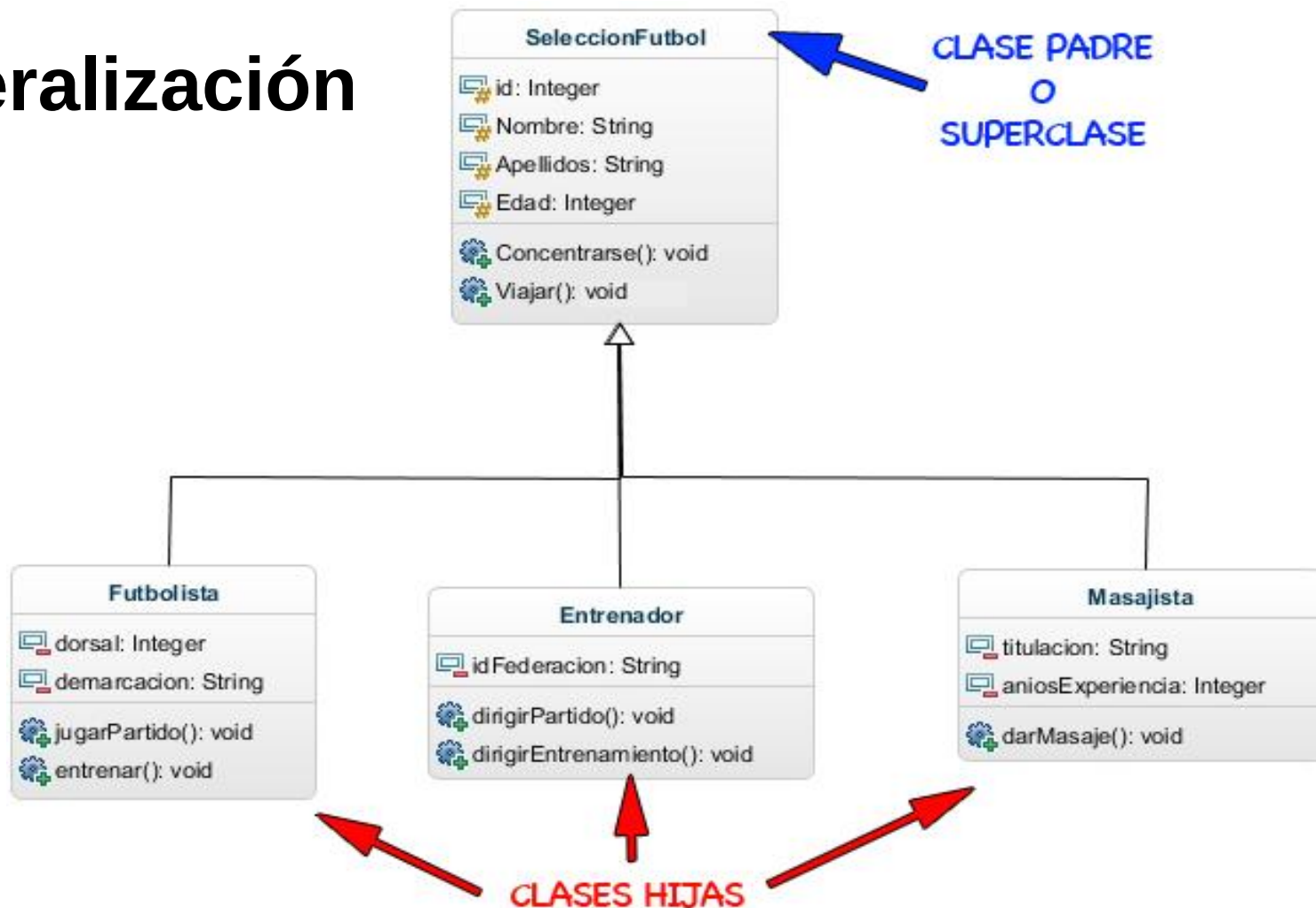
# Diseñando una clase para que sea extensible

## Generalización



# Diseñando una clase para que sea extensible

## Generalización



# Diseñando una clase para que sea extensible

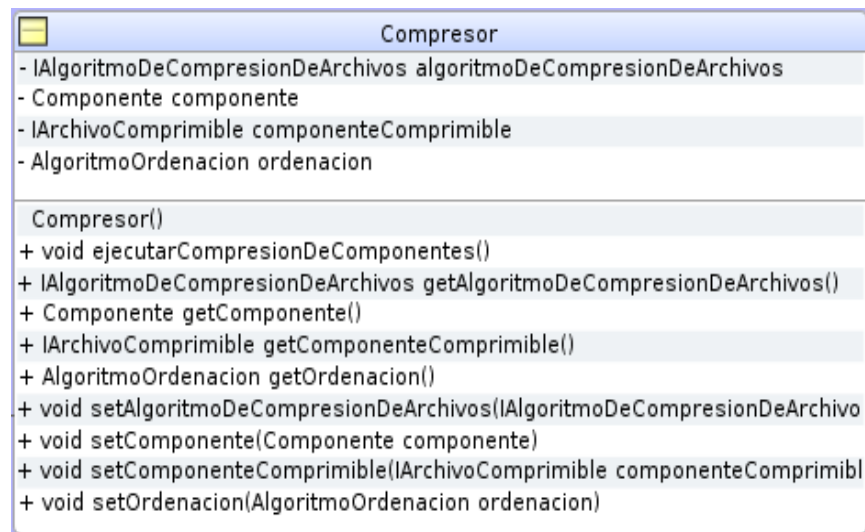
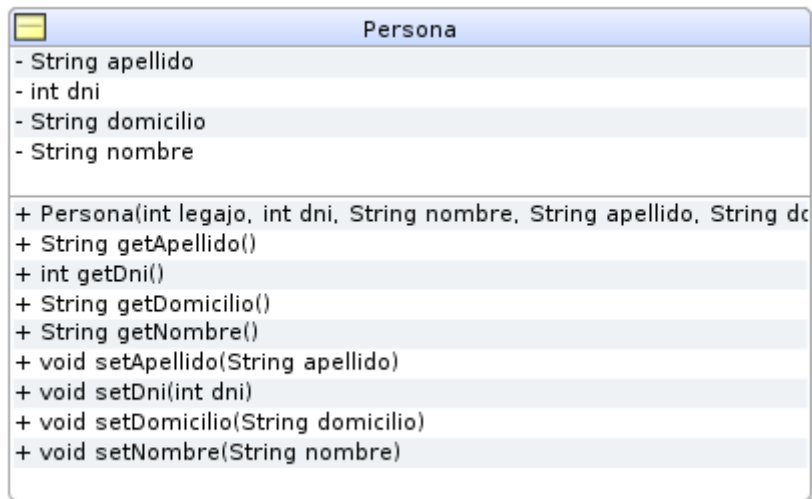
## Especialización

- Antes de diseñar una clase, analizar primero las clases existentes en el sistema.
- Buscar una clase que tenga características comunes con la que estoy por diseñar.
- Si la relación entre ambas clases respeta el principio “es un”, generar una nueva clase a partir de la clase existente.

# Diseñando una clase para que sea extensible

## Especialización

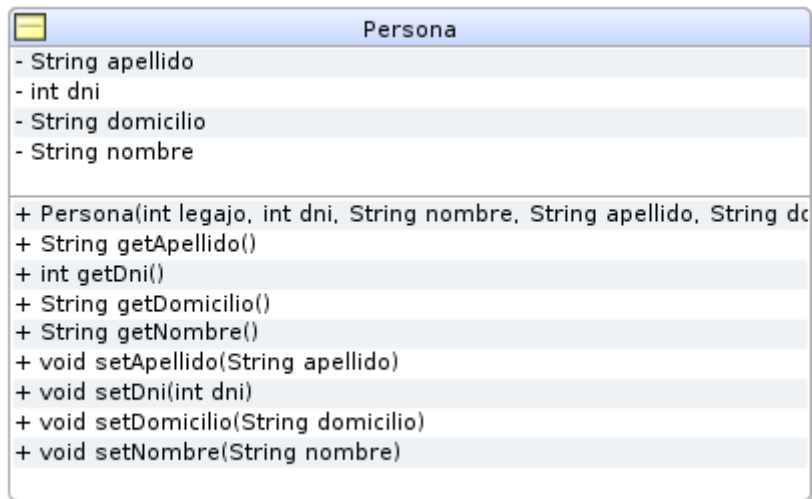
- Antes de diseñar una clase, analizar primero las clases existentes en el sistema.



# Diseñando una clase para que sea extensible

## Especialización

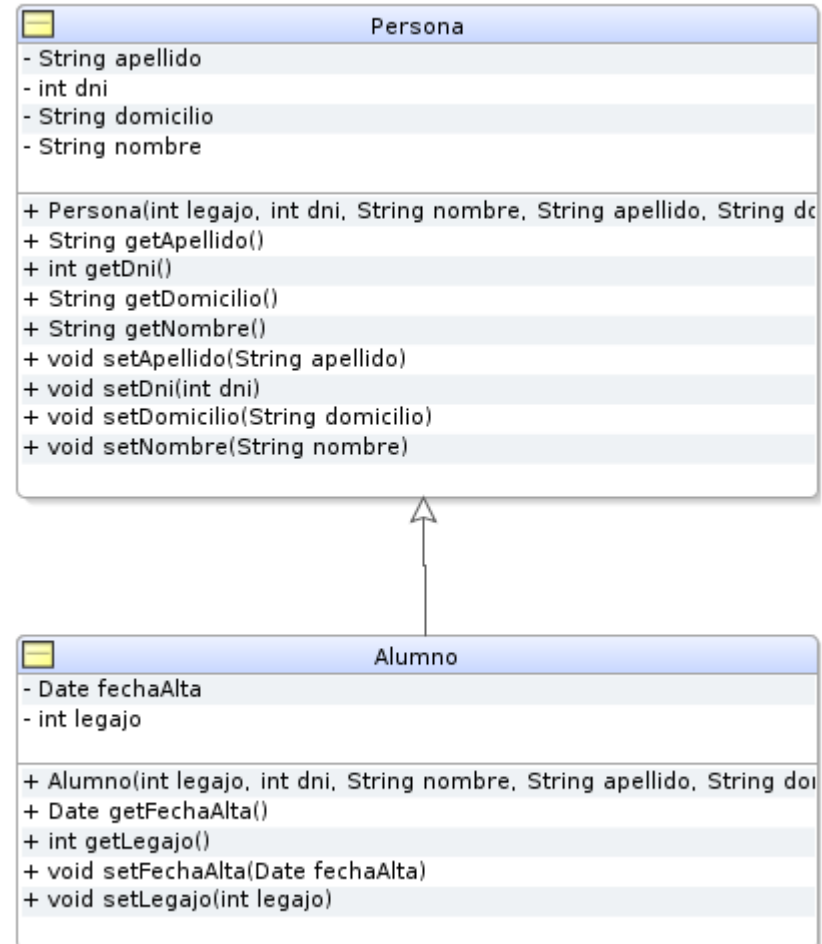
- Buscar una clase que tenga características comunes con la que estoy por diseñar.



# Diseñando una clase para que sea extensible

## Especialización

- Si la relación entre ambas clases respeta el principio “es un”, generar una nueva clase a partir de la clase existente.



# SOLID



Solid es un acrónimo inventado por **Robert C. Martin** para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

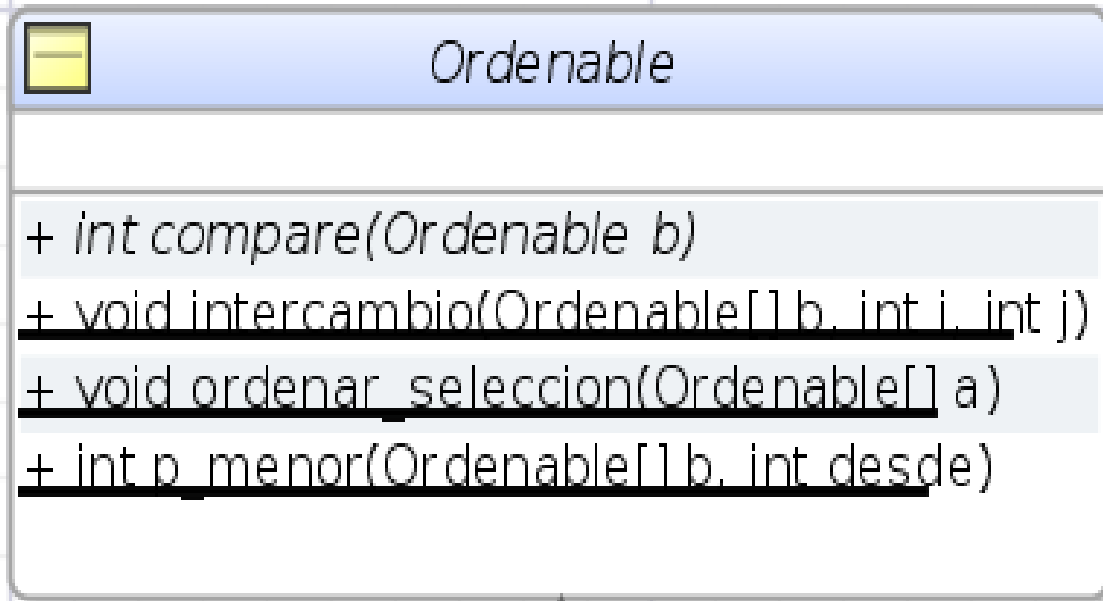
El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo.

# SOLID

Inicial	Significado (acrónimo)	Concepto
S	SRP	<b>Principio de Única Responsabilidad (Single responsibility principle)</b> la noción de que un <b>objeto</b> solo debería tener una única responsabilidad.
O	OCP	<b>Principio Abierto/Cerrado</b> la noción de que las “entidades de software ... deben estar abiertas para su extensión, pero cerradas para su modificación”.
L	LSP	<b>Principio de sustitución de Liskov</b> la noción de que los “objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa”.
I	ISP	<b>Principio de Segregación de la Interfaz (Interface segregation principle)</b> la noción de que “muchas interfaces cliente específicas son mejores que una interfaz de propósito general.”
D	DIP	<b>Principio de Inversión de Dependencia (Dependency inversion principle)</b> la noción de que uno debería “Depender de Abstracciones. No depender de implementaciones.” La <b>Inyección de Dependencias</b> es uno de los métodos que siguen este principio.



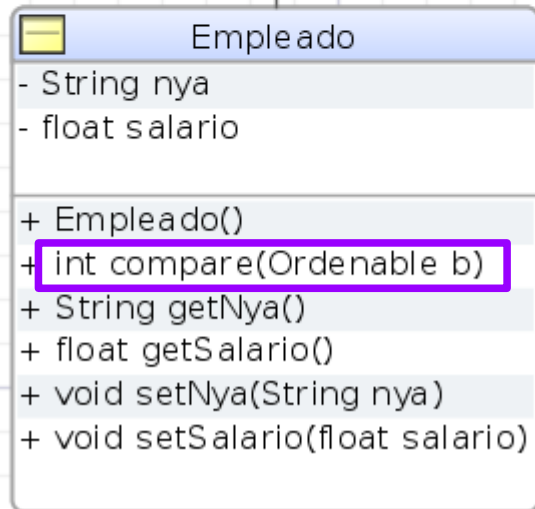
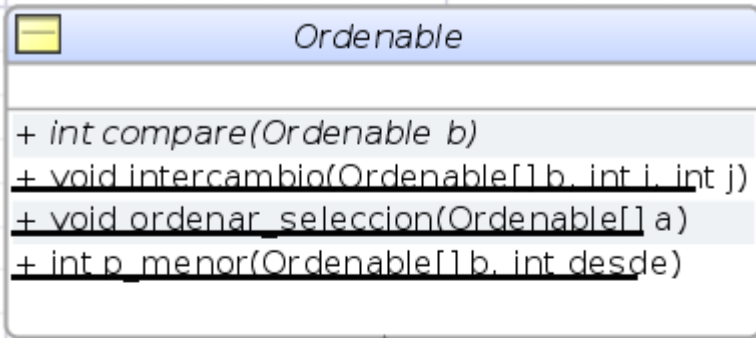
# Ejemplo de clase abstracta



La clase abstracta *Ordenable* representa el algoritmo de ordenación por selección.

El método abstracto ***int compare(Ordenable b)*** determina si el objeto `this` es `<`, `=` o `>` que el parámetro `b`. Retorna cero cuando son iguales, un valor negativo cuando `this < b` y un valor positivo cuando `this > b`.

# Ejemplo de clase abstracta



El método ***compare(..)*** deberá ser implementado en las clases concretas que implementen la clase abstracta **Ordenable**.

Hasta el momento de crear la clase concreta, no se puede establecer el criterio de comparación entre `this` y `b`.

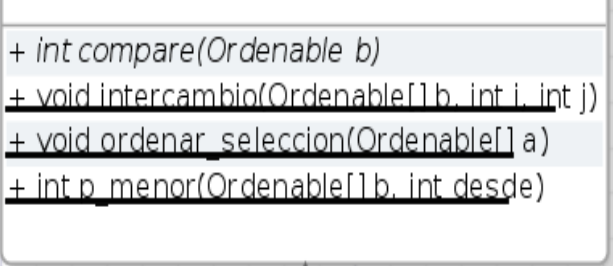
# Ejemplo de clase abstracta

```
public abstract class Ordenable
{
    public abstract int compare(Ordenable b);

    public static int p_menor(Ordenable[] b, int desde)
    {...}

    public static void intercambio(Ordenable[] b, int i, int j)
    {...}

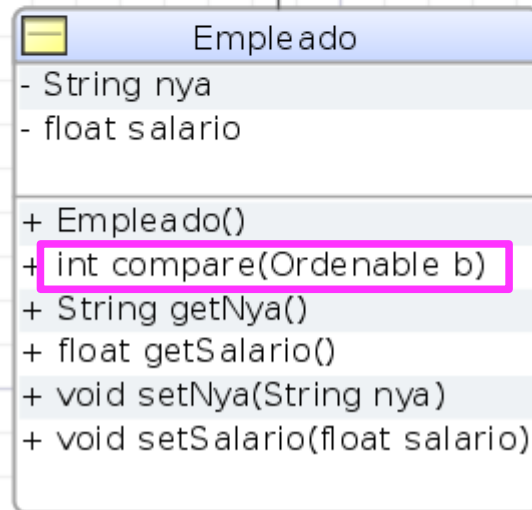
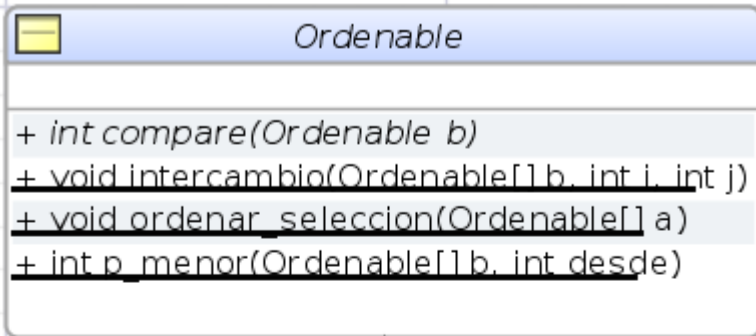
    public static void ordenar_seleccion(Ordenable[] a)
    {...}
}
```



The UML class diagram for the `Ordenable` class shows the following methods:

- `+ int compare(Ordenable b)`
- `+ void intercambio(Ordenable[] b, int i, int j)`
- `+ void ordenar_seleccion(Ordenable[] a)`
- `+ int p_menor(Ordenable[] b, int desde)`

# Ejemplo de clase abstracta



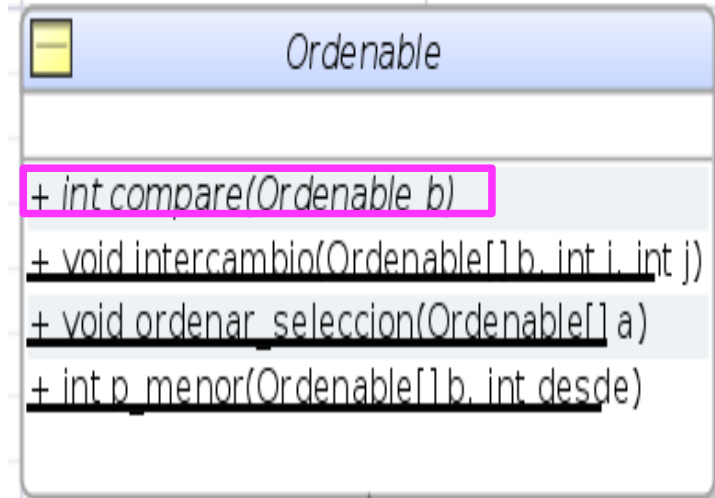
```
public class Empleado extends Ordenable
{
    private String nya;
    private float salario;

    public Empleado()
    {
        super();
    }

    @Override
    public int compare(Ordenable b)
    {
        int rta = 0;
        Empleado eb = (Empleado) b;
        if (getSalario() < eb.getSalario())
            rta = -1;
        else if (getSalario() > eb.getSalario())
            rta = 1;
        return rta;
    }
}
```

# Ejemplo de clase abstracta

```
public static int p_menor(Ordenable[] b, int desde)
{
    int N = b.length;
    Ordenable menor = b[desde];
    int posmenor = desde;
    for (int i = desde + 1; i < N; i++)
    {
        if (menor.compare(b[i]) > 0)
        {
            posmenor = i;
            menor = b[i];
        }
    }
    return posmenor;
}
```



Dentro de la clase abstracta `Ordenable`, el método concreto ***p\_menor(...)*** invoca al método abstracto ***compare(...)***

# Ejercicio Teórico practico

- Volver al ejemplo de Animal – Mamifero – Gato.
- Ampliarlo agregando más clases.
- Agregar métodos abstractos (que solamente puedan ser abstractos)
- Trabajar con alguna colección de Animales y ver cómo funciona el polimorfismo.