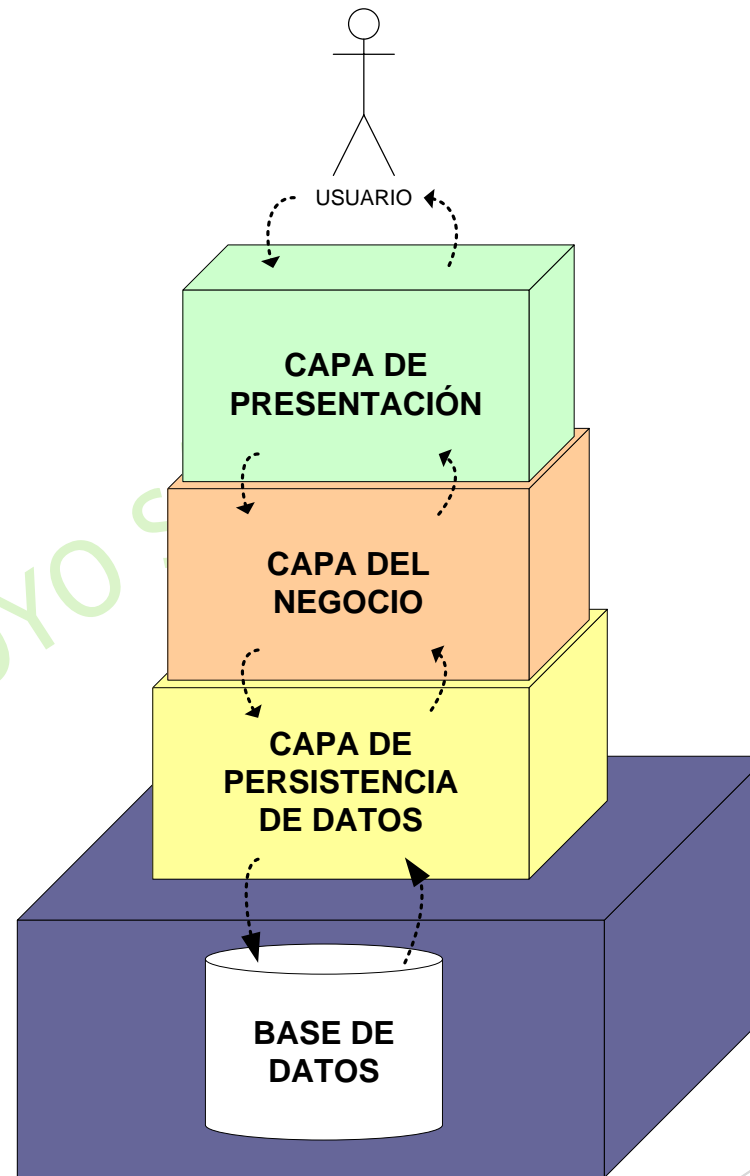


UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE JAVA



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

Índice

1. El cambio como constante. Arquitectura de una aplicación
2. Encapsulamiento
3. Constructores
4. Sobrecarga de métodos
5. Relaciones entre clases: Composición
6. Relaciones entre clases: Herencia
7. Polimorfismo
8. Interfaces

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

► 1 – El cambio como constante. Arquitectura de una aplicación

- En el mundo del desarrollo de software solo existe una **CONSTANTE**:

“TODO CAMBIA”

- Si tu aplicación no funciona bien o no hace lo que el usuario necesita está claro que **HABRÁ QUE CAMBIARLA**.
- Pero si tu aplicación funciona y satisface las necesidades del usuario entonces es muy probable que se pidan funcionalidades nuevas o que haya que hacerla más segura o actualizar las dependencias... vamos que sí o sí **HABRÁ QUE CAMBIARLA**.

¿Piensas que la página de Facebook que ves a día de hoy es la misma que se publicó en su primera versión?



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- ▶ Como desarrolladores de software tenemos que **MENTALIZARNOS DE QUE EL CAMBIO ES INEVITABLE**, buscando siempre “nadar a favor de la corriente” y nunca en contra. Esto significa que tendremos que tomar ciertas medidas en nuestro código para poder “gestionar correctamente” el cambio.
- ▶ La arquitectura de la aplicación, la adopción de una metodología Agile, los patrones de diseño, la refactorización del código para que se “lea” mejor, usar un criterio uniforme de nombrado, el uso de un repositorio de código como Git, la automatización de las pruebas... son técnicas que ayudan a “soportar mejor el cambio”.
- ▶ Todas estas técnicas buscan **mejorar la legibilidad del código y desacoplar las distintas partes del sistema** para que, si tenemos que modificar una de ellas, el impacto en el resto de piezas sea el menor posible y se realice con el menor coste posible.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- ▶ Algunas técnicas para la gestión del cambio las estudiaréis en este módulo, otras en el módulo de Entornos de Desarrollo, otras en la empresa... La cuestión es que es vital incorporarlas en la práctica diaria para poder siempre “nadar a favor de corriente”.

Arquitectura de una aplicación

- ▶ Antiguamente, se realizaba un desarrollo **MONOLÍTICO**, “todo en un bloque”, en el que todo el código de la aplicación estaba en un solo fichero. Esto provocaba un código difícil de manejar, de mantener, de localizar y solucionar los errores... **Un código que no estaba preparado para el cambio.**
- ▶ Actualmente, un proyecto de software se basa en una **arquitectura en capas**. Esto divide el software en partes más pequeñas y manejables. Además favorece el trabajo simultáneo de distintos equipos de trabajo.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- ▶ Esta forma de estructurar el software nos lleva a la metáfora de un **edificio donde cada planta es como una capa del software**.
 - ▶ Aunque esta metáfora no es 100% representativa porque podríamos empezar a construir las dos capas de arriba del software y todavía no tener los cimientos puestos...
 - ▶ La metáfora de la confección de un traje complejo me parece más acertada. Tenemos que juntar distintas piezas... “Probárselo” al cliente... Empezar por la parte de arriba/abajo, lo de dentro/fuera...
- ▶ En cualquier caso, **cada capa del software debe:**
 - ▶ Tener una responsabilidad BIEN DEFINIDA.
 - ▶ Interactuar sólo con las capas ADYACENTES.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

► Una posible arquitectura de tres capas:

► La capa de presentación se responsabiliza de:

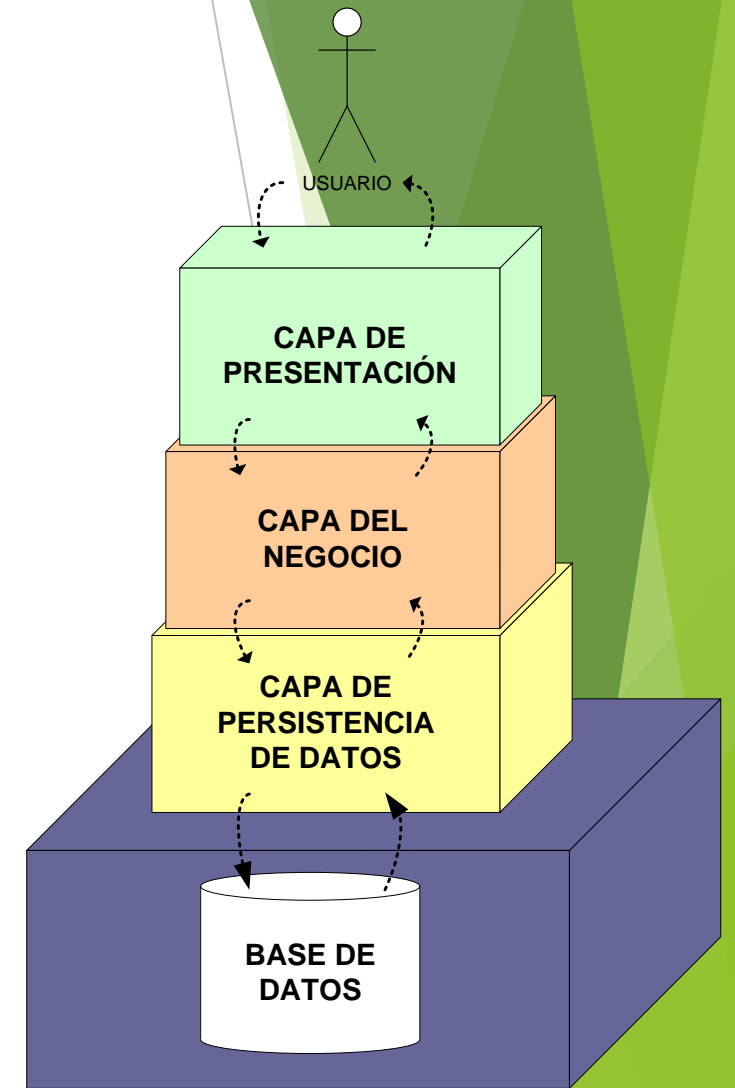
- Atender las peticiones del usuario y las traslada a la capa de negocio.
- Presentar al usuario los resultados que le devuelve la capa de negocio.

► La capa del negocio se responsabiliza de:

- Recibir las peticiones de la capa de presentación y realizar los cálculos/operaciones propias del negocio que se esté modelando, utilizando la capa de persistencia si es necesario.
- A esta capa también se le llama **capa de dominio**.

► La capa de persistencia se responsabiliza de:

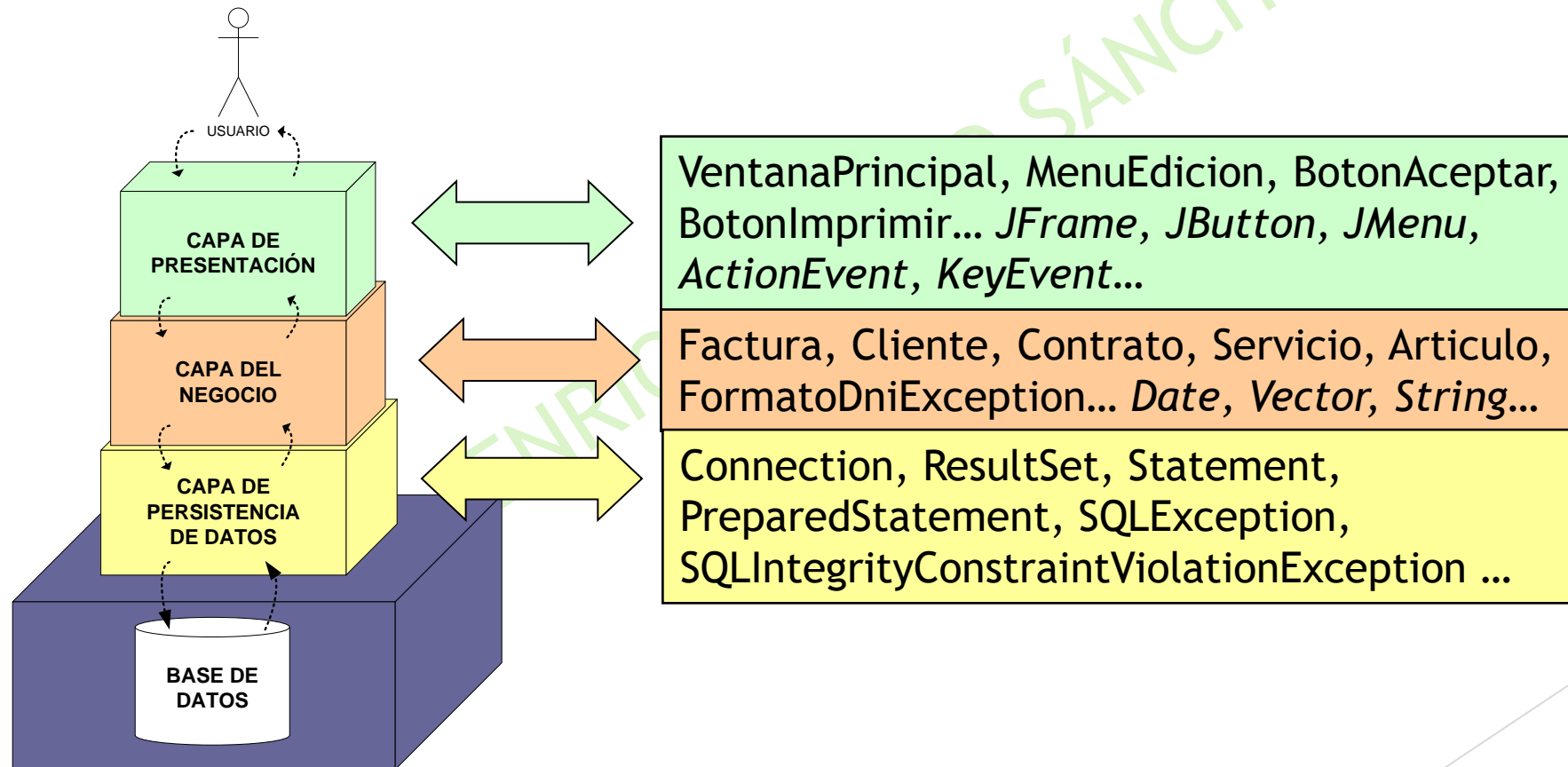
- Guardar y recuperar los objetos de la capa de negocio en la base de datos, realizando las adaptaciones necesarias para ello.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- En cada capa se crearán los objetos que sean necesarios para realizar la operación requerida por el usuario. Los objetos irán creándose, interactuando entre ellos y muriendo según vaya haciendo falta.

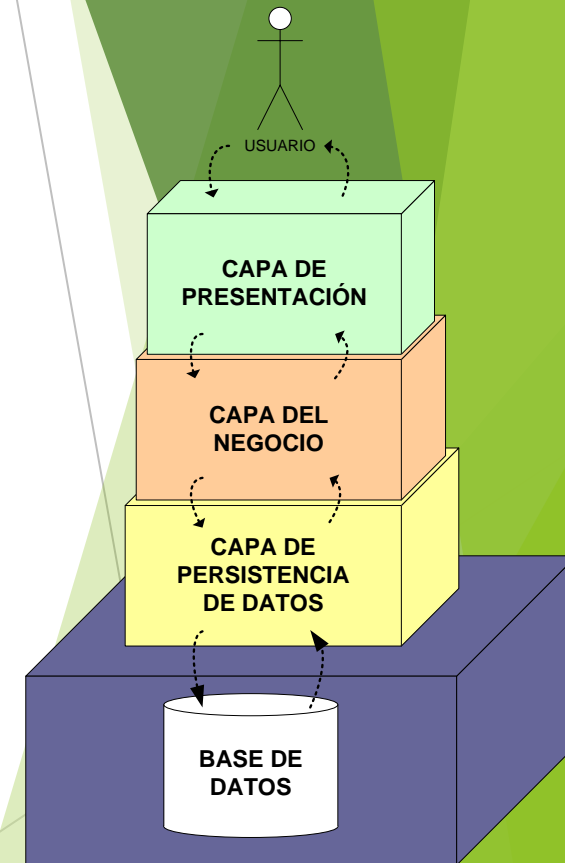


Pueden crearse tantas capas como sean necesarias (normalmente 3 o más)

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- ▶ Cada capa “**ofrece sus servicios**” a la capa superior y “**utiliza los servicios**” de la capa inferior. De este modo vamos ganando capacidades a medida que subimos por las capas.
- ▶ Ejemplo con una web de banca online:
 - ▶ La capa de presentación ofrece al usuario un **servicio de interacción con la aplicación**. Aquí podríamos hablar de objetos ventana, botón, lista desplegable, menús...
 - ▶ La capa de negocio ofrece a la capa de presentación **el servicio de realización de las operaciones bancarias** solicitadas por el usuario (transferencias, ingresos, pago de recibos...).
 - ▶ La capa de persistencia de datos ofrece a la capa de negocio **el servicio de almacenamiento y recuperación de la información** requerida para la realización de las operaciones bancarias (consulta los movimientos bancarios en la BD, añade un movimiento más al sistema...)



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

1 - El cambio como constante. Arquitectura de una aplicación

- ▶ En esta unidad didáctica **NO** vamos a realizar software por capas.
- ▶ Vamos a estudiar distintas herramientas de la orientación a objetos que se utilizan habitualmente para crear cada una de las capas para que sean “**resistentes al cambio**”.

Encapsulamiento

Sobrecarga de métodos

Herencia

Sobreescritura de métodos

INTERFACES

CONSTRUCTORES

Composición

POLIMORFISMO

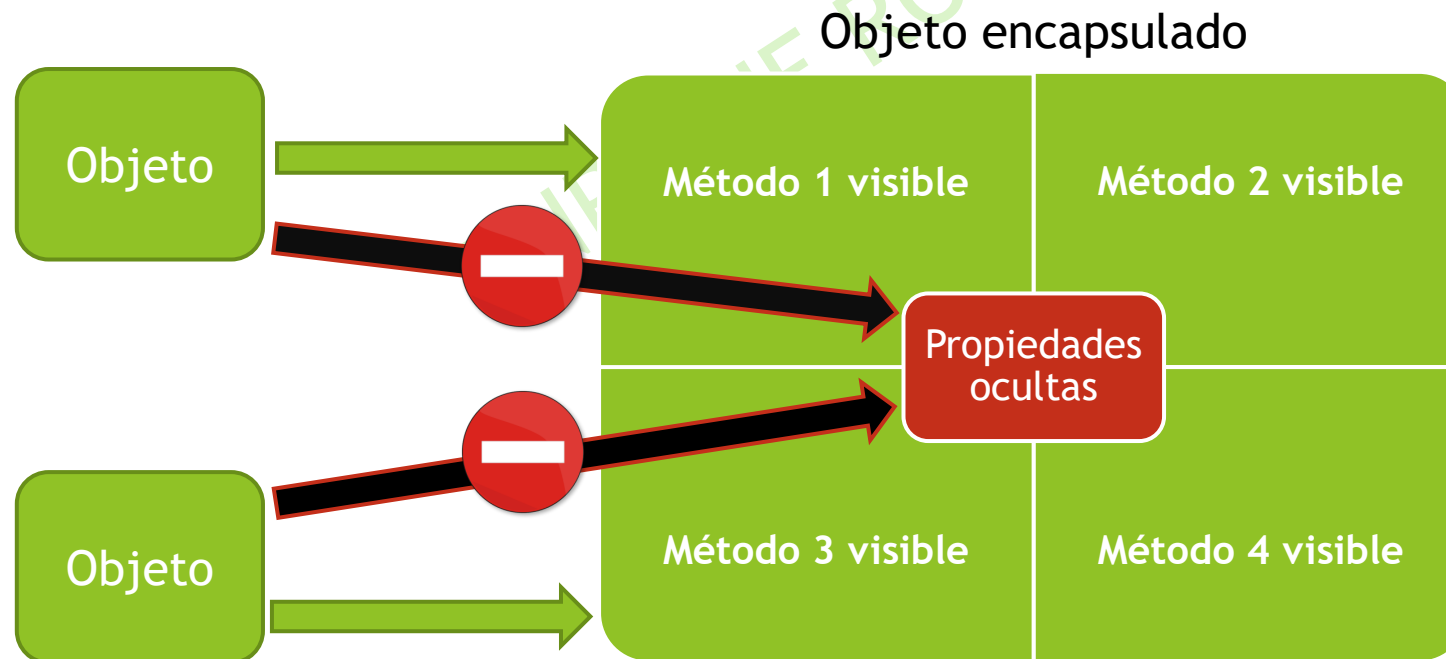
Clases abstractas

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

► 2 – Encapsulamiento

- El encapsulamiento es una técnica de diseño de clases que consiste en **ocultar el estado interno del objeto** (sus propiedades) y mostrar sólo los métodos.
- Es como si el objeto tuviera “**SUS SECRETOS**”, de modo que los objetos externos sólo pueden hacer uso de los métodos y nunca acceder directamente a las propiedades del objeto.



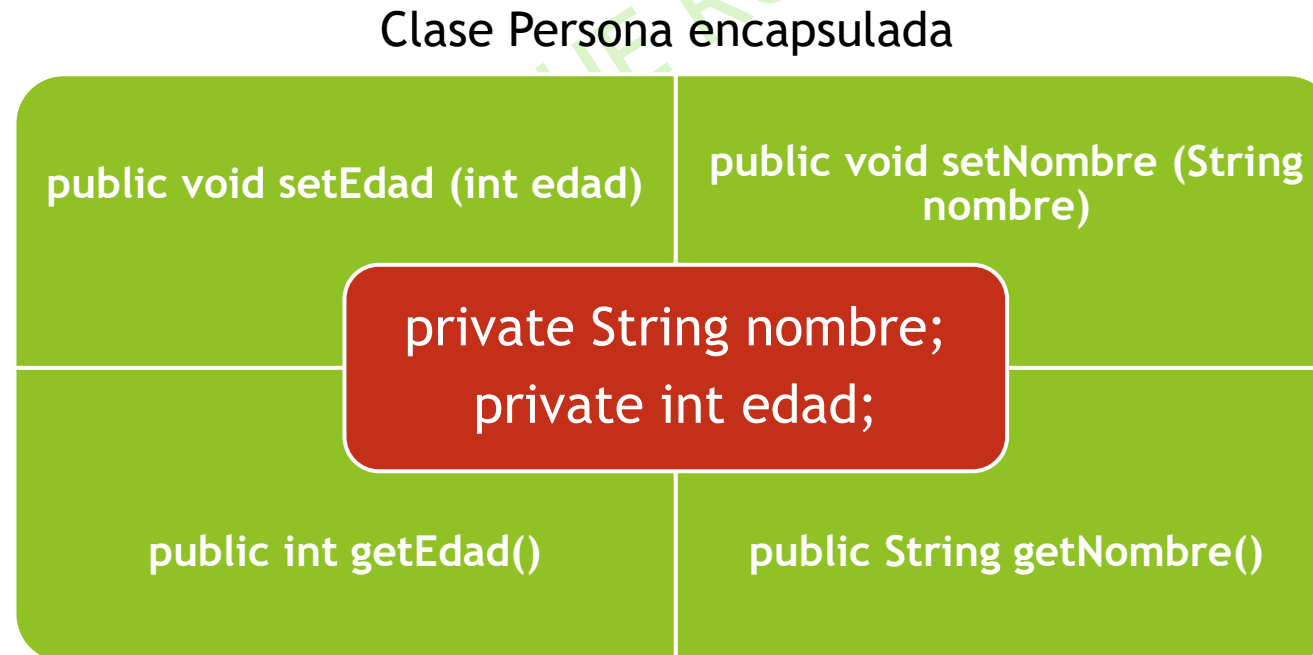
UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- Para realizar el encapsulamiento de una clase necesitamos usar los modificadores de acceso del lenguaje, que son:



- Normalmente, se usan **public** y **private**. Un ejemplo:



protected y
“friendly” los
estudiaremos un
poco más
adelante

Si hacemos que
nuestras
propiedades sean
private entonces
SOLO podrán
accederse desde
dentro de la clase
que las contiene

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- ▶ Hasta hoy podíamos “hackear” nuestros objetos desde fuera y saltarnos las validaciones que hacemos en los métodos. Ejemplo:

```
Bombilla b = new Bombilla(); // Nuestra bombilla SE FUNDE en su encendido nº 10
```

```
for (int i=1; i<=20; i++)
```

```
{
```

```
    b.apagar();
```

```
    b.encender();
```

```
}
```

```
if (b.fundida == true)
```

```
    b.fundida = false;
```

OBJETO HACKEADO.
Cambiamos el estado
interno a nuestra
conveniencia.



- ▶ El encapsulamiento impide que un objeto externo pueda manipular directamente el estado de nuestro objeto dejándolo en un estado incoherente. De este modo, conseguimos que nuestros objetos sean más robustos y confiables.

Descargamos del
repositorio
U3.P1.Encapsulamiento
y lo probamos.

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

► A PARTIR DE HOY, TODAS NUESTRAS CLASES DEBEN ENCAPSULARSE.

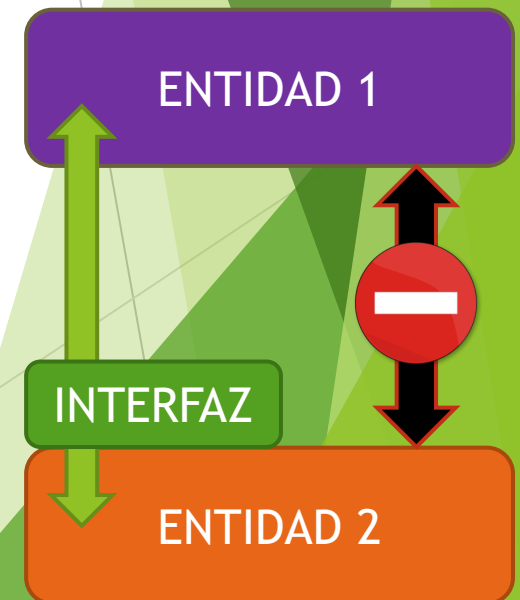
- Solo permitiremos el uso de propiedades públicas para crear constantes estáticas que puedan ser accedidas desde fuera. Ejemplo: Math.PI

► Concepto de interfaz de una clase/objeto:

- En general, una interfaz es un mecanismo de interacción o comunicación entre dos entidades que no sabrían “hablar” entre ellas directamente.

► Ejemplos:

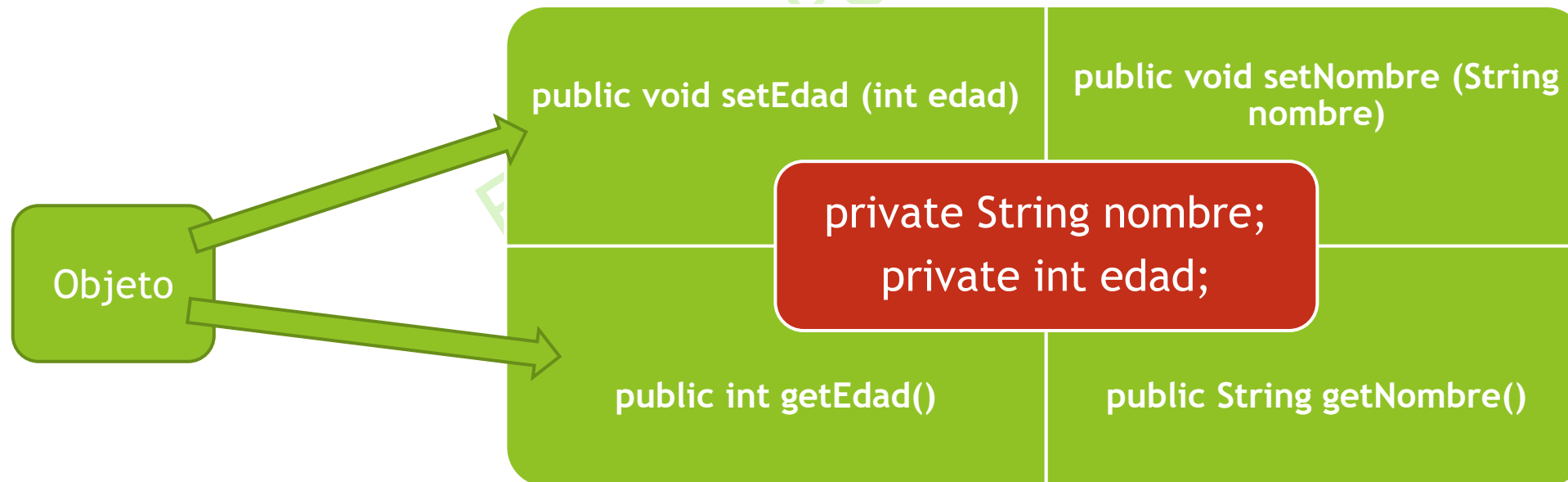
- El volante, los pedales, la palanca de cambio, el cuenta-revoluciones... todos estos elementos componen la INTERFAZ DE COMUNICACIÓN entre un humano y un vehículo.
- Una ventana, el puntero del ratón, botones, menús... todos estos elementos componen la INTERFAZ GRÁFICA DE COMUNICACIÓN entre un humano y una aplicación.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- ▶ Con el encapsulamiento, nuestros objetos **ocultan sus propiedades** y ofrecen al resto de objetos **una interfaz de acceso** a ellas mediante su lista de métodos públicos.
- ▶ Podemos decir entonces que un objeto/clase ofrece **un conjunto de servicios** al resto de objetos mediante su **interfaz de métodos públicos**.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- ▶ El modificador de acceso de paquete o “friendly”:
 - ▶ Si a una propiedad no se le pone ningún modificador de acceso (public, private...) entonces se dice que tiene accesibilidad “a nivel de paquete” o “friendly” (para los amigos o vecinos). Esto haría que esa propiedad se comportara como:
 - ▶ Public para las clases que estén en su mismo paquete.
 - ▶ Private para el resto de las clases.
- ▶ Ejemplo:

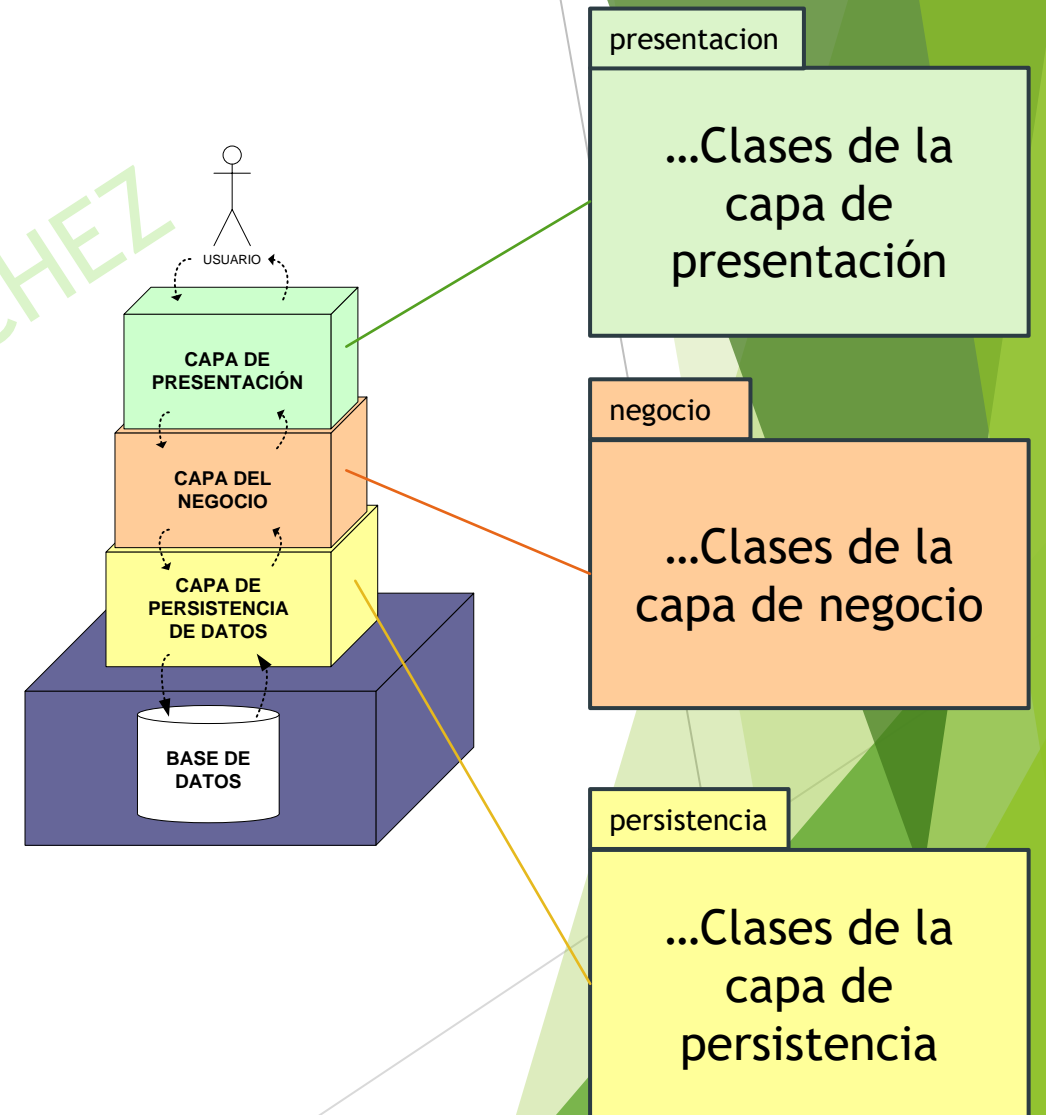
```
public class Persona {  
    String nombre;  
    int edad;  
    //...  
}
```



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- ▶ Normalmente las clases que pertenecen a un mismo paquete:
 - ▶ Están relacionadas entre sí.
 - ▶ Suelen cooperar entre ellas.
 - ▶ Han sido desarrolladas por un mismo equipo de desarrolladores.
- ▶ Todo esto nos lleva a establecer una relación de “vecindad” o “confianza” entre estas clases.
- ▶ El modificador “friendly” permite que ciertas propiedades/métodos sean accesibles para las “clases vecinas” pero no para el resto del mundo.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- ▶ El concepto de “encapsulamiento” también se puede aplicar a nivel de paquete.
 - ▶ En este caso nos interesa que algunas clases de un paquete sean accesibles “desde fuera” y otras no.
 - ▶ Cuando creamos una clase dentro de un paquete tenemos dos opciones:
 - ▶ Ponerla como **public**, de modo que sería visible “desde fuera” del paquete que la contiene.
 - ▶ Ponerla como “**friendly**”, de modo que sería visible por las clases que estén en su mismo paquete, pero no “desde fuera”.

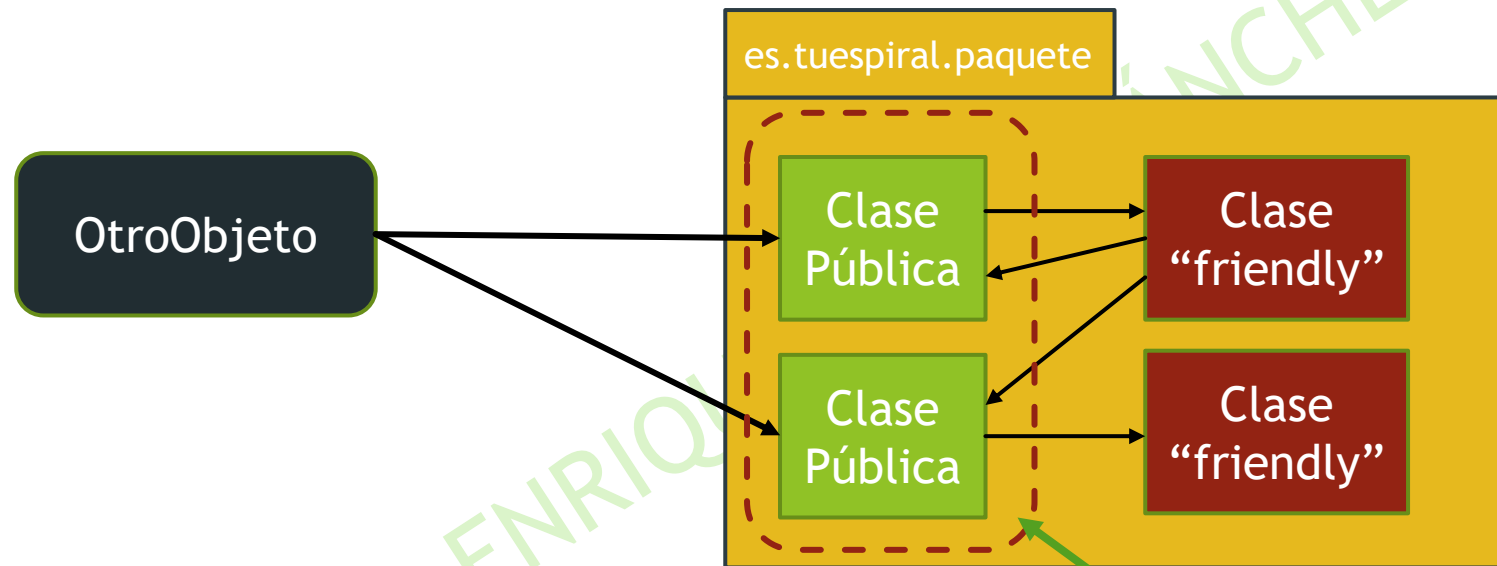
```
package es.tuespiral.paquete;  
public class PersonaPublica {  
    String nombre;  
    int edad;  
    ...  
}
```

```
package es.tuespiral.paquete;  
class PersonaPaquete {  
    String nombre;  
    int edad;  
    ...  
}
```


UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

2 - Encapsulamiento

- De este modo conseguimos que ciertas clases de un paquete sean públicas y “den la cara” y otras clases “se queden atrás” actuando como un apoyo o complemento a las clases públicas.



- Así que también podemos hablar de la **interfaz de un paquete** como el conjunto de clases públicas de dicho paquete.

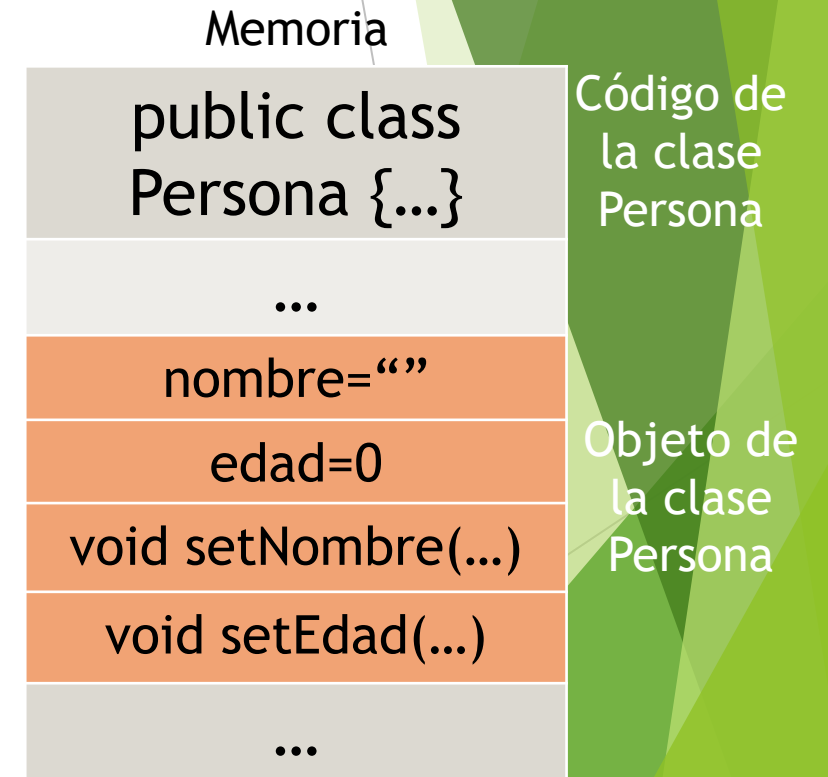
UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

3 - Constructores

► 3 – Constructores

- Los constructores son métodos que nos permiten crear un objeto a partir de la clase.
- Los estamos usando cada vez que hacemos:
 - ... = **new** Persona();
 - ... = **new** Bombilla();
 - ... = **new** Scanner(System.in);
- Ahora vamos a aprender a crearlos y a estudiar las reglas con las que funcionan.
- Veamos un ejemplo:

Constructor



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

3 - Constructores

- Vamos a crear un constructor que nos permita inicializar todas las propiedades de un objeto Persona a la vez.

```
public class Persona {  
    private String nombre, apellidos;  
    private int edad, altura;  
    private boolean casada;  
  
    public Persona(String nombre, String apellidos, int edad, int altura, boolean casada) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
        this.altura = altura;  
        this.casada = casada;  
    }  
  
    // Getters y Setters  
    public String getNombre() {  
        return nombre;  
    }  
}
```

this sirve para que la clase hable de sí misma, de sus propiedades y métodos. Se le llama autorreferencia

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

3 - Constructores

- Esto nos permite crear los objetos más cómodamente y los tenemos listos para usarse justo después de la llamada al constructor:

```
public class PruebaPersona {  
    public static void main(String[] args) {  
        Persona p = new Persona("Pedro", "López Haro", 25, 180, false);  
    }  
}
```

- Lo que hacíamos hasta ahora era mucho más largo y dejaba el objeto en un estado inicial incoherente o “no utilizable”:

Estado inicial
incoherente

Estado
“usable”
después de
los setters

```
Persona p = new Persona();  
p.setNombre("Pedro");  
p.setApellidos("López Haro");  
p.setEdad(25);  
p.setAltura(180);  
p.setCasada(false);
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

3 - Constructores

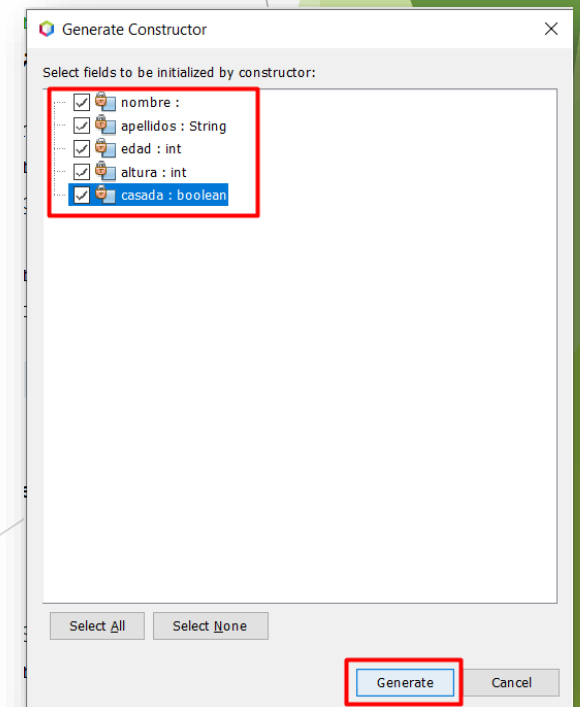
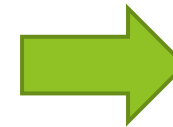
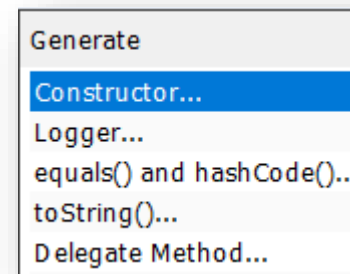
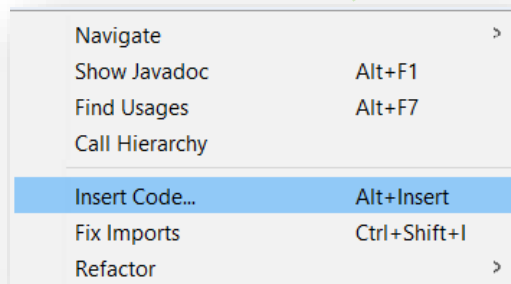
- ▶ La sintaxis de un constructor es la siguiente:

public NombreDeLaClase (lista de parámetros) { ... }

- ▶ Ejemplos:

- ▶ `public Bombilla (int potencia, String marca) { ... }`
- ▶ `public Triangulo (double lado1, double lado2, double lado3) { ... }`
- ▶ `public CajaRegistradora () { ... }` // Constructor sin parámetros

- ▶ NetBeans te lo pone fácil



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

3 - Constructores

- ▶ **El constructor “por defecto”:** cuando el desarrollador NO CREA ningún constructor para una clase, el lenguaje Java crea automáticamente un constructor sin parámetros. Este es el constructor que hemos usado hasta ahora y se le llama constructor “por defecto”.
 - ▶ **Metáfora:** al acusado de un juicio le ponen un “abogado de oficio” solo si no lleva su propio abogado.
- ▶ **OJO:** si el desarrollador crea un constructor, Java ya no te crea el constructor “por defecto”... esto provoca errores porque estamos tan acostumbrados a usarlo que tendemos a pensar que siempre está ahí.

Descargamos del repositorio
U3.P2.Constructores y lo probamos.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

4 - Sobrecarga de métodos


► 4 – Sobrecarga de métodos

- Se llama sobrecarga de métodos a **crear varias versiones de un método variando los parámetros de entrada**. Ejemplo:

```
public class Persona {  
    private String nombre, apellidos;  
    private int edad, altura;  
    private boolean casada;  
  
    public Persona(String nombre, String apellidos, int edad, int altura, boolean casada) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
        this.altura = altura;  
        this.casada = casada;  
    }  
  
    public Persona(String nombre, String apellidos) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
    }  
  
    // Getters y Setters
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

4 - Sobrecarga de métodos

- Esto nos ayuda a tener que recordar menos nombres de métodos, disponiendo de distintas “versiones” que comparten el mismo nombre.
- Podemos sobrecargar cualquier método, ya sea un constructor o un método “normal”. 
- Para crear las distintas “versiones”, podemos variar el número y tipo de los parámetros de entrada, pero nunca cambiar el parámetro de salida. Si intentamos añadir el siguiente método a la clase de la imagen de la derecha el IDE nos daría un error:

► public **short** suma (short a, short b)

No habría forma de saber si quieres usar una versión u otra

```
public class Sobrecarga {  
  
    public int suma(int a, int b) {  
        return a+b;  
    }  
  
    public int suma(int a, int b, int c) {  
        return a+b+c;  
    }  
  
    public int suma(short a, short b) {  
        return a+b;  
    }  
  
    public int suma(short a, short b, short c) {  
        return a+b+c;  
    }  
  
}
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

4 - Sobrecarga de métodos

- ▶ La sobrecarga de constructores es bastante frecuente.
- ▶ La sobrecarga de métodos “no constructores” no es tan usual. Generalmente se usa cuando una clase tiene que ofrecer una serie de operaciones a otras clases o tipo de datos.
 - ▶ **Ejemplo:** Clase `java.util.Arrays`
 - ▶ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>

```
binarySearch(byte[] a, byte key)
binarySearch(byte[] a, int fromIndex, int toIndex, byte key)
binarySearch(char[] a, char key)
binarySearch(char[] a, int fromIndex, int toIndex, char key)
binarySearch(double[] a, double key)
binarySearch(double[] a, int fromIndex, int toIndex, double key)
binarySearch(float[] a, float key)
binarySearch(float[] a, int fromIndex, int toIndex, float key)
binarySearch(int[] a, int key)
binarySearch(int[] a, int fromIndex, int toIndex, int key)
```

Esta clase ofrece distintas versiones de sus métodos para adaptarse a distintos tipos de datos y también para modificar su comportamiento interno (casos con los parámetros *fromIndex* y *toIndex*)

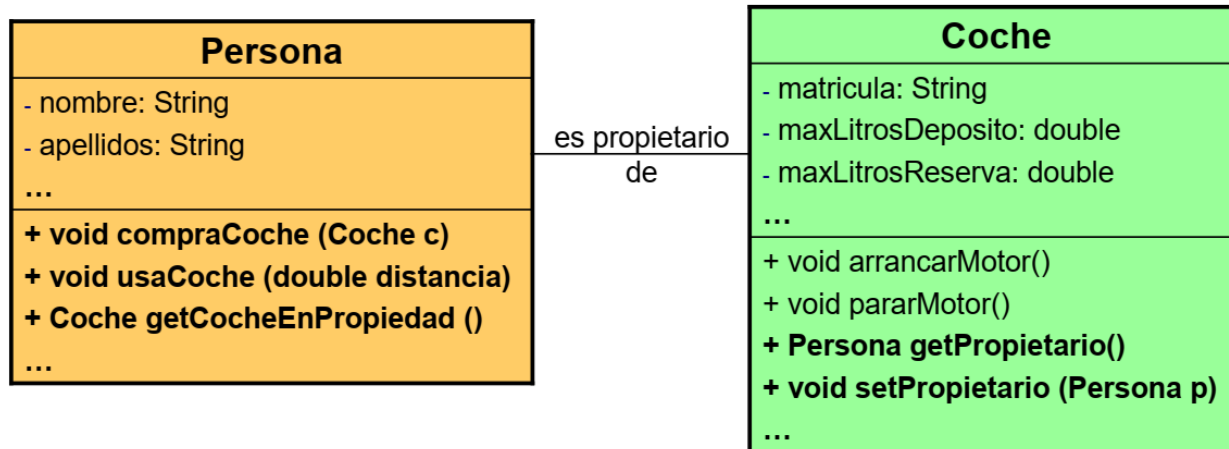
Realizar los ejercicios
1 al 2 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

5 - Relaciones entre clases: Composición

► 5 – Relaciones entre clases: Composición

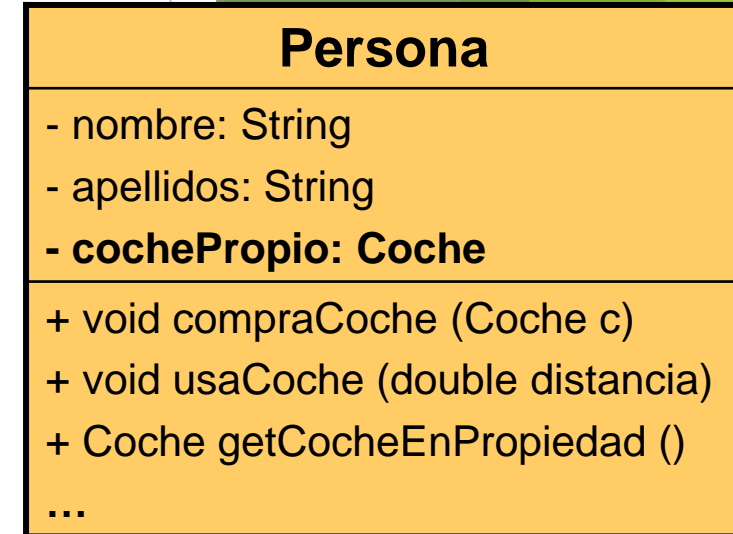
- Ahora vamos a estudiar técnicas de diseño de clases, cuando éstas se relacionan entre sí.
- El primer tipo de relación que vamos a estudiar es la composición, que es el más sencillo y el más utilizado. En este caso las clases presentan una relación que se puede expresar como: “está compuesto por”, “contiene”, “es propietario de”...
- Ejemplo:



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

5 - Relaciones entre clases: Composición

- ▶ En estos casos, la forma de llevar esto al código consiste en que el objeto “poseedor o contenedor” guarde una referencia al “objeto poseído o contenido”.
- ▶ Con este modelo podríamos “**navegar**” desde un objeto Persona hasta su objeto coche y manipular dicho objeto **pero no a la inversa**. Es decir, si yo tengo un objeto Coche, no podría conocer a su dueño... Sólo con añadir una referencia en la clase Coche a la Persona se arregla el problema:



Realizar el ejercicio
3 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

5 - Relaciones entre clases: Composición

- ▶ También nos podría interesar que una persona tenga varios coches y sería tan sencillo como añadir como propiedad una **colección de referencias a objetos coche** (con un array, por ejemplo).
- ▶ La composición es una técnica que se utiliza habitualmente. Además, está fuertemente relacionada con cómo se almacenan los datos de un objeto en una base de datos relacional.
- ▶ Ejemplo de tablas de base de datos que representan a dos clases relacionadas mediante composición.

Tabla persona		
dni	nombre	apellidos
27485658E	Juan	López Haro
87485159R	Ana	Smith
...

Tabla coche			
matricula	marca	modelo	dni_propietario
2342-RTF	Volvo	S30	27485658E
1264-RFZ	Seat	Ibiza	87485159R
...	



Persona	
- nombre: String	
- apellidos: String	
- coches: Coche[]	
+ void compraCoche (Coche c)	
+ void usaCoche (double distancia)	
+ Coche getCocheEnPropiedad ()	
...	

Realizar el boletín de ejercicios extra de "Composición"

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► 6 – Relaciones entre clases: Herencia

- Heredar algo, en el sentido general de la palabra, consiste en RECIBIR algo de alguien que, normalmente, tiene una relación de parentesco con el heredero.
- De algún modo los elementos heredados se REUTILIZAN. Así podemos hablar de la herencia de bienes y capitales o también de la herencia genética.
- En los lenguajes orientados a objetos existe el concepto de **herencia entre clases**, que es una potente herramienta de **reutilización código**.
- La reutilización de código nos permite desarrollar nuestras aplicaciones más rápidamente y mejorar su mantenimiento.

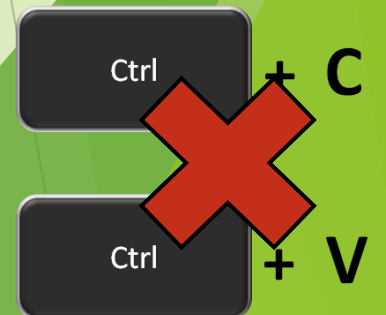


reduce
Reuse
RECYCLE

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Imagina que tenemos una clase **Persona** que hemos ido perfeccionando con el tiempo y que es capaz de guardar mucha información (nombre, apellidos, DNI, pasaporte, nacionalidad, teléfonos, dirección postal, dirección mail, fecha nacimiento, nivel de estudios...)
- ▶ Ahora nos interesa hacer dos clases más: **Empleado** y **Cliente**.
 - ▶ Un Empleado “es una” Persona pero que además tiene una fecha de contratación y un salario.
 - ▶ Un Cliente “es una” Persona que además tiene una fecha de alta en el sistema y un histórico de compras.
- ▶ ¿Cómo podríamos reutilizar la clase Persona en las nuevas clases?
 - ▶ **Opción cutre – Copiar/Pegar:** provocaría una duplicidad del código de modo que si, en un futuro, hay que cambiar algo de la Persona tendría que acordarme de cambiarlo también en el Empleado y el Cliente... **MALA PRÁCTICA, es fácil cometer errores... Poco resistente al CAMBIO...**

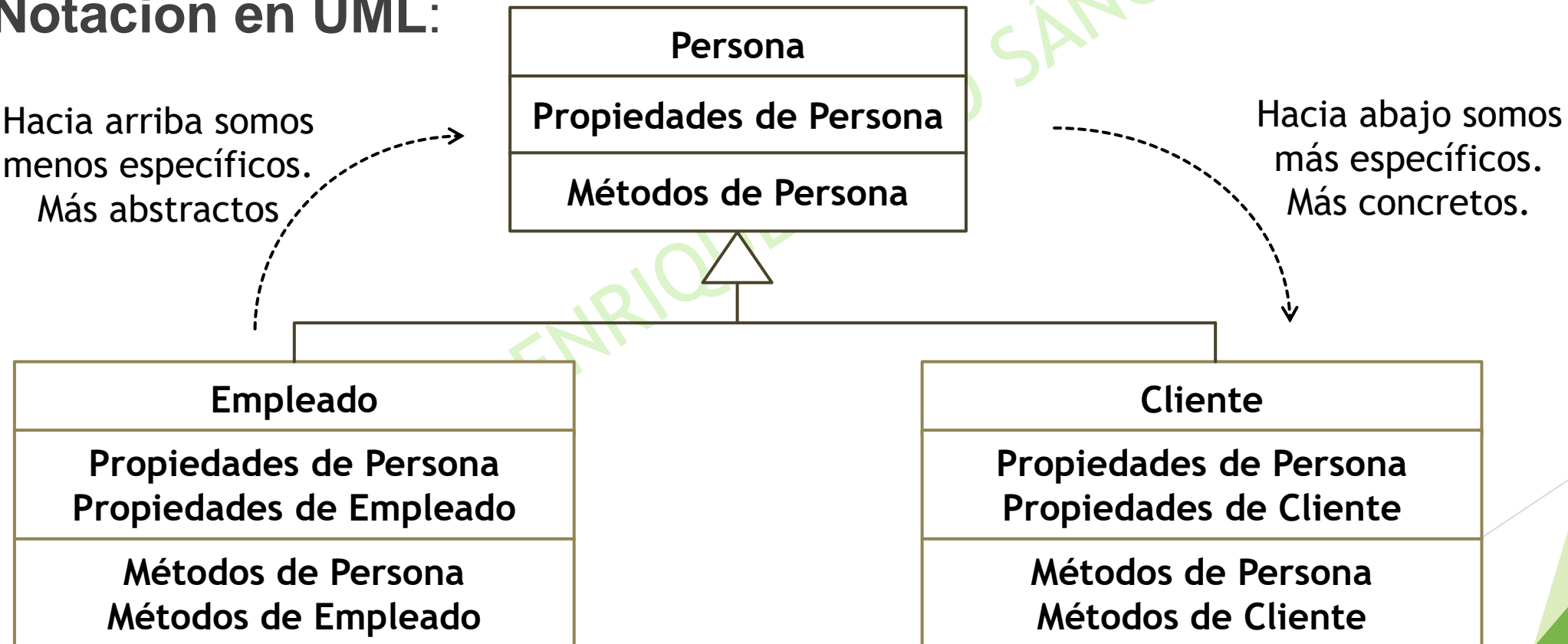


UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- **Opción ganadora – la HERENCIA:** la idea consiste en hacer que las clases Empleado y Cliente *hereden* de la clase Persona, de esta forma los cambios que hagamos en esta clase se propagan automáticamente a las clases herederas.

- **Notación en UML:**



La herencia es uno de los "conceptos estrella" del paradigma orientado a objetos

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ La sintaxis de la herencia en Java es muy sencilla:

```
public class Empleado extends Persona {  
    private double salario;  
    private LocalDate fechaContratacion;  
    // Getters y setters  
}
```

- ▶ De este modo tan sencillo conseguimos que la clase Empleado reciba toda la funcionalidad de la clase Persona.
- ▶ Decimos entonces que la clase Empleado “**extiende**” a la clase Persona o bien que “hereda” de esta. También se habla de “Superclase” y “subclases” o de clase base y clases derivadas o de clases madre/padre y clases “hijas”...

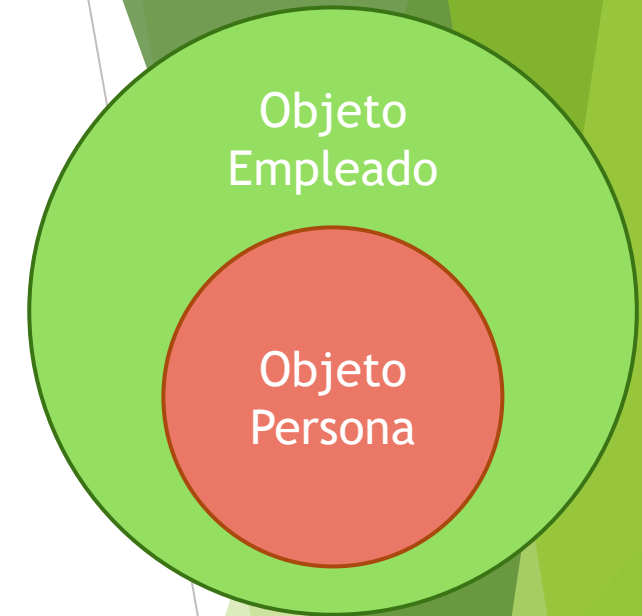
ENRIQUE ROYO SÁNCHEZ



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Cuando creamos un objeto de la clase Empleado tenemos que imaginar que éste tiene **un objeto Persona “encerrado” dentro**.
- ▶ El objeto Empleado podrá acceder o no a las propiedades y métodos de la clase Persona dependiendo de los **modificadores de acceso** que se hayan usado. Entonces:
 - ▶ Los **miembros public** de Persona serán accesibles por Empleado y por cualquier otro objeto de nuestra aplicación.
 - ▶ Los **miembros private** de Persona no serán accesibles ni por Empleado ni por cualquier otro objeto de la aplicación.
 - ▶ Los **miembros “friendly”** de Persona serán accesibles por las clases que compartan paquete con Persona, sin importar si hay herencia o no.
 - ▶ Los **miembros protected** de Persona serán accesibles por Empleado y también por aquellas clases que compartan paquete con Persona.



Cuando nos referimos a las propiedades y métodos de una clase indistintamente, hablamos de los **MIEMBROS DE LA CLASE**

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► Tabla resumen:

Un miembro en la superclase definido como:	¿Es accesible por una subclase?	¿Es accesible por otra clase en el mismo paquete?	¿Es accesible por otra clase fuera del paquete?
private	NO	NO	NO
“friendly”	Sólo si ambas están en el mismo paquete	SI	NO
protected	SI	SI	NO
public	SI	SI	SI

REGLA GENERAL:

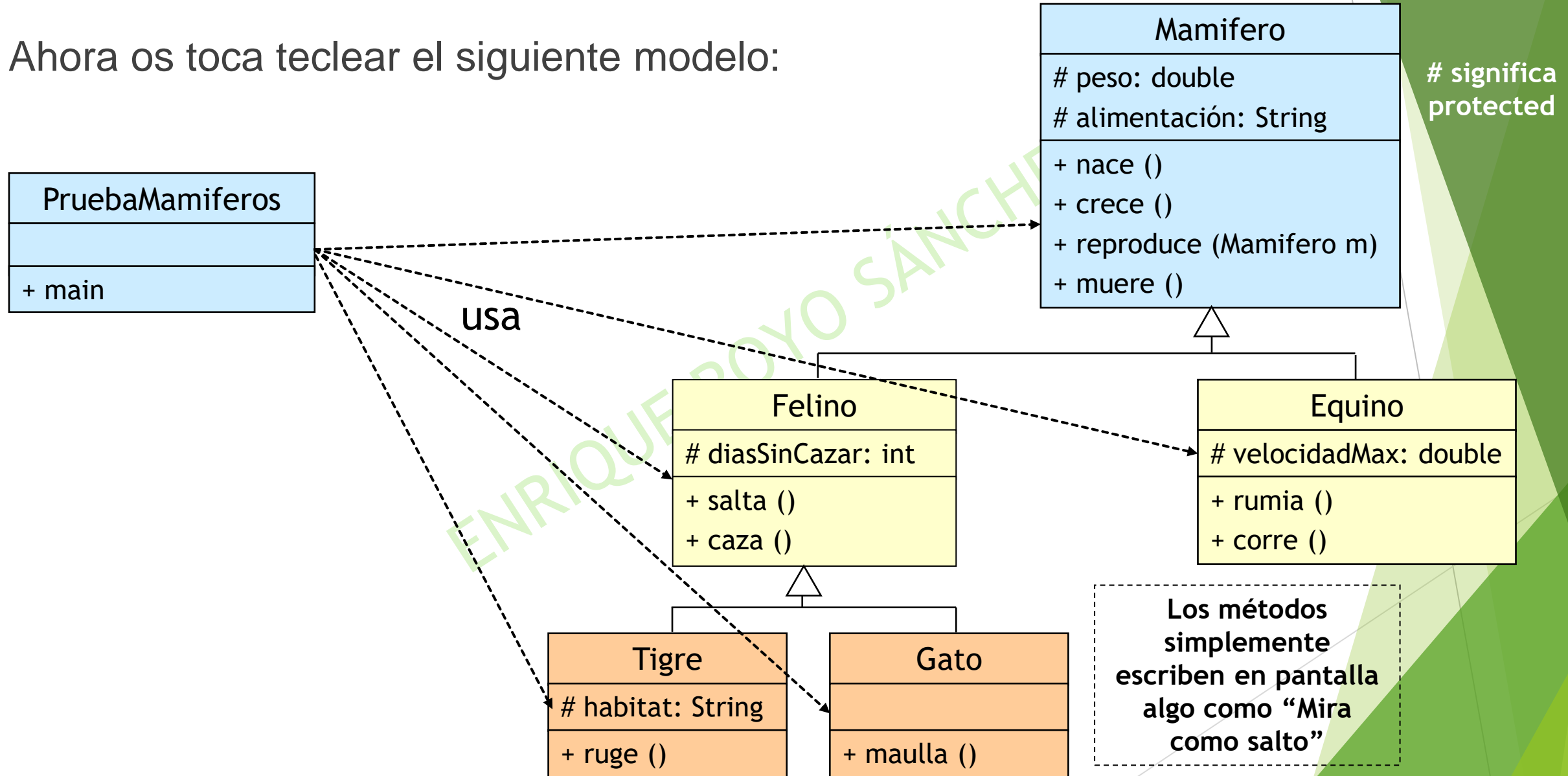
Las propiedades siempre deberían ser privadas porque representan el estado interno del objeto. El uso de cualquier otro modificador debería estar justificado.

Estudiamos el siguiente proyecto del repositorio:
U3.P3.Herencia

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Ahora os toca teclear el siguiente modelo:



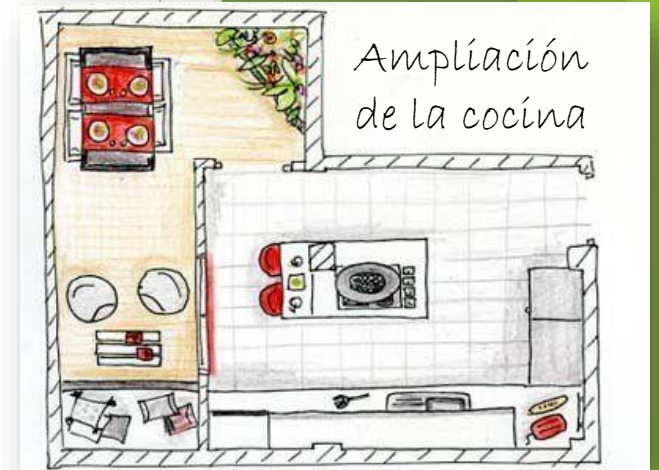
UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► ¿Cuándo nos viene bien usar la herencia?

Siempre que haya una **relación tipo “es un/a”** entre dos clases y, además, nos encontremos en al menos una de estas situaciones:

- Nos interesa que la subclase hija **amplíe la funcionalidad** de la superclase, añadiendo nuevos miembros a los ya existentes. Los ejemplos anteriores reflejan esta situación.
- Nos interesa que la subclase **modifique el comportamiento de ciertos métodos** de la superclase “reescribiendo el código”. Decimos entonces que la subclase **sobrescribe** (**override** en inglés: anular, invalidar, desestimar) uno o varios métodos de la superclase. Veamos un ejemplo de esto:



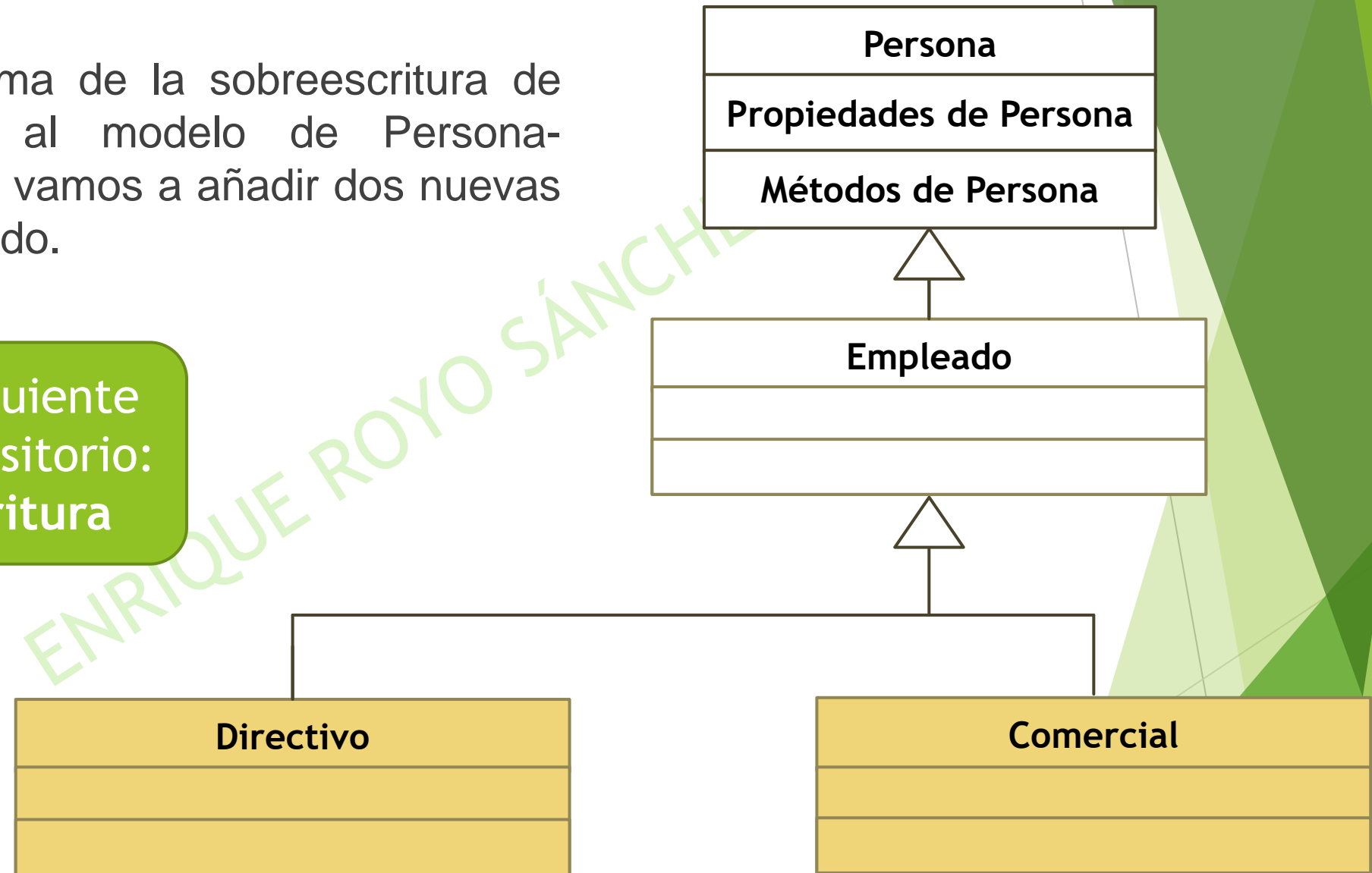
~~Código de la Superclase~~
Código de la Subclase

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Para aprender el tema de la sobreescritura de métodos volvemos al modelo de Persona-Empleado y además, vamos a añadir dos nuevas subclases de Empleado.

Estudiamos el siguiente proyecto del repositorio:
U3.P4.Sobrescritura



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Como podéis observar en el código anterior se mezclan los dos usos típicos de la herencia:
 - ▶ **Ampliación de una clase, añadiendo nuevos métodos:** por ejemplo, la clase `Director` añade los métodos `setBonusAnual` y `getBonusAnual`, que no estaban en la clase `Empleado`.
 - ▶ **Modificación del comportamiento de los métodos existentes en una clase:** las clases `Comercial` y `Director` sobrescriben el comportamiento del método `calculaNomina`.



Realizamos los ejercicios 4 y 5 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► El operador instanceof

- Con tantas clases hijas, madres, abuelas, hermanas esto es un lío. Ahora nos gustaría poder preguntar a un objeto algo como “¿eres un ejemplo de la clase X?”

- El operador **instanceof** hace algo parecido. Veámoslo:

```
Persona p = new Persona();
```

```
String texto = “Mi texto”;
```

```
if (p instanceof Persona) ➡ devuelve true
```

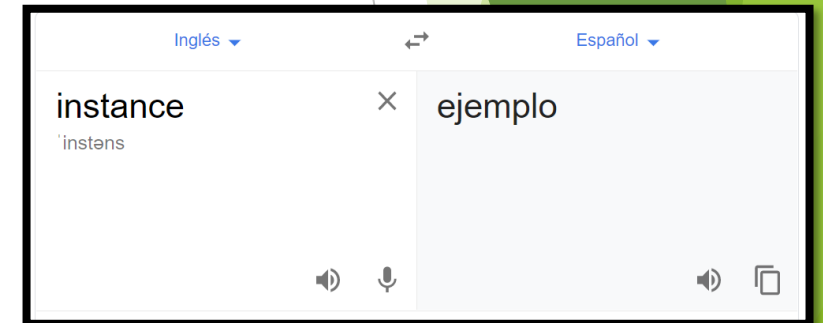
```
...
```

```
if (texto instanceof String) ➡ devuelve true
```

```
...
```

```
if (texto instanceof Persona) ➡ devuelve false
```

```
...
```



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► “Object está en todas partes”

- ¿Recuerdas cuando en la Unidad 1 decíamos que un principio de la programación orientada a objetos es que “**todo es un objeto**”?
- Pues los creadores de Java se lo tomaron al pie de la letra y crearon la clase **Object** que funciona como ancestro común de todas las clases. Todas las clases la tienen como superclase directa o indirectamente.
- Cuando creamos una clase que no hereda explícitamente de otra el **compilador le añade automáticamente** el texto en rojo:

```
public class Persona extends Object {...}
```

UD 1 - POO Y ELEMENTOS BÁSICOS DE JAVA

1 - Principios de la programación O.O.

► «Todo es un objeto»

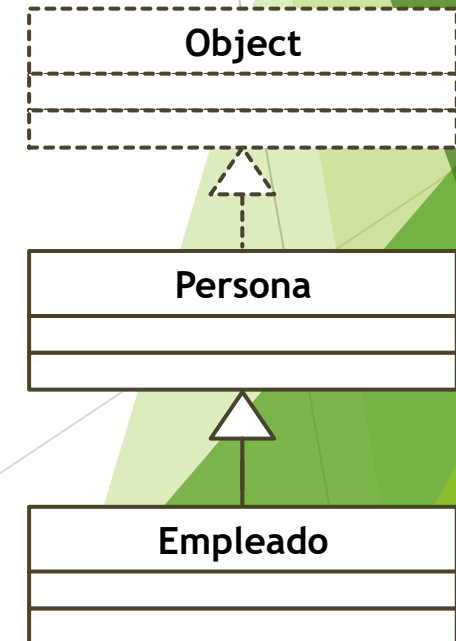
Los objetos se caracterizan por sus:

- Propiedades o atributos: *color, peso, nombre, apellidos, precio, altura...*
- Comportamientos o acciones que podemos realizar con ellos: *arrancar, parar, saltar, comer, abrir, cerrar, pagar, cobrar...*

Propiedades: color, textura, cantidad de aire...



Comportamientos: botar, rodar, girar...

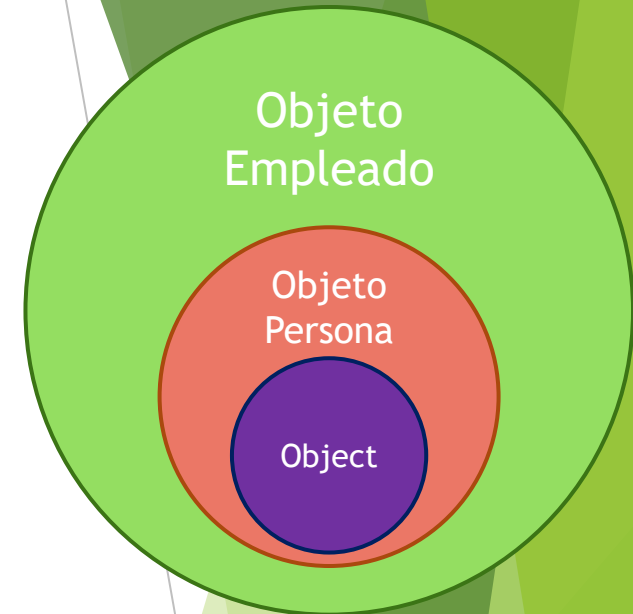


UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Como, además decíamos que cuando un objeto hereda de otro es como si lo tuviera encerrado dentro, entonces todos los objetos contienen a Object en su interior... como decíamos: “**Object está en todas partes**”
- ▶ Fíjate entonces las respuestas que da el operador **instanceof** en el siguiente ejemplo:

```
Empleado emp = new Empleado();  
if (emp instanceof Empleado) // devuelve true  
    System.out.println("Soy Empleado");  
If (emp instanceof Persona) // devuelve true  
    System.out.println("También soy Persona");  
If (emp instanceof Object) // devuelve true  
    System.out.println("Y también soy un Object");
```



Los objetos se estructuran POR CAPAS como las CEBOLLAS

¿y tú de quién eres?



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Así que para Java solo existen dos tipos de datos:

Objetos (hijos legítimos del “dios Object”)	Tipos primitivos (pobres fuera del “paraíso de Object”)
String, Scanner, Persona, int[] ...	int, double, boolean...

- ▶ Vamos a echar un ojo a la clase **Object** para ver qué es lo que tiene dentro.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>

- ▶ Analicemos algunos de sus métodos:



*El “Dios Object”
está en todas
partes*

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ El método **clone** permite crear una copia exacta del objeto (no lo vamos a usar)
- ▶ Los métodos **wait**, **notify**, **notifyAll** se utilizan en programación multihilo para “dormir” y “despertar” a uno o varios hilos de ejecución (la programación multihilo es fascinante pero se sale del alcance de este curso).
- ▶ El método **getClass** devuelve un objeto de la clase `Class` que te devuelve información sobre la clase a la que pertenece un objeto (lo usaremos dentro de poco...)
- ▶ El método **hashCode** devuelve un entero que es como el “DNI” de un objeto en memoria. Esto sirve para almacenar objetos en tablas Hash (lo estudiaremos en la siguiente unidad didáctica)



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ El método **equals (Object o)** recibe una referencia a un objeto y la compara con la referencia del objeto actual, devolviendo *true* si apuntan a la misma zona de memoria y *false* en caso contrario.
- ▶ El método **toString()** devuelve un texto que debe representar al objeto.
 - ▶ Este método suele ser **sobrescrito a menudo** incluyendo la información más relevante de las propiedades de un objeto o clase.
 - ▶ Además, este método se llama automáticamente cuando se quiere imprimir la representación textual de un objeto. Ejemplo:

```
Bombilla b = new Bombilla();
```

```
System.out.println(b); // Automáticamente se llama a b.toString()
```

¿a==b?

Estudiamos el código
U3.P5.DestripandoObject

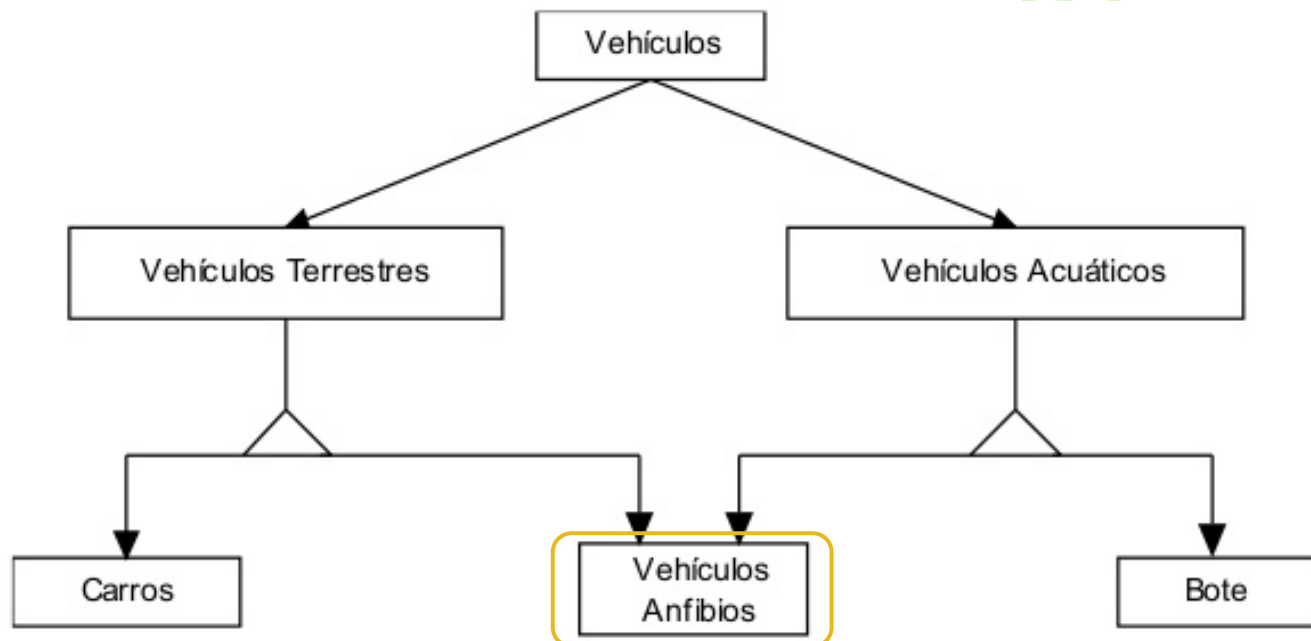
Realizamos el
ejercicio 6 del
boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► ¿Una clase puede tener dos padres/madres?

- Es decir, ¿una clase puede heredar de dos clases? ¿se permite la herencia múltiple?
- **Java no lo permite** pero hay otros lenguajes que sí. Esto permitiría resolver problemas como este:



Los objetos tienen familias monoparentales

Java no permite la herencia múltiple porque si hubiera un método común en ambas superclases (por ejemplo: `arrancar()`), el compilador no sabría qué código escoger...

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► El modificador final en clases y métodos

- Si eres programador en una empresa y “un poquito hacker” puedes atacar el sistema “desde dentro” o extraer información de forma ilícita del siguiente modo:
 - Creas una **clase “impostora”** que **herede de otra clase “legítima”** propia del sistema y modificas el código para que se use la clase impostora en vez de la original.
 - La clase impostora tendría los mismos métodos que la original pero éstos podrían ser sobrescritos para que hagan “otras cosas adicionales”.
- Para solucionar esto, Java permite utilizar el **modificador final sobre clases y métodos** para aportar seguridad a nuestro código. Veamos cómo funciona:



FINAL

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Si hacemos que una clase sea **final** entonces impedimos que se puedan sacar subclases de ella. La hacemos “estéril”. Sintaxis:

```
public final class ClaseEsteril {  
    ...  
}
```

- Si se intenta heredar de una clase declarada como **final**, el compilador lo marcará como un error y no se podrá generar el *bytecode* del programa.
- La API de Java está llena de clases declaradas como **final** para dotar de más seguridad al código generado.
 - Ejemplos: **String**, **Math**, **Scanner**...



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Otra opción menos radical de “securizar” nuestro código consiste en “esterilizar” algunos métodos pero permitiendo que la clase pueda tener hijos. Es decir, aplicamos **final algunos métodos para que no puedan ser sobrescritos**. Ejemplo:

```
public class ClaseFertil {  
    public final void esteMetodoNoSePuedeSobrescribir() {...}  
    ...  
}
```

```
public class ClaseHija extends ClaseFertil {  
    // Si lo intentamos sobrescribir el compilador nos da un error  
    public void esteMetodoNoSePuedeSobrescribir() {...}  
}
```



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

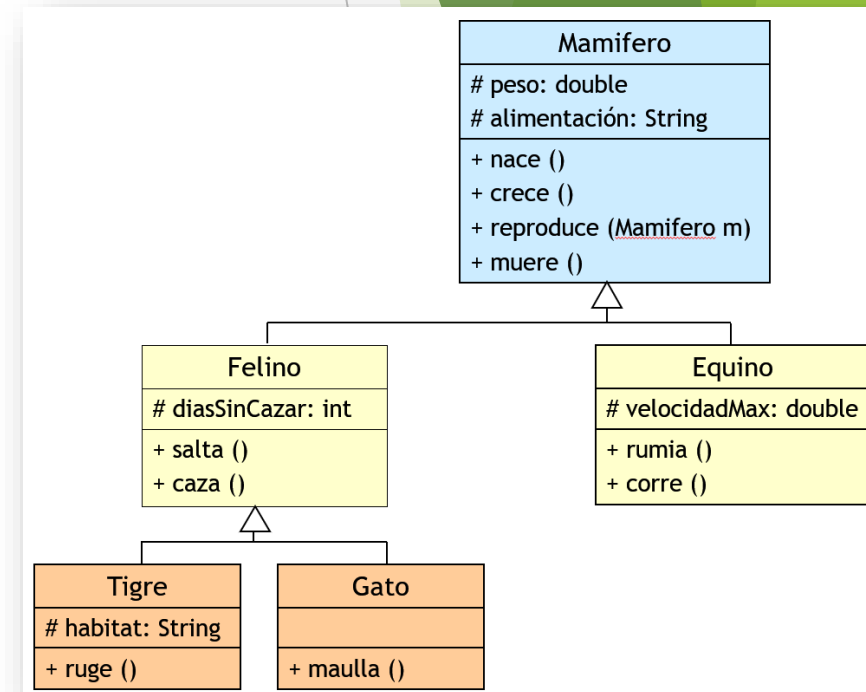
► El modificador *abstract* en clases y métodos

- Cuando usamos una jerarquía de clases, normalmente las clases superiores suelen ser conceptos abstractos que **no tienen una correspondencia con ningún objeto real**.

- **Ejemplo:** en el mundo real existen ejemplares de tigres o de gatos pero existe ningún ejemplar de “felino” o “mamífero”. Estos son conceptos que nos permiten identificar a un colectivo de seres vivos que tienen unas características comunes pero que no tienen “existencia física”.

- Entonces podría ser interesante “prohibir” la creación de objetos de ciertas clases porque son “demasiado abstractas”. Es decir, lo que queremos prohibir es:

Felino f = **new** Felino();
Mamifero m = **new** Mamifero;



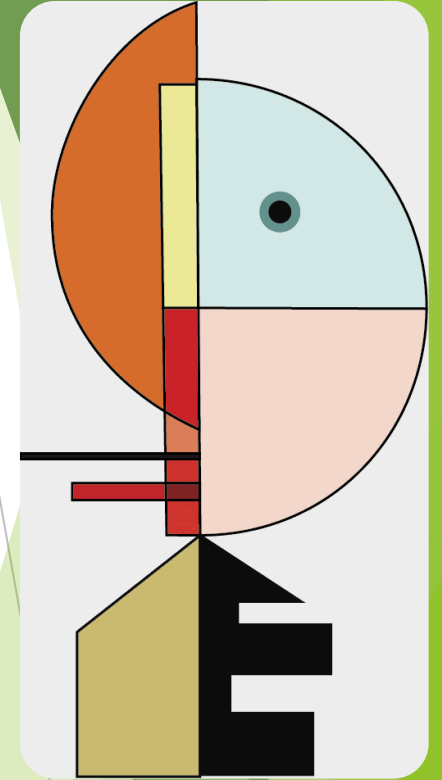
UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- El modificador **abstract** aplicado a clases impide que se puedan crear objetos a partir de ellas. La sintaxis es:

```
public abstract class Mamifero {  
    // Propiedades  
    // Métodos con su código  
}
```

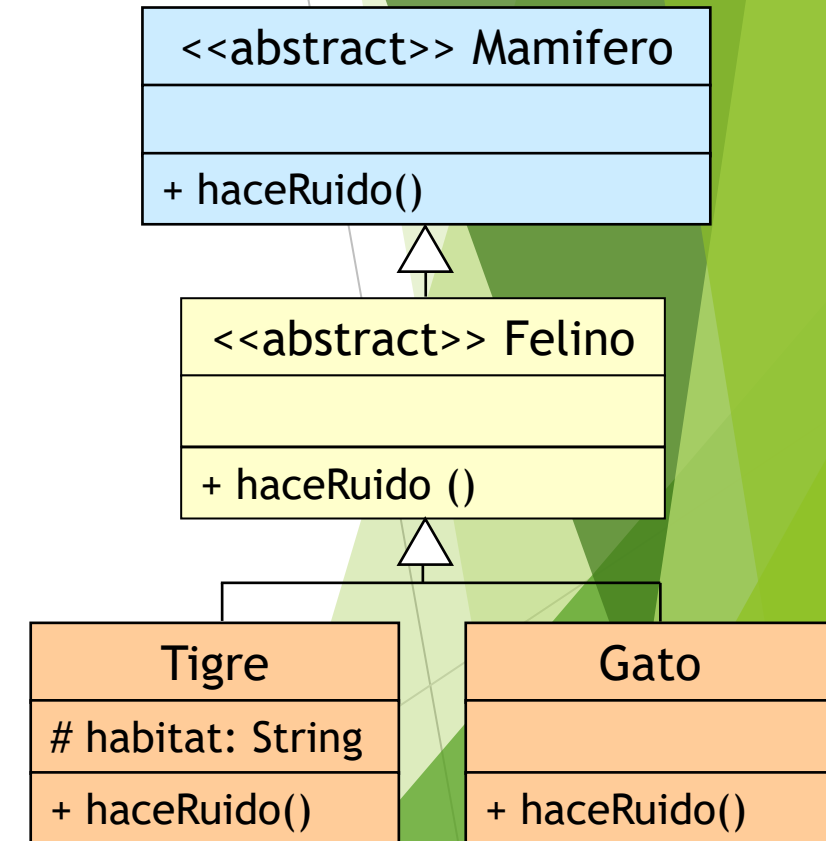
- Esta clase podría ser extendida por otras clases hijas pero **nunca podríamos crear un objeto de ella** con un *new Mamifero()*.
- De este modo, si queremos usar el código escrito en la clase Mamifero **estamos obligados a sacar una subclase de ella**.
 - Esta nueva subclase también podrá ser abstracta (public abstract class Felino) o “concreta” (public class Tigre)



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- ▶ Vamos a darle una última vuelta de tuerca a este concepto. ¿Cómo escribo el código de un método como *haceRuido()* cuando nunca crearemos un objeto de la clase Mamifero o de la clase Felino?
 - ▶ Sabemos que un gato hace “Miau” y un tigre hace “Roaaaar” pero ¿qué ruido hace un Mamifero?
 - ▶ Digamos que la clase Mamifero es tan abstracta que no tenemos información suficiente para escribir el código de *haceRuido()*
- ▶ Necesitamos un mecanismo de poder decirle al compilador que **NO vamos a escribir el código de un método porque queremos que lo hagan las subclases.**



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Esto se hace mediante el modificador **abstract** aplicado a **métodos**. Veamos la sintaxis:

```
public abstract class Mamifero {
```

```
// ...
```

```
    public abstract void haceRuido();
```

```
}
```

- Los métodos abstractos terminan en ; y no llevan { }
- Si una clase **tiene 1 o más métodos abstractos entonces debe ser declarada como abstracta**.
- Sin embargo, pueden existir clases declaradas como abstractas pero que no contengan ningún método abstracto.

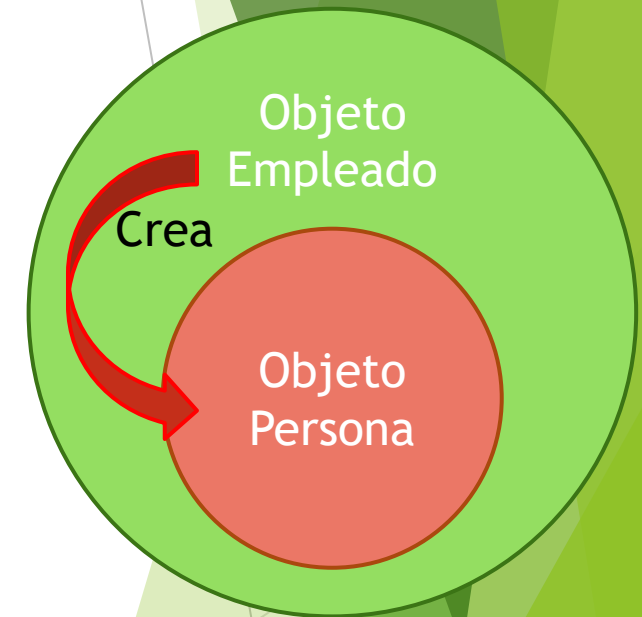
Realizamos los ejercicios 7-8 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► La herencia y los constructores

- Sabemos que en la herencia tenemos un objeto de la superclase “encerrado” en otro objeto de la subclase, entonces **¿cómo se crean ambos objetos? ¿creamos primero a la Persona y después al Empleado? ¿O al revés?**
- Siempre crearemos los objetos usando el constructor de la subclase (Empleado) y este constructor será **el responsable** de crear el objeto de la superclase (Persona).
- El tema de los constructores y la herencia es “delicado” porque el lenguaje Java **“nos intenta ayudar a su manera” insertando cierto código automáticamente**, que más de una vez nos lía más que otra cosa. Veamos cómo se hace todo el proceso.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

► Tomemos un ejemplo sencillo:

```
public class Persona {  
    private String nombre, apellidos;  
  
    public Persona() {  
        nombre = "";  
        apellidos = "";  
    }  
    // getters y setters  
}
```

El objeto Empleado llama al constructor de la superclase como primera línea.
Si el programador no la escribe, Java la pone por nosotros.

```
public class Empleado extends Persona {  
    private final double SALARIO_MINIMO = 1050;  
    private double salario;  
  
    public Empleado() {  
        // La primera línea del constructor de la subclase debe  
        // ser una llamada al constructor de la superclase.  
        // Si el desarrollador no la pone explícitamente, Java  
        // añadiría esta línea:  
        super();  
        salario = SALARIO_MINIMO;  
    }  
}
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- Los problemas se producen cuando la versión del constructor que se llama desde la subclase **NO EXISTE en la superclase**. Ejemplo:

```
public class Persona {  
    private String nombre, apellidos;  
  
    public Persona(String nombre, String apellidos) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
    }  
    // getters y setters  
}
```

**No existe el constructor
public Persona()**

Recordamos que Java sólo lo crea automáticamente si el desarrollador no crea ningún constructor.

```
public class Empleado extends Persona {  
    private final double SALARIO_MINIMO = 1050;  
    private double salario;  
  
    public Empleado() {  
        // La primera línea del constructor de la subclase debe  
        // ser una llamada al constructor de la superclase.  
        // Si el desarrollador no la pone explícitamente, Java  
        // añadiría esta línea:  
        super();  
        salario = SALARIO_MINIMO;  
    }  
}
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

6 - Relaciones entre clases: Herencia

- La solución en estos casos es que el desarrollador escriba explícitamente la versión correcta del constructor de la superclase:

```
public class Persona {  
    private String nombre, apellidos;  
  
    public Persona(String nombre, String apellidos) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
    }  
    // getters y setters  
}
```

Descargamos y probamos
U3.P6.HerenciaConstructores

Tenemos que escribir explícitamente la versión a usar.
Si no lo hacemos, Java insertaría automáticamente la línea `super()` y fallaría.

```
public class Empleado extends Persona{  
    private final double SALARIO_MINIMO = 1050;  
    private double salario;  
  
    public Empleado(String nombre, String apellidos) {  
        // Ahora debemos escribir explícitamente la llamada  
        // a la versión correcta del constructo de la  
        // superclase:  
        super(nombre, apellidos);  
        salario = SALARIO_MINIMO;  
    }  
}
```

Realizamos el
ejercicio 9 del
boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

7 - Polimorfismo

► 7 – Polimorfismo

- Sabemos que la herencia de clases nos permite reutilizar nuestro código ampliándolo y modificándolo. Pero además la herencia nos aporta otra ventaja más: **el polimorfismo**.
- Algo **polimorfo** es algo que se puede presentar de varias formas. De ahí las imágenes con la plastilina...
- También podríamos hablar de que cada uno de nosotros “somos polimorfos” en el siguiente sentido: yo soy profesor/a para mis alumnos/as. Contribuyente para Hacienda. Arrendador/a para mi inquilino. Conductor/a para la DGT. Padre/Madre para mis hijas. Hijo/a para mis padres...
- Este es el sentido de polimorfismo que vamos a aplicar sobre una jerarquía de clases.



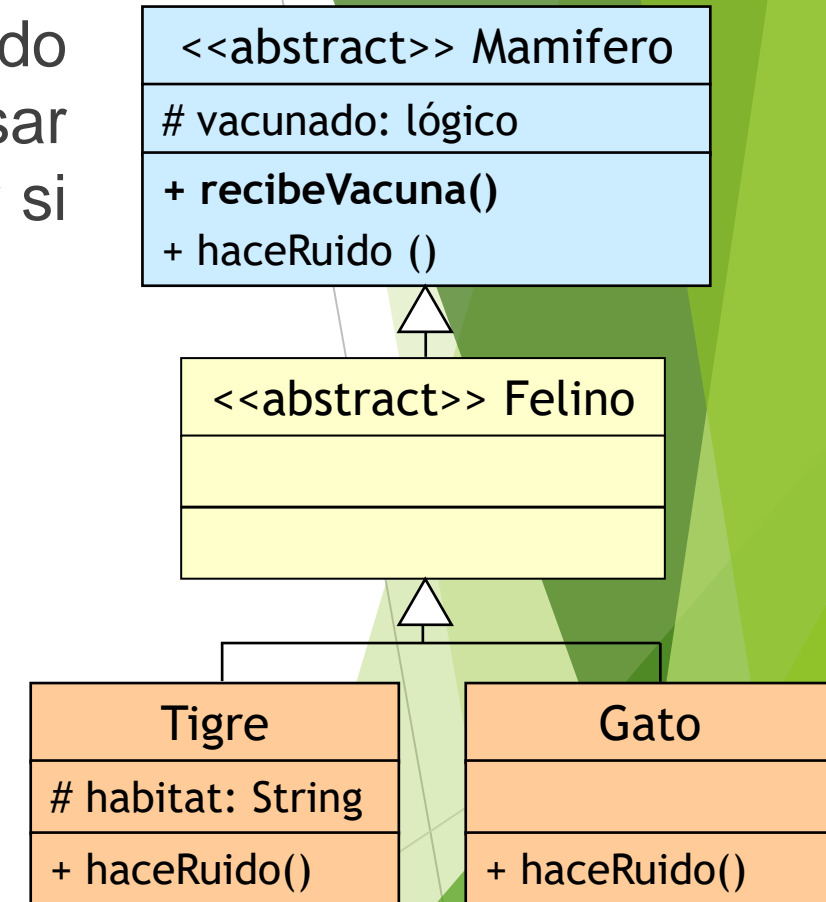
UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

7 - Polimorfismo

- En una jerarquía de clases, las clases se relacionan formando una especie de “familia”. Y, en ocasiones, nos puede interesar tratar a todos los miembros de la familia por igual, sin importar si es “el abuelo” o “un nieto”... Observa el siguiente código:

Un veterinario tiene que ponerle la vacuna a todos los Mamiferos de un zoo, sin importar de qué tipo sean.

```
public class Veterinario {  
    public static void main(String[] args) {  
        Tigre t1 = new Tigre();  
        Tigre t2 = new Tigre();  
        Gato g1 = new Gato();  
        Gato g2 = new Gato();  
  
        Mamifero[] animales = new Mamifero[4];  
        animales[0] = t1;  
        animales[1] = t2;  
        animales[2] = g1;  
        animales[3] = g2;  
  
        for(Mamifero a : animales)  
            a.recibeVacuna();  
    }  
}
```



Código disponible en
U3.P7.PolimorfismoAnimal

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

7 - Polimorfismo

► En este código hay varias cosas interesantes:

- Se crea un array de referencias a Mamifero, aunque sabemos que esta clase es abstracta y que no podremos crear objetos de ella.
- Llenamos el array con objetos Tigre y Gato porque heredan de Mamifero, de modo que “saben comportarse como Mamifero”, porque estos objetos son Tigre/Gato pero también son Felino y Mamifero.
- **Esto es el polimorfismo**, un objeto puede tener el comportamiento propio de la clase a la que pertenece pero también puede comportarse como un objeto de cualquiera de sus superclases.
- Tener un array de Mamifero en vez de dos arrays, uno de Tigres y otro de Gatos, es **una gran ventaja** porque nos permite aplicar un tratamiento común (vacuna) a todos los objetos de forma cómoda. Imagina el problema que generaría no poder usar el polimorfismo y que hubiera 20 subclases.

```
public class Veterinario {  
    public static void main(String[] args) {  
        Tigre t1 = new Tigre();  
        Tigre t2 = new Tigre();  
        Gato g1 = new Gato();  
        Gato g2 = new Gato();  
  
        Mamifero[] animales = new Mamifero[4];  
        animales[0] = t1;  
        animales[1] = t2;  
        animales[2] = g1;  
        animales[3] = g2;  
  
        for(Mamifero a : animales)  
            a.recibeVacuna();  
    }  
}
```

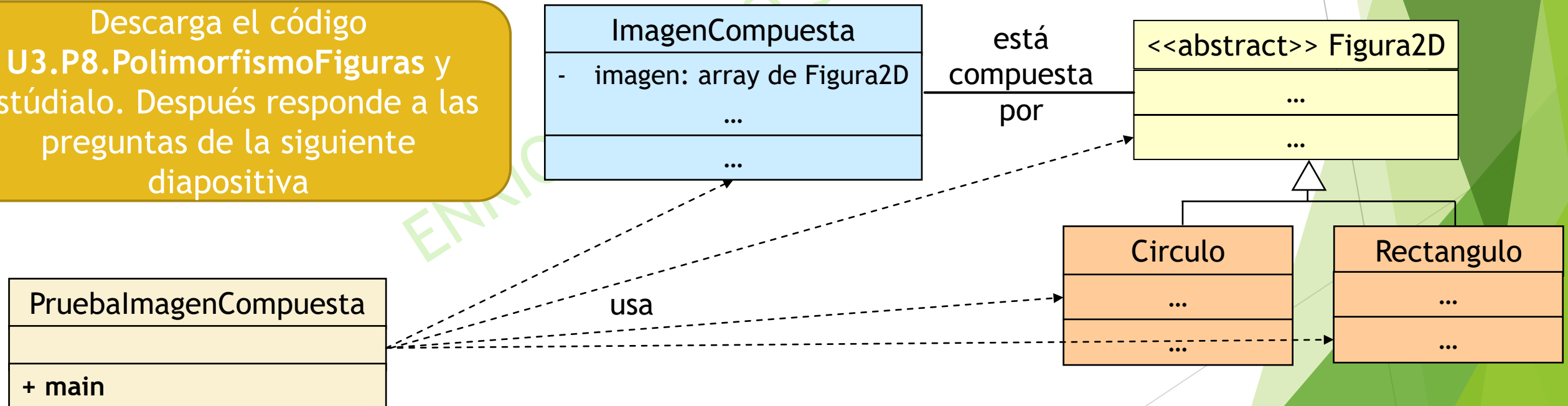
*A los ojos del
veterinario todos los
objetos tigres/gatos se
comportan como
Mamiferos*

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

7 - Polimorfismo

- Podríamos decir entonces que las **subclases también “heredan el tipo” de sus superclases**. El Gato “es Gato”, pero también “es Felino” y también “es Mamifero”... “heredando estos tipos” de sus superclases.
- Vamos a poner otro ejemplo con el siguiente diagrama UML:

Descarga el código
U3.P8.PolimorfismoFiguras y
estúdialo. Después responde a las
preguntas de la siguiente
diapositiva



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

7 - Polimorfismo

- ▶ ¿Qué significa que la clase Figura2D sea *abstract*?
- ▶ ¿Qué métodos con código heredan las subclases?
- ▶ ¿Qué métodos sin código heredan las subclases? ¿Qué tienen que hacer con estos métodos?
- ▶ ¿Qué significa que las clases Circulo y Rectángulo sean *final*?
- ▶ ¿Existe en una versión del constructor sin parámetros en la clase Circulo? ¿Y en la clase Figura2D?
- ▶ Explica la primera línea de código del constructor de la clase Circulo?
- ▶ Explica cómo la clase ImagenCompuesta trabaja con un array de Figura2D si esta clase es abstracta y no podemos crear objetos de ella.
- ▶ Si no existiera el polimorfismo qué propiedades tendría que tener la clase ImagenCompuesta para poder trabajar con las distintas figuras.

Realizamos el
ejercicio 10 del
boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

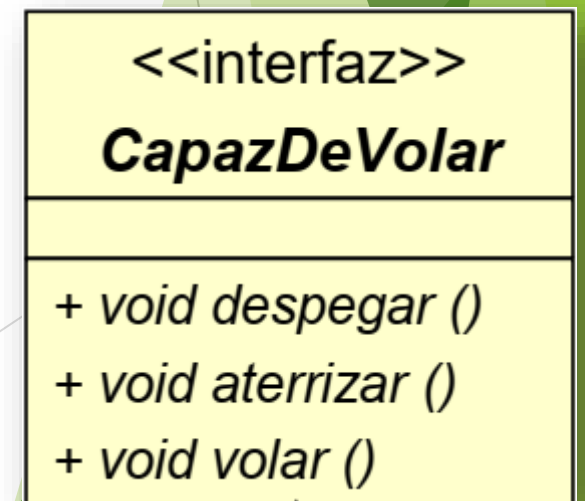
8 - Interfaces

► 8 – Interfaces

- Recordemos que las clases son las plantillas o moldes con las que podemos crear los objetos.
- Pues bien, **las interfaces son las plantillas o moldes con las que podemos crear clases...** Es el molde del molde.
- Cuando queremos crear una clase utilizando una interfaz como plantilla, entonces decimos que la “**clase implementa la interfaz**”.
- Veamos un ejemplo de interfaz:

```
public interface CapazDeVolar {  
    void despegar();  
    void aterrizar();  
    void volar();  
}
```

Todos los métodos son considerados como **public abstract**. Estos modificadores pueden omitirse, o escribirse también, si se desea.



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- Normalmente, las interfaces expresan “capacidades”. Ejemplos: Transportable, Almacenable, PuedeSaltar, CapazDeVolar, Imprimible, Comestible, Vendible...
- Ahora veamos cómo creamos una clase a partir de la interfaz anterior:

```
public class Helicóptero implements CapazDeVolar {
```

```
    // ... Propiedades del helicóptero
```

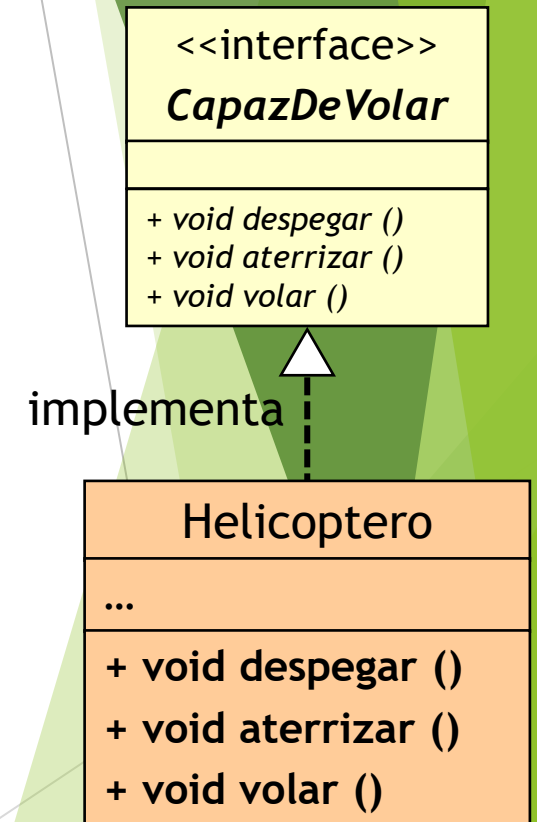
```
    public void despegar() { /* Código aquí */ }
```

```
    public void aterrizar() { /* Código aquí */ }
```

```
    public void volar() { /* Código aquí */ }
```

```
    // ... Otros métodos
```

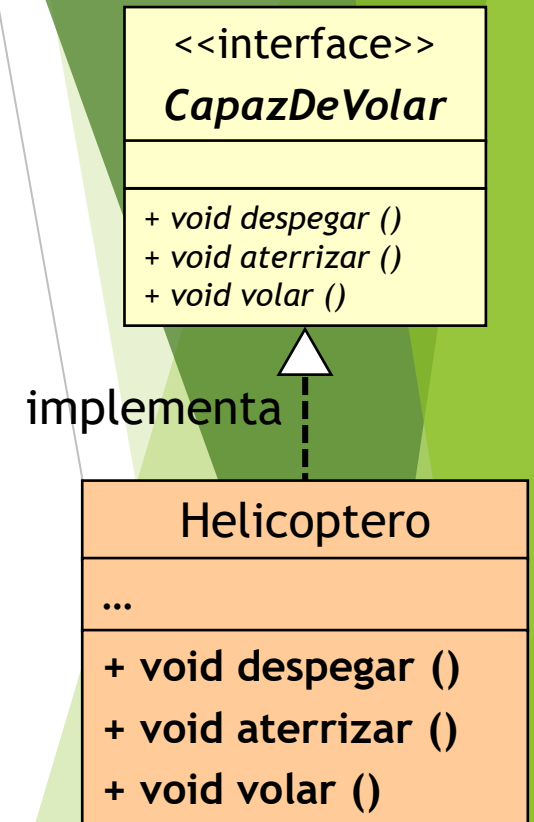
```
}
```



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- ▶ Dado que la clase Helicóptero la hemos creado tomando la interfaz como plantilla, esto **nos obliga a escribir código para cada uno de los métodos declarados en la interfaz**.
- ▶ La interfaz **se comporta como un “contrato”** que deben cumplir las clases que implementen dicha interfaz. El contrato de la interfaz podría decir algo como:
 - ▶ “La clase que implemente la interfaz tiene la OBLIGACIÓN de escribir código para todos los métodos de la interfaz”
 - ▶ “La clase que implemente la interfaz tiene DERECHO a escribir el código como le apetezca y crear tantos métodos adicionales como desee”
- ▶ Las interfaces son una herramienta del lenguaje con mucha capacidad expresiva. Permiten que un arquitecto/diseñador comunique a su equipo de desarrollo las capacidades que deben tener las clases del sistema. Existe una técnica de diseño de software “orientada a interfaces”, que las utiliza como “ladrillo principal”.

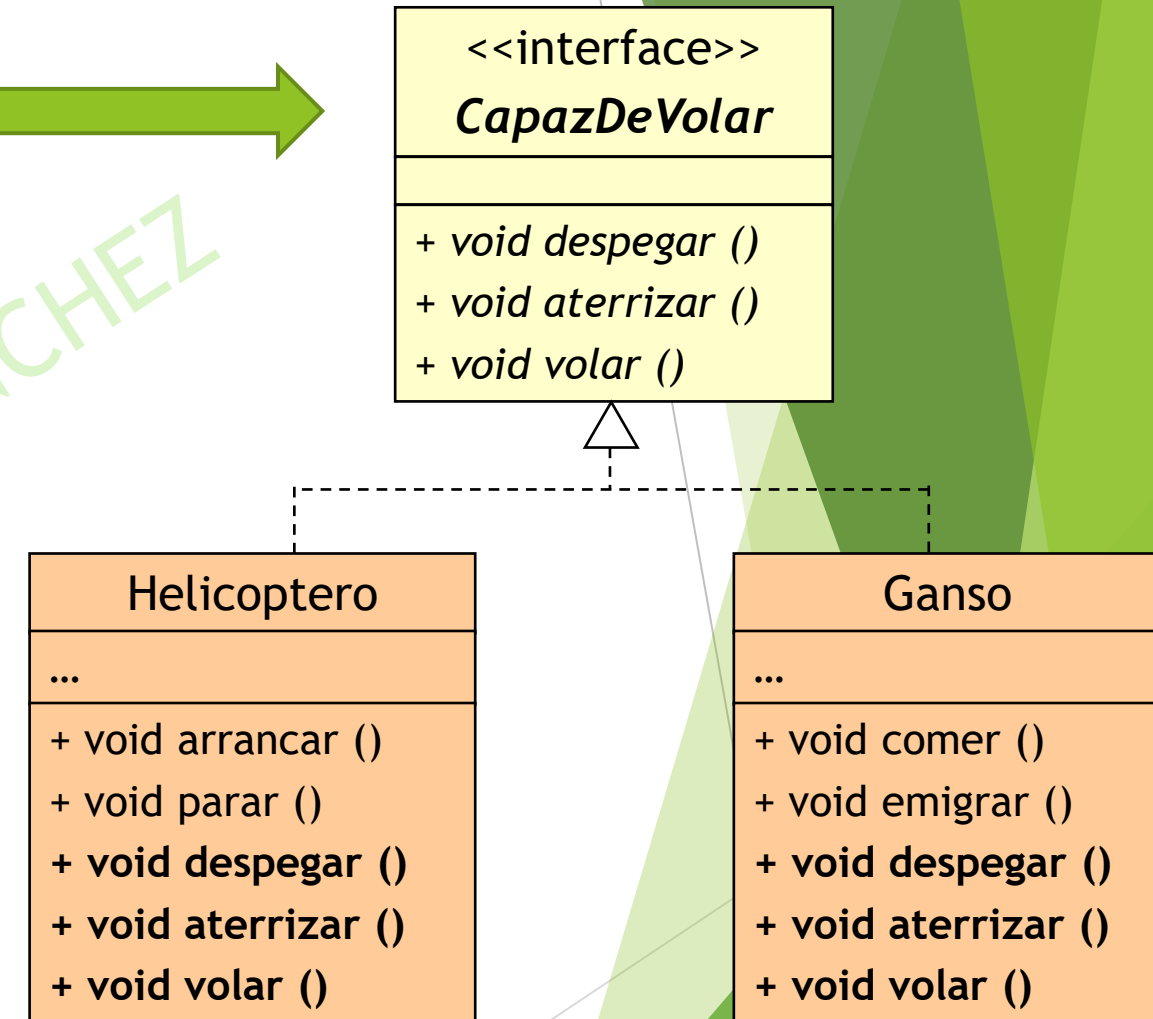
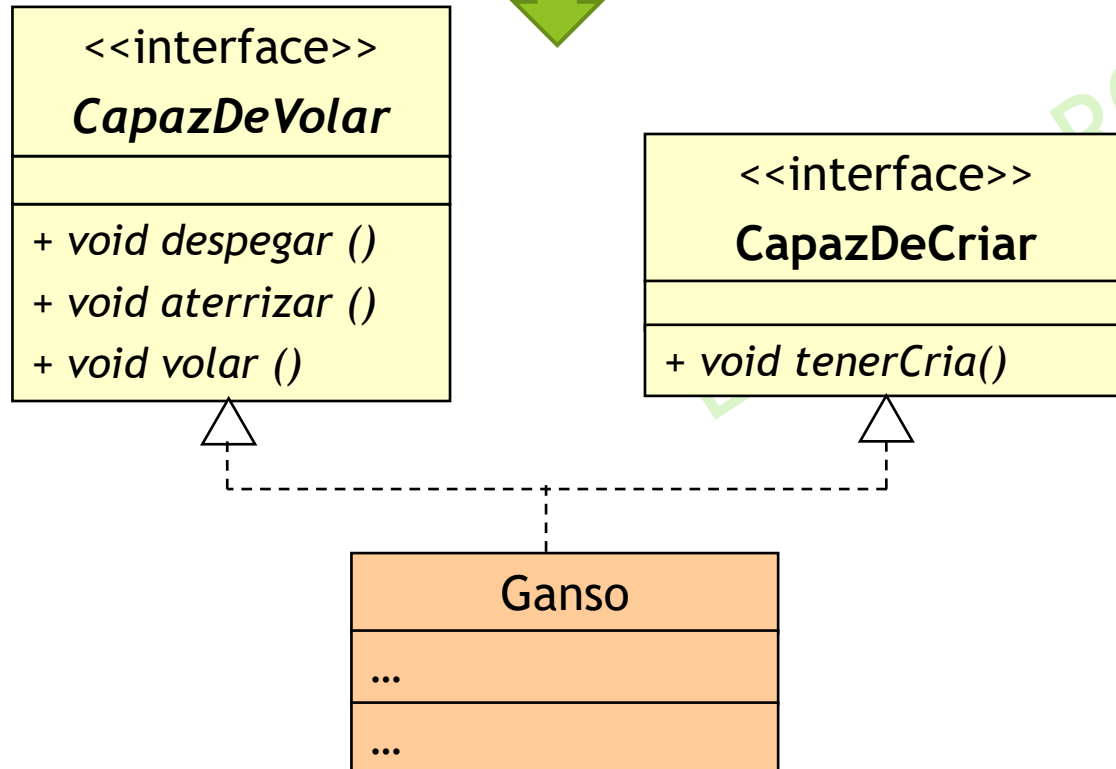


Lee el ejercicio
11 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

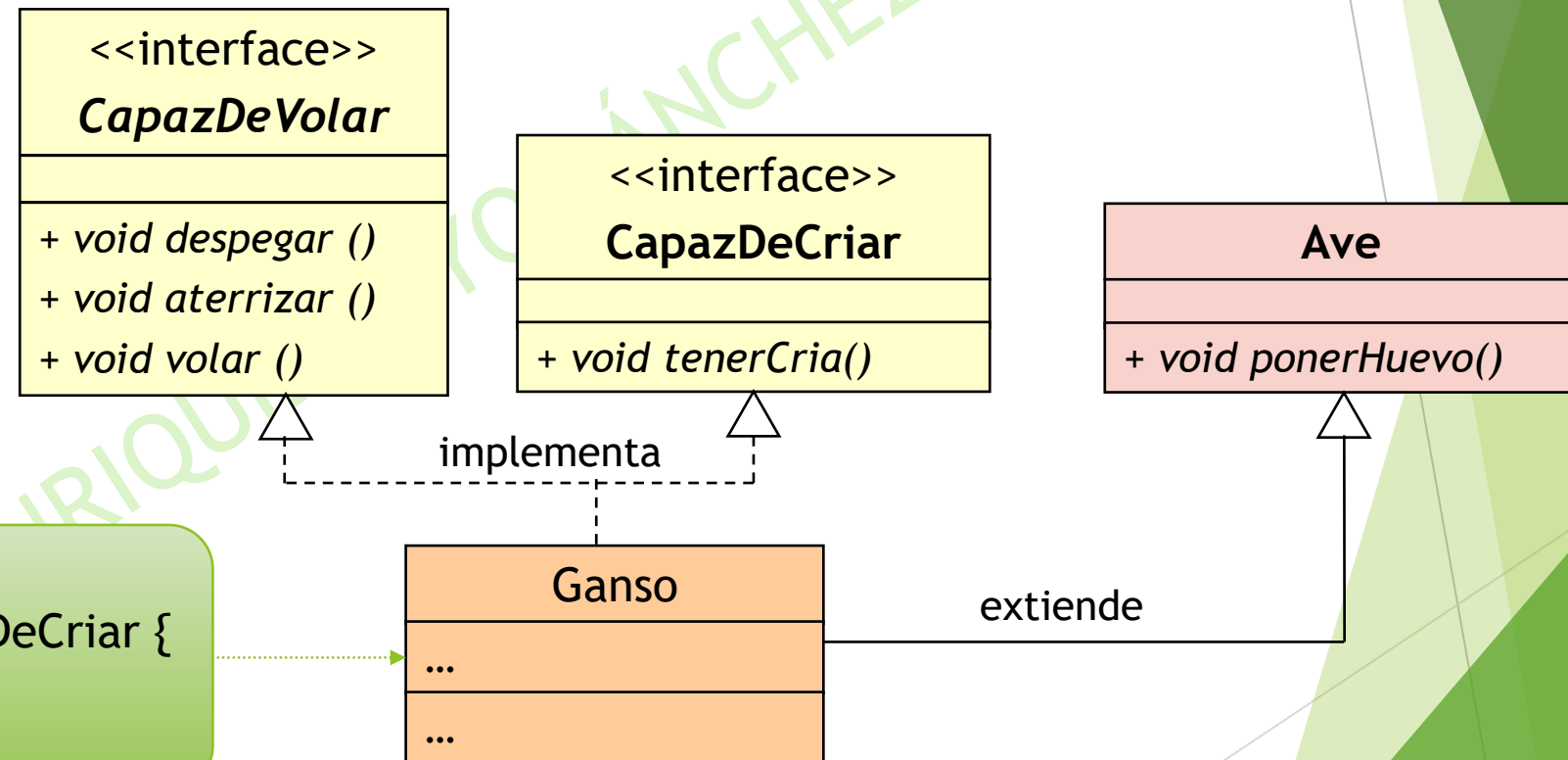
- ▶ Varias clases pueden implementar la misma interfaz.
- ▶ Y también una clase puede implementar más de una interfaz



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- **OJO:** Una clase puede implementar más de una interfaz **pero sólo puede heredar de una superclase.**



```
public class Ganso extends Ave
implements CapazDeVolar, CapazDeCriar {
    ...
}
```

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- Las interfaces normalmente solo llevan métodos (públicos y abstractos) pero también pueden llevar propiedades que serán tratadas como constantes estáticas y públicas. Es decir, todas serán declaradas automáticamente como **public static final**.

- Ejemplo:

```
public interface CapazDeVolar {  
    double GRAVEDAD = 9.8;  
    void despegar();  
    void aterrizar();  
    void volar();  
}
```



UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- Además, el **polimorfismo también funciona con interfaces** permitiendo que objetos de muy distinta índole puedan ser tratados como un grupo. Ejemplo:

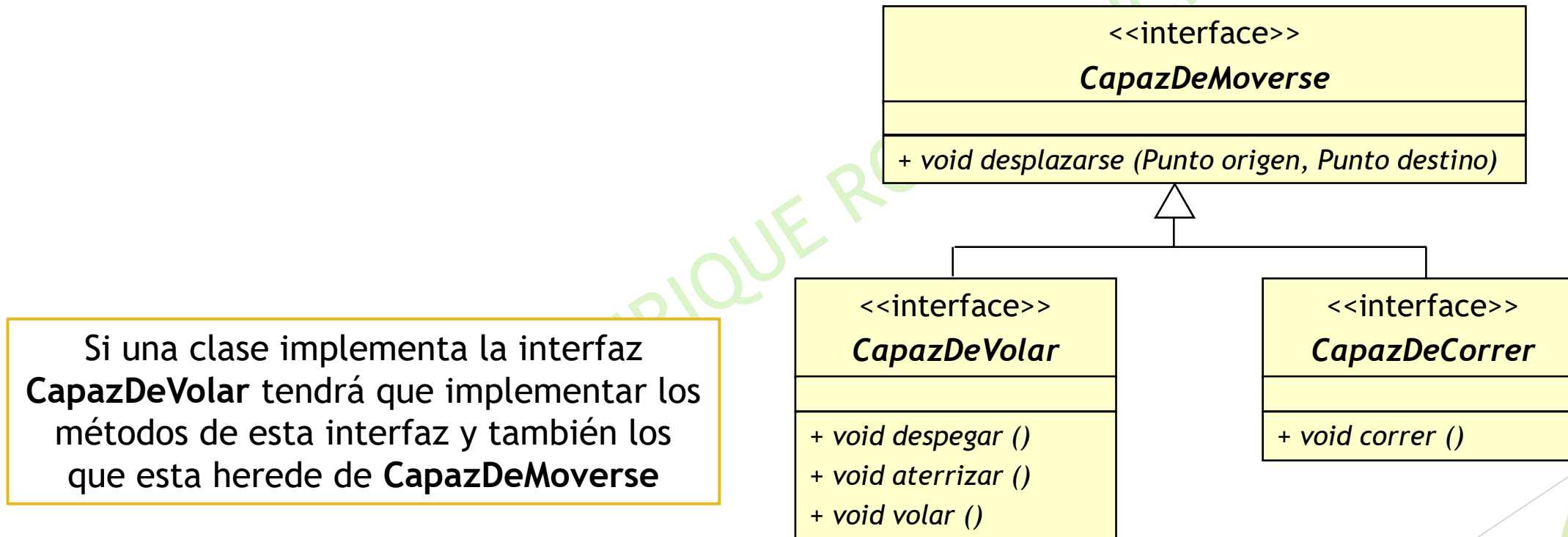
...

```
CapazDeVolar[ ] voladores = new CapazDeVolar [3];  
voladores[0] = new Helicoptero();  
voladores[1] = new Ganso();  
voladores[2] = new Helicoptero();  
for (CapazDeVolar v: voladores) {  
    v.despegar();  
}
```


UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

8 - Interfaces

- Y para rizar el rizo: **también existe la herencia entre interfaces.** Eso sí, sigue las mismas reglas que la herencia entre clases, es decir, una interfaz solo puede heredar de una y solo una “superinterfaz”.



Si una clase implementa la interfaz **CapazDeVolar** tendrá que implementar los métodos de esta interfaz y también los que esta herede de **CapazDeMoverse**

En las siguientes unidades manejaremos algunas interfaces de las librerías del lenguaje y le veremos más sentido a este elemento

Realizamos los ejercicios del 12 al 16 del boletín

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE

Cierre de la unidad

- ▶ En esta unidad hemos visto que en el mundo del software **TODO CAMBIA** y que necesitamos preparar nuestro código para que **“abrace el cambio”** y no luche contra él.
- ▶ Con el encapsulamiento y un buen diseño de los constructores y métodos conseguiremos que nuestras **clases sean más sólidas**.
- ▶ Mediante la composición y la herencia tenemos formas de combinar clases para construir objetos **más potentes y con posibilidad de ser reutilizados**, ampliando sus comportamientos y/o rescribiéndolos.
- ▶ El polimorfismo permite dar un tratamiento uniforme a un conjunto de objetos de tipos distintos pero que tienen algo en común, **esto permite que nuestro código se adapte al cambio con mayor facilidad**.
- ▶ Seguiremos trabajando todas estas técnicas durante el resto del curso, poniéndolas en distintos contextos y profundizando en ellas.

No es la especie más fuerte la que sobrevive, ni la más inteligente, sino la **que responde mejor al cambio**



Charles Darwin

UD 3 - ELEMENTOS AVANZADOS DEL LENGUAJE



The end screen