UD11. POO. Herencia y Polimorfirsmo

Módulo: Programación

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Germán Gascón Grau
g.gascongrau@edu.gva.es

Contenidos

- Herencia
- Polimorfismo
- Clases abstractas
- Clases abstractas vs Interfaces

Pilares básicos de la POO

- **Abstracción**
  - Generalmente los programas “simulan” sistemas o procesos que existen en el mundo real.
  - Identificar las partes relevantes del sistema o proceso que queremos simular.
  - Definir los datos necesarios (atributos) y las acciones (métodos) para llevar a cabo la “simulación”
- **Encapsulamiento**
  - Los datos (valores de los atributos) son la parte más valiosa de los objetos.
  - La mejor forma de mantenerlos a salvo, es que sólo el propio objeto (que es especialista en eso datos) sea capaz de manipularlos.
  - Por tanto, los atributos siempre serán privados de forma que la única forma de cambiar el valor de un atributo sea a través de un método.
- **Herencia**
  - Permite crear nuevas clases a partir de otras existentes.
  - Con ello conseguimos reutilizar código reduciendo la duplicación de código, aumentando la eficiencia y la legibilidad.
- **Polimorfismo**
  - Característica que permite que un objeto de un tipo pueda albergar valores de un subtipo.
  - Dependiendo del tipo que albergue podrá comportarse de diferentes formas.

Herencia

- En los lenguajes de programación orientados a objetos (OO), el mecanismo básico para la **reutilización de código** es la herencia.
- Permite crear nuevas clases que heredan características presentes en clases anteriores:
  - La clase original se llama clase padre, base o superclase.
  - La nueva o nuevas clases se denominan clases hijas, derivadas o subclases.
- En **Java**, sólo se puede implementar la **herencia simple** → se puede heredar de una única clase.

Herencia

- En Java es especialmente relevante la herencia en dos aspectos:
  - La herencia se emplea (intrínsecamente) en el propio lenguaje a partir del conjunto de librerías que tiene.
  - El lenguaje apoya la definición de nuevas clases heredadas de las que ya están definidas.

Herencia

- Para crear una clase hija de otra se hace uso de la palabra **extends** en la clase hija seguida del nombre de la clase padre:

```
class B extends A {...}
```

Herencia. Persona.java

```
public class Persona {
    private final String dni;
    private final String nombre;
    private final String apellidos;

    public Persona(String dni, String nombre, String apellidos) {
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    public String getApellidos() {
        return apellidos;
    }

    public String getDni() {
        return dni;
    }

    public String getNombre() {
        return nombre;
    }

    @Override
    public String toString() {
        return "Persona{" +
                "dni='" + dni + '\'' +
                ", nombre='" + nombre + '\'' +
                ", apellidos='" + apellidos + '\'' +
                '}';
    }
}
```

Herencia. Alumno.java

```
public class Alumno extends Persona {
    private final String nia;
    private final String grupo;

    public Alumno(String dni, String nombre, String apellidos, String nia, String grupo) {
        // Para crear un alumno, primero creamos un objeto persona
        super(dni, nombre, apellidos);
        this.nia = nia;
        this.grupo = grupo;
    }

    public String getGrupo() {
        return grupo;
    }

    public String getNia() {
        return nia;
    }

    @Override
    public String toString() {
        return "Alumno{" +
                "dni='" + getDni() + '\'' +
                ", nombre='" + getNombre() + '\'' +
                ", apellidos='" + getApellidos() + '\'' +
                ", grupo='" + grupo + '\'' +
                ", nia='" + nia + '\'' +
                '}';
    }
}
```

Funcionamiento de la herencia

- Una subclase hereda **TODOS** los atributos y métodos de la superclase, aunque no todos los miembros tienen porqué ser accesibles.
- Serán accesibles todos los métodos y propiedades con modificadores de acceso **public**, **protected** y “por defecto” (sin especificar).

Tipo acceso	Modificador	Acceso desde subclase mismo paquete	Acceso desde subclase distinto paquete
Privado	**private**	NO	NO
Sin especificar		SI	NO
Protegido	**protected**	SI	SI
Público	**public**	SI	SI

Funcionamiento de la herencia

- En las clases derivadas pueden añadirse atributos y métodos adicionales.
- Una subclase **NO** hereda los constructores:
  - Cada nueva clase (incluso las derivadas), debe definir sus propios constructores.
  - Si no se implementa ningún constructor, el compilador de Java añade uno predeterminado sin argumentos.
  - Orden de ejecución de los constructores: desde el nivel más alto de la jerarquía de herencia, hasta el más específico.

Funcionamiento de la herencia

- La clase `Alumno` hereda los atributos dni, nombre y apellidos y los métodos getter y toString() de la clase `Persona`.
- Respecto al acceso a los atributos dni, nombre y apellidos, no podrán ser accedidos directamente por la clase `Alumno`, ya que han sido declarados `private` en otro archivo java.
- Si quisiéramos que la clase `Alumno` pudiese acceder directamente a los atributos de Persona, podríamos declararlos como `protected`.
- Por supuesto, puede acceder indirectamente utilizando los getters implementados en la clase `Persona`.

Funcionamiento de la herencia

- Podemos definir un atributo en la subclase con el mismo nombre que en la superclase:
  - El atributo de la superclase queda "oculta", es decir, existirán 2 atributos con el mismo nombre.
- Podemos definir un método en la subclase con el mismo nombre y la misma cabecera que en la superclase:
  - El método es reemplazado por el nuevo → **redefinimos** el método.
  - El modificador de acceso debe ser el mismo o menos restrictivo que el de la superclase.
- En los dos casos anteriores (atributos o métodos), cuando referenciamos al atributo o método con su nombre (o this.nombre), estamos refiriéndonos al atributo o método de la subclase.
- Para referirnos a atributos o métodos de una superclase que hayan sido ocultados o sobreescritos, podemos hacerlo con **super**.nombre.

Funcionamiento de la herencia

- Para impedir que se pueda redefinir un atributo o un método se le antepondrá el modificador **final**.
- Al modificador **final** aplicado a una clase hace que no se puedan definir clases derivadas.

super

- La palabra reservada **super** permite llamar a una propiedad o método de la superclase:
  - **this** → hace referencia a la instancia de la clase actual
  - **super** → hace referencia a la superclase respecto a la clase actual
- **super** es imprescindible para poder acceder a métodos redefinidos o anulados por herencia.

Constructores

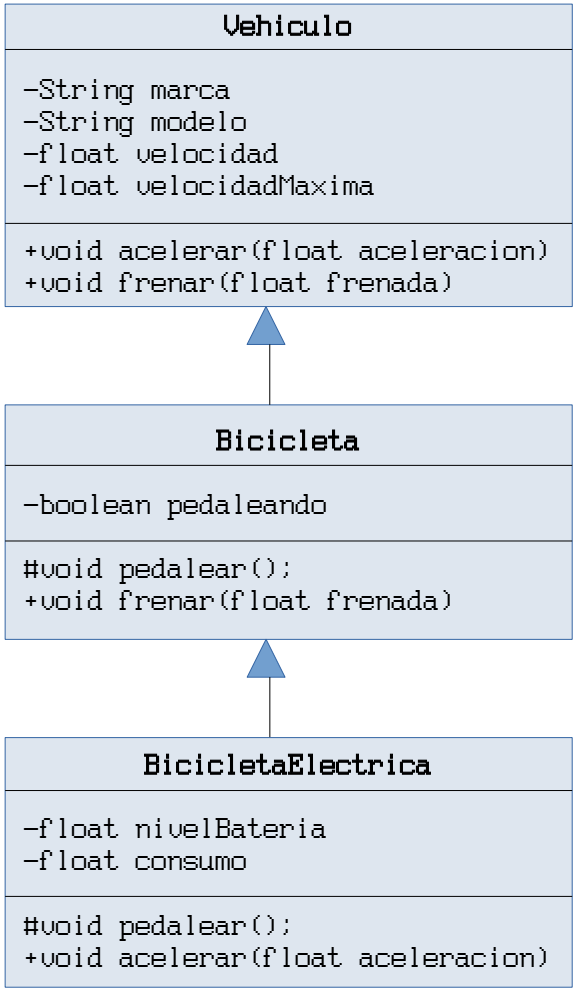
- Los constructores tienen la posibilidad de **invocar a otro constructor de su propia clase** con la sentencia **this(...)**.
- Los constructores de las subclases tienen la posibilidad de **invocar a los constructores** de las **superclases** con la sentencia **super(...)**.
- Las llamadas **super(...)** o **this(...)**, si se utilizan, **deben ser obligatoriamente la primera sentencia del constructor**.

Constructores

Por defecto Java realiza estas acciones:

- **Si la primera instrucción de un constructor** de una subclase es una sentencia que **no es ni super ni this** ...
  - **Añade de forma invisible e implícita una llamada super()** al constructor por defecto de la superclase
  - Esto puede dar errores si en la superclase hemos definido algún constructor y no hemos definido el constructor sin parámetros (pérdida del constructor por defecto). El compilador no encuentra el constructor.
  - Si en la superclase no hemos definido ningún constructor, no habrá problemas.
- Si se emplea **super(...)** en la primera instrucción ...
  - Se llama al constructor seleccionado de la superclase
- Si la primera instrucción es **this(...)**
  - Se llama al constructor seleccionado según lo indique this y luego continúa con las sentencias del constructor.

Veamos otro ejemplo de Herencia



** Constructores y getters no incluidos para simplificar el diagrama de clases*

Herencia. Vehiculo.java

```
public class Vehiculo {
    private final String marca;
    private final String modelo;
    private float velocidad;
    private final float velocidadMaxima;

    public Vehiculo(String marca, String modelo, float velocidadMaxima) {
        this.marca = marca;
        this.modelo = modelo;
        this.velocidad = 0; // Inicialmente el Vehículo está parado
        this.velocidadMaxima = velocidadMaxima;
    }

    public String getMarca() {
        return marca;
    }

    public String getModelo() {
        return modelo;
    }

    public float getVelocidad() {
        return velocidad;
    }

    public float getVelocidadMaxima() {
        return velocidadMaxima;
    }

    public void acelerar(float aceleracion) {
        velocidad += aceleracion;
        if (velocidad > velocidadMaxima) {
            velocidad = velocidadMaxima;
        }
    }

    public void frenar(float frenada) {
        velocidad -= frenada;
        if (velocidad < 0) {
            velocidad = 0;
        }
    }
}
```

Herencia. Bicicleta.java

```
public class Bicicleta extends Vehiculo {
    private boolean pedaleando;

    public Bicicleta(String marca, String modelo, float velocidadMaxima) {
        // Primero construmimos un Vehículo
        // super() invoca al constructor de la clase Padre
        super(marca, modelo, velocidadMaxima);
        // Posteriormente establecemos los atributos propios de Bicicleta
        this.pedaleando = false;
    }

    public boolean isPedaleando() {
        return pedaleando;
    }

    protected void pedalear() {
        pedaleando = true;
        super.acelerar(1);
    }

    @Override
    public void frenar(float frenada) {
        super.frenar(frenada);
        pedaleando = false;
    }
}
```

Herencia. BicicletaElectrica.java

```java
public class BicicletaElectrica extends Bicicleta {
    private float nivelBateria;
    private final float consumo;

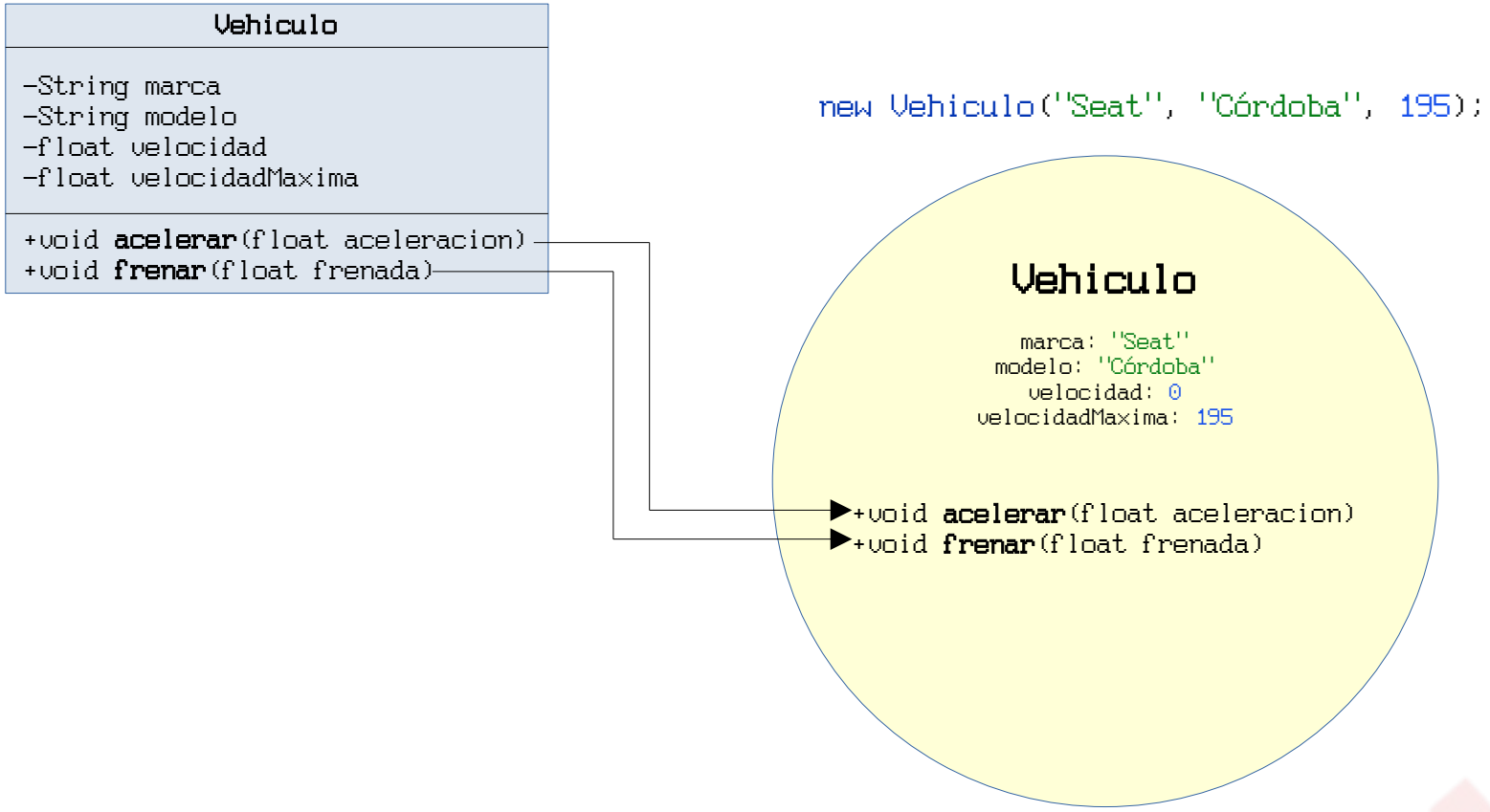
    public BicicletaElectrica(String marca, String modelo, int velocidadMaxima, float nivelBateria, float consumo) {
        // Primero construimos una Bicicleta
        super(marca, modelo, velocidadMaxima);
        // Luego establecemos los atributos específicos de BicicletaElectrica
        this.nivelBateria = nivelBateria;
        this.consumo = consumo;
    }

    public float getNivelBateria() {
        return nivelBateria;
    }

    // Sobreescribimos acelerar
    @Override
    public void acelerar(float aceleracion) {
        if (nivelBateria > 0) {
            super.acelerar(aceleracion);
            nivelBateria -= consumo;
        }
    }

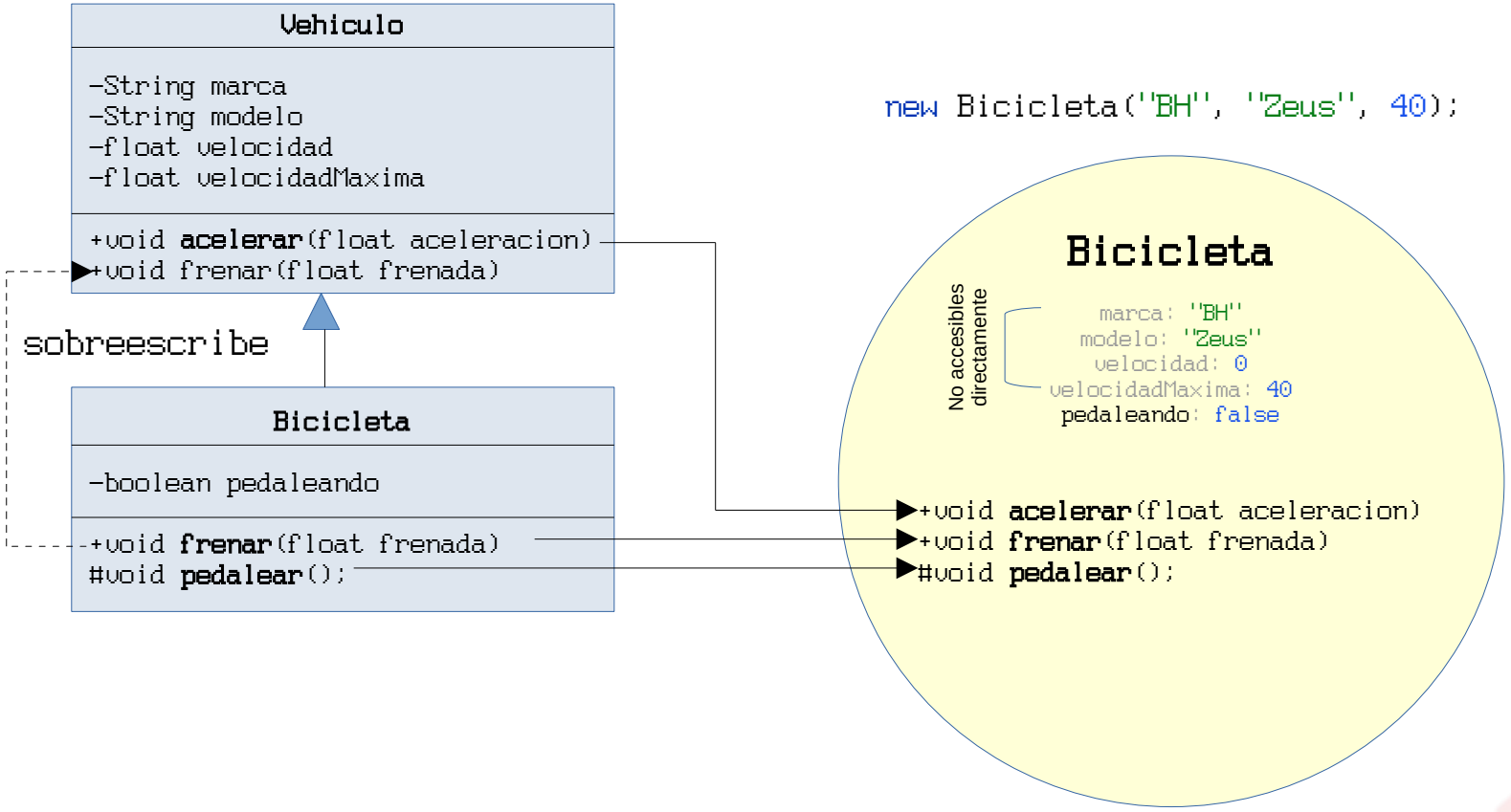
    @Override
    protected void pedalear() {
        super.pedalear();
        // La batería se carga un poco
        nivelBateria += consumo;
    }
}
```

Veamos otro ejemplo de Herencia



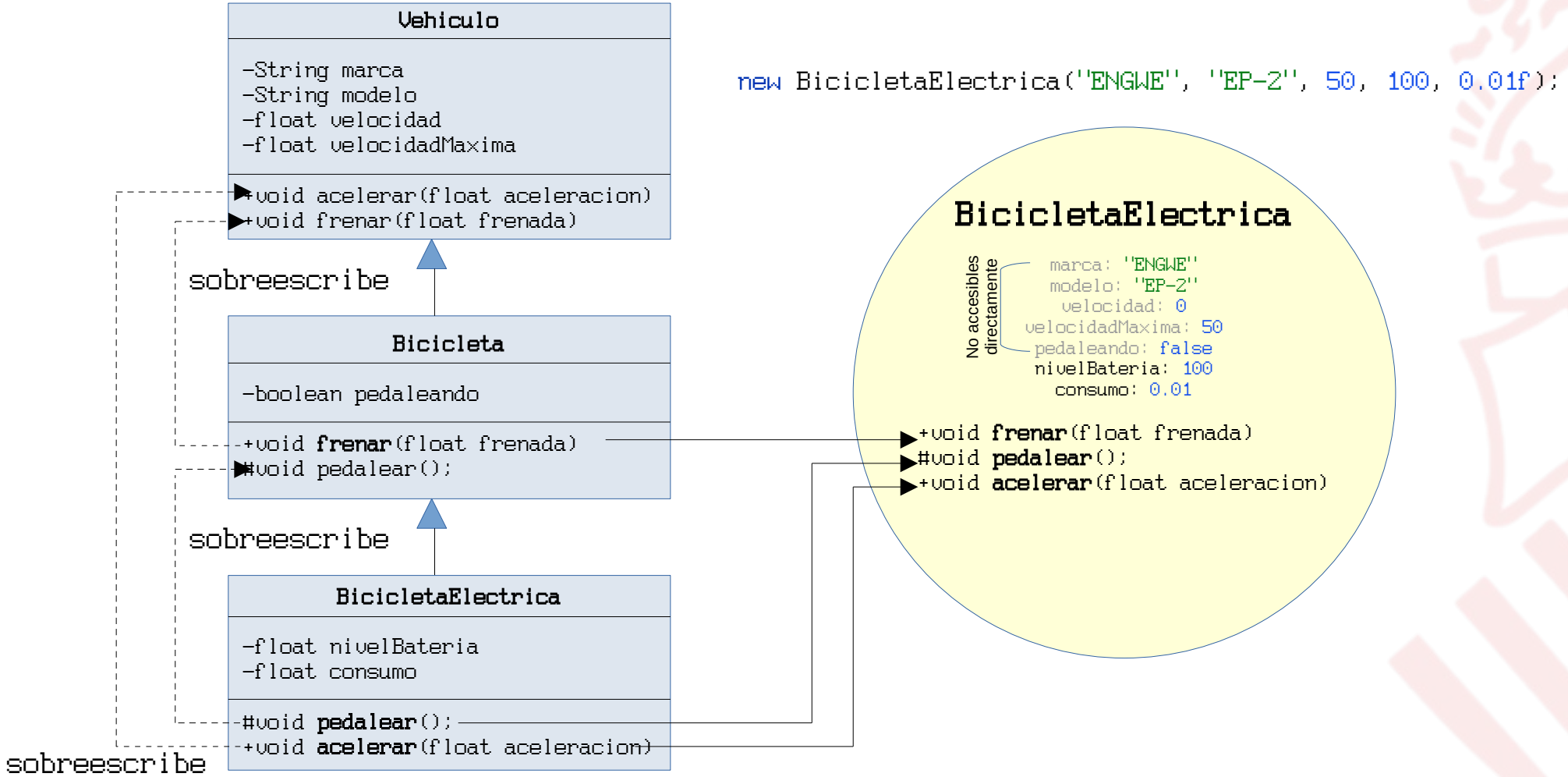
** Constructores y getters no incluidos para simplificar el diagrama de clases*

Veamos otro ejemplo de Herencia



** Constructores y getters no incluidos para simplificar el diagrama de clases*

Veamos otro ejemplo de Herencia



** Constructores y getters no incluidos para simplificar el diagrama de clases*

Ejercicio práctico 1

```
public class Medico {
    private boolean trabajaEnHospital;

    public void atiendePaciente() {
        // Código para hacer un chequeo
    }
}


public class MedicoFamilia extends Medico {
    private boolean llamaAPacientes;

    public void darConsejo() {
        // Código para dar un consejo
    }
}


public class Cirujano extends Medico {
    public void atiendePaciente() {
        // Código para realizar una cirujía
    }

    public void hacerIncision() {
        // Código para realizar una incisión
    }
}
```

Ejercicio práctico 1

A partir del código anterior contesta a las siguientes preguntas:

- ¿Cuántos métodos tiene la clase Medico?
- ¿Cuántos métodos tiene la clase Cirujano?
- ¿Cuántos métodos tiene la clase MedicoFamilia?
- ¿Puede un MedicoFamilia atender a un paciente? ¿Qué método se ejecutará?
- ¿Puede un MedicoFamilia realizar una incisión? ¿Por qué?

Ejercicio práctico 2

Vamos a plantear el **diseño**, no el código, de un programa **simulador de animales**.

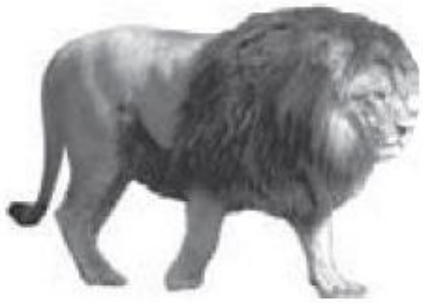
Objetivo: cómo hacer el proceso de abstracción de la información.

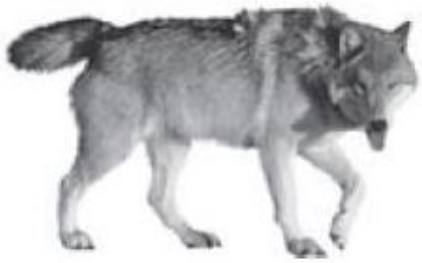
Tenemos **6 tipos de animales** que tendrán características comunes que tendremos que identificar y agrupar.

Utilizaremos **herencia** para evitar duplicar código en las subclases.

Ejercicio práctico 2

Los 6 tipos de animales que vamos a considerar son:






Ejercicio práctico 2

1. Identificar aspectos comunes

- ¿Qué tienen en común estos 6 tipos?

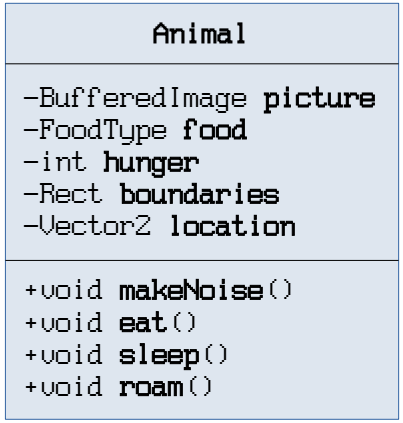
  Todos los objetos son animales, por eso crearemos una superclase que sea común llamada Animal

  En la clase Animal pondremos los atributos y los métodos que todos los animales necesiten.

Ejercicio práctico 2

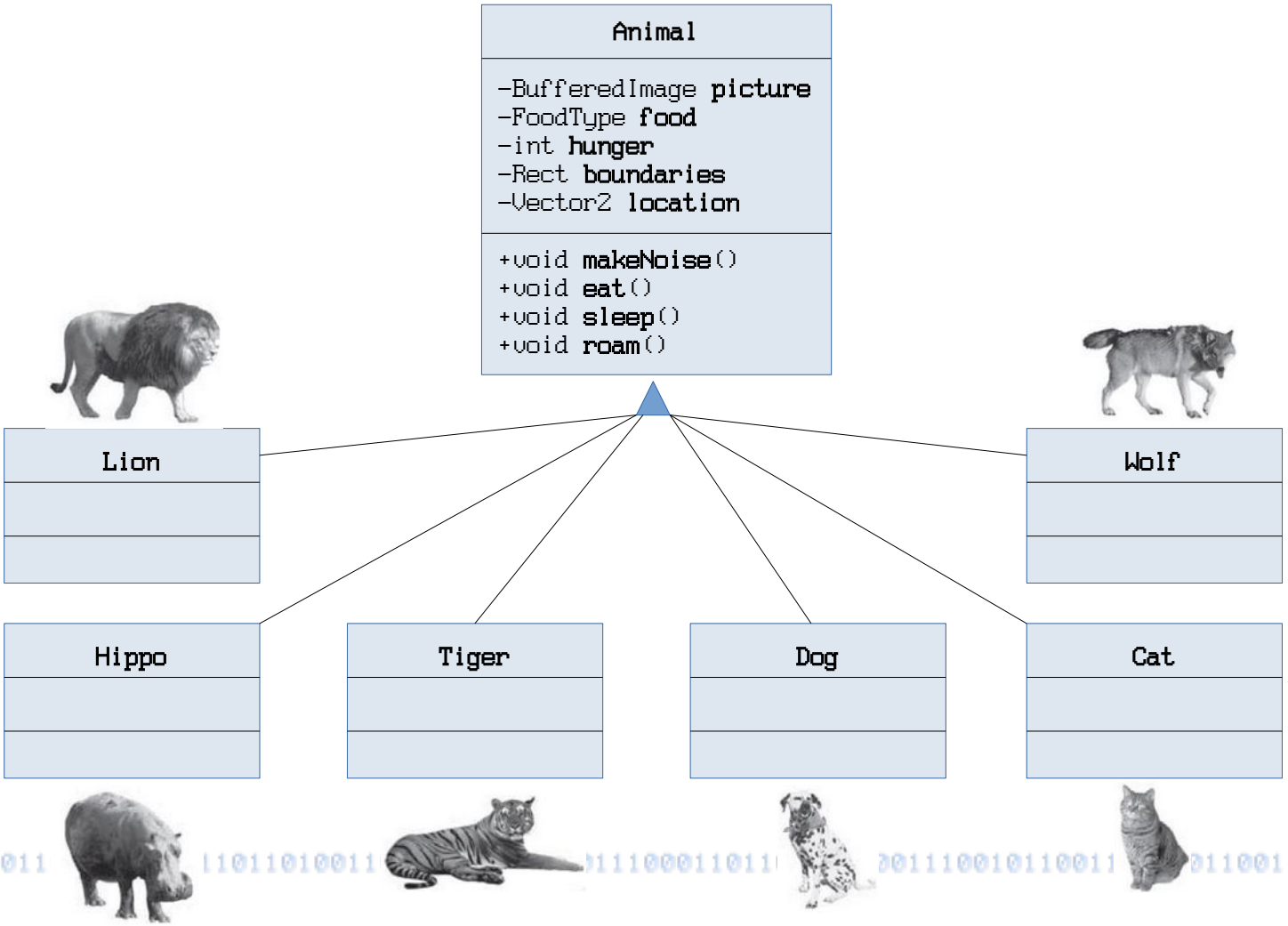
2. Diseñar una clase que represente el estado y el comportamiento común de todos los animales.

- Atributos
  - **picture**: foto del animal
  - **food**: carnívoro o hervíboro
  - **hunger**: nivel de hambre (entero)
  - **boundaries**: medidas (alto x ancho)
  - **location**: coordenadas x e y
- Métodos
  - **makeNoise()**: ruido del animal
  - **eat()**: comportamiento al comer
  - **sleep()**: comportamiento al dormir
  - **roam()**: comportamiento cuando no come ni duerme



Ejercicio práctico 2

3. Comprobar si las subclases necesitan modificar su comportamiento por tener algo específico o particular.



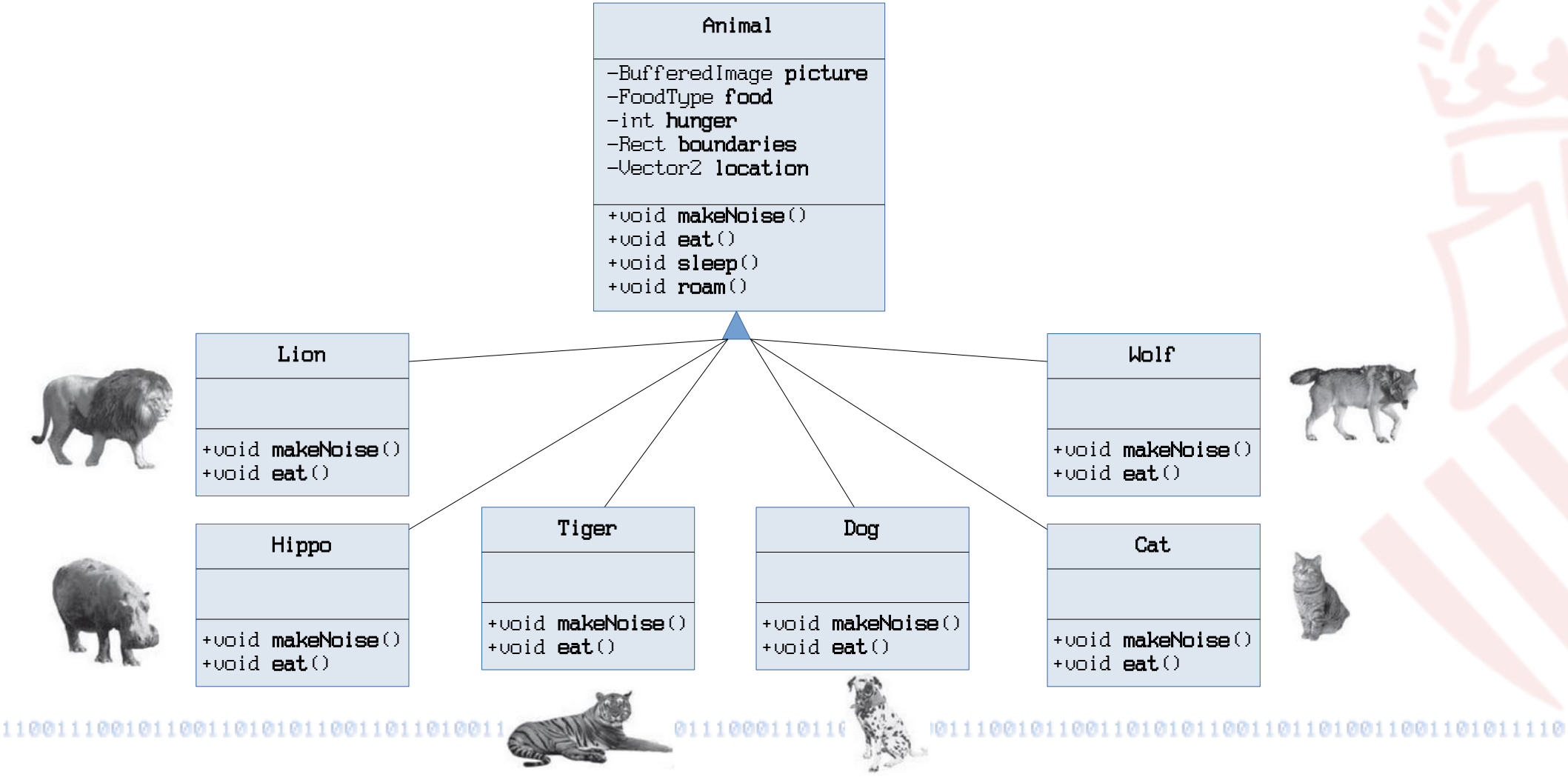
Ejercicio práctico 2

3. Comprobar si las subclases necesitan modificar su comportamiento por tener algo específico o particular.

En un primer refinamiento, al observar la clase Animal y sus posibles subclases, nos damos cuenta que **eat**() y **makeNoise**() serán sobreescritos por los subclases.

Ejercicio práctico 2

3. Comprobar si las subclases necesitan modificar su comportamiento por tener algo específico o particular.



Ejercicio práctico 2

4. Buscar más abstracciones y comportamientos comunes

La clase `Wolf` y `Dog` tienen comportamientos comunes, y las clases `Lion`, `Tiger` y `Cat` también.

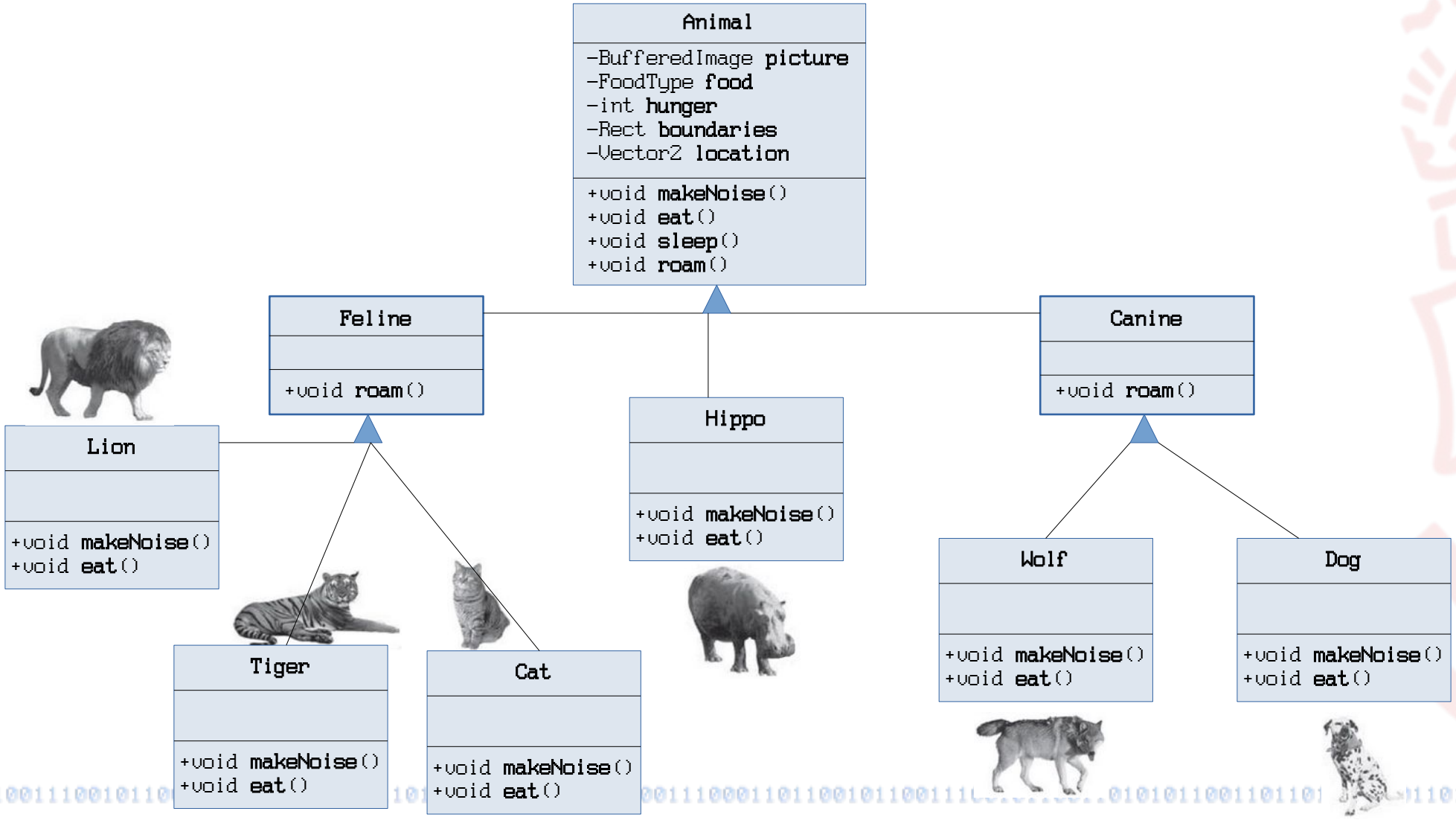
Podemos dividir la jerarquía en "familias biológicas" organizando los animales en:

- **Felinos**
- **Caninos**

Los caninos tendrán una implementación del método roam() y los felinos tendrán otra, mientras que el hipopótamo utilizará el roam() de Animal.

Ejercicio práctico 2

4. Buscar más abstracciones y comportamientos comunes



Ejercicio práctico 2

Analicemos la clase `Wolf`

- ¿Cuántos métodos tiene?
- ¿Dónde está definido cada método?
- Identificar la clase del método que se ejecuta en las siguientes instrucciones:

```
Wolf w = new Wolf();
w.makeNoise();
w.roam();
w.eat();
w.sleep();
```

Test SER o NO SER

- Es un test que deberíamos pasar siempre a la hora de trabajar con herencia.
- La subclase **extends** la superclase.
- La subclase **ES** una superclase.
  - Un cirujano ***ES*** un médico.
  - Un tigre **ES** un felino y un felino ES un animal.
- Un motor **NO ES** un Coche, pero un Coche **TIENE** un motor.
- La subclase podrá hacer cualquier cosa que haga la superclase (o incluso más cosas).

¿Cuándo hacer uso de la herencia?

- Cuando quieras modelar un **comportamiento más específico** que aquel definido en la superclase.
- Cuando el **comportamiento de varias clases sea igual y general**.
- NO UTILICES herencia cuando NO se cumpla la regla SER, aunque puedas reutilizar código.

Ventajas de la herencia

- Evita duplicar código:
  - Si ponemos el código en la superclase, lo heredan las subclases y cualquier cambio que afecte al comportamiento general se hará únicamente en la superclase y afectará a todas las subclases.
- Facilita y simplifica el mantenimiento del código.
- Código más sencillo y flexible.
- Es más fácil extender que desarrollar desde el principio.

Resumen

- Una subclase **EXTIENDE** a una superclase.
- Una subclase **hereda todos los miembros** de una superclase, pero dependiendo de su visibilidad, serán accesibles o no.
- Los **métodos** heredados **pueden ser sobreescritos** en la subclase.
- Los **atributos no se sobreescriben**; si se vuelven a definir, no serán los mismos.
- Emplearemos el **test SER** para **verificar** la jerarquía de herencia.
- La relación SER trabaja en una única dirección (el hijo hereda del padre pero no a la inversa).

Resumen

- Cuando un método es sobrescrito en una subclase y el método es invocado por una instancia (un objeto) de la subclase, la versión del método utilizada es la versión sobrescrita.
- La versión **más específica** del método es la que se aplica.
- Si se cumple ...
  - B extends A (un `Feline` extends `Animal`)
  - B es A (un `Feline` es un `Animal`)
  - Si C es B → C es A (un `Cat` es un `Animal`)

Polimorfismo

- El **polimorfismo** es una característica de la POO que permite que objetos de diferentes clases puedan ser tratados como objetos de una **clase común**.
- Este concepto facilita que una sola interfaz pueda ser utilizada para representar distintas formas de comportamiento, dependiendo de la clase específica del objeto que la implemente.

Polimorfismo

- Siguiendo con nuestro ejemplo, una forma de polimorfismo sería:

```
Animal myDog = new Dog();
```

- `myDog` es una referencia de tipo `Animal` pero que apunta a un objeto de tipo `Dog`.
- Esto es posible debido a la relación SER. Como todo `Dog` ES un `Animal`, puede ser apuntado por un objeto de tipo `Animal`.

Polimorfismo. Upcasting

- A una variable de una clase A podemos, además de asignarle objetos de la clase A, asignarle también cualquier objeto de una clase que herede de A.

```
class Persona {...}
class Alumno extends Persona {...}

Alumno a = new Alumno();
Persona p = a;      // casting implícito
Persona p = (Persona) a;// explícito: no es necesario
```

- Esta operación siempre se puede hacer, sin necesidad de indicarle explícitamente al compilador.

Polimorfismo. Downcasting

- Es el caso en el que una variable de una subclase hace referencia a un objeto de la superclase.

```
Persona p = new Alumno();
Alumno a = (Alumno) p; // explícito; es necesario
```

- Esta operación sólo se puede hacer si el objeto referenciado por "p" es realmente de tipo Alumno.
- En caso contrario se provoca un error lanzándose la excepción de tipo **ClassCastException**.

Polimorfismo. Ejemplo

```
public class Vet {
    public void giveShot(Animal a) {
        // Se invocará al makeNoise() del Animal que se haya
pasado como parámetro
        a.makeNoise();
    }
}

Dog tobby = new Dog();
Cat catty = new Cat();
Vet vet = new Vet();
vet.giveShot(tobby); // Guau, Guau
vet.giveShot(catty); // Miaaaaauuu
```

- Cuando el veterinario le ponga la vacuna al **Animal**, dependiendo de qué **Animal** se haya pasado como parámetro emitirá un sonido u otro, ya que se llamará al método concreto. En este ejemplo, se llamará al `makeNoise()` de **Dog** y de **Cat**.

Polimorfismo. Ejemplo con Arrays

```
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Lion();
animals[3] = new Hippo();
animals[4] = new Tiger();

for (int i = 0; i < animals.length; i++) {
    animals[i].makeNoise();
}
```

Lo mejor de polimorfismo es que podemos recorrer el array e ir llamando al método makeNoise() y se ejecutará la versión concreta para cada Animal.

- Cuando i valga 0 será un **Dog** y se ejecutará el método `makeNoise()` de **Dog**, cuando i valga 1 será un **Cat** y emitirá el sonido de un gato, y así sucesivamente.

Clases abstractas

- Tiene sentido crear un `Tiger` o un `Wolf`.
- Pero, ¿tiene sentido crear un `Animal`? ¿Qué forma tiene? ¿Cómo es?
- Necesitamos la clase `Animal` para heredar de ella, pero realmente no se necesita crear un `Animal`, pero sí un `Wolf`, un `Tiger` o un `Dog`.
- Marcando una clase como abstracta (**abstract**), evitamos que pueda ser instanciada, es decir que se creen objetos de esa clase.

Clases abstractas

- Al diseñar, tendremos que decidir si una clase será abstracta o concreta:
  - Clase **concreta**: se crearán objetos de la clase.
  - Clase **abstracta**: no se podrán crear objetos.
- Para crear una clase abstracta, lo haremos poniendo **abstract** antes de la definición de la clase:

```
abstract public class NombreClase {


}
```

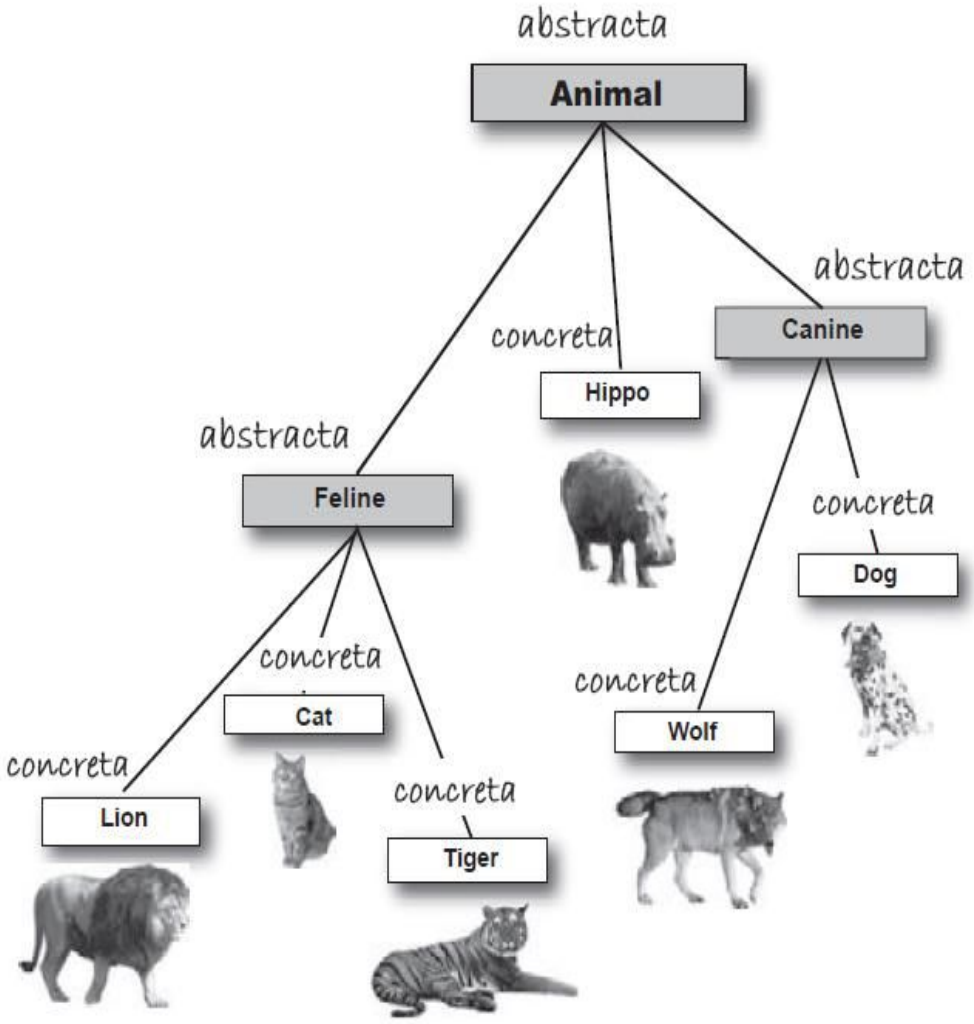
- El compilador garantiza que no se podrán crear objetos de esa clase.

Clases abstractas

- Las clases abstractas son muy útiles, ya que ayudan a definir la **interfaz común** en toda la jerarquía de herencia.
- Una clase abstracta puede tener **métodos concretos** y **métodos abstractos**.
  - Los **métodos concretos** son métodos que tienen implementación.
  - Los **métodos abstractos** son métodos que no tienen implementación.
- Cuando una clase concreta extienda a una clase abstracta deberá implementar todos sus métodos abstractos o de lo contrario provocará un error de compilación.
- Esto permite que las clases abstractas declaren qué métodos deben existir aunque haya algunos que aún no se sepa cómo implementarlos.

Clases abstractas

- En nuestro ejemplo del simulador de Animales, las clases `Animal`, `Feline` y `Canine` serían claras candidatas a ser clases abstractas.



Métodos abstractos

- Clase abstracta: Debe ser extendida (extends).
  - Puede contener métodos abstract y no abstract.
- Método abstracto: Debe ser sobrescrito.
  - Un método abstracto no tiene implementación (código).
  - Son métodos genéricos implementados (sobreescritos) en las subclases.
  - Los métodos abstractos deben estar en clases abstractas.

```
public abstract void eat();
```

Métodos abstractos

- Existen para ser heredados (polimorfismo) estableciendo así una interfaz común en toda la jerarquía.
- Se deben **IMPLEMENTAR = sobrescribir** en las subclases. Se debe conservar:
  - El tipo de retorno.
  - El número y tipo de argumentos.
- La primera clase concreta en el árbol debe implementar todos los métodos abstractos.
- Por ejemplo, `Dog` debe sobrescribir los métodos abstractos de `Canine` y de `Animal`.

La clase Object

- Es la superclase de todas las clases.
- Todas las clases heredan de la clase `Object`.
- Por ello, los ArrayList pueden contener cualquier objeto de cualquier clase.
- Cualquier clase que no herede explícitamente de otra clase, hereda implícitamente de `Object`.
- En el ejemplo:
  - `Dog` no hereda de `Object` porque hereda de `Canine`.
  - `Canine` no hereda de `Object` porque hereda de `Animal`.
  - Pero `Animal` sí hereda de `Object`.

La clase Object

- La clase `Object` define un comportamiento común a todos los objetos.
- Los métodos más importantes de la clase `Object` son:
  - `boolean equals(Object o)`
  - `int hashCode()`
  - `Class<?> getClass()`
  - `String toString()`
  - `Object clone() throws CloneNotSupportedException`
- Cualquier clase que creemos hereda los métodos de la clase `Object`.

Método equals

- ```
  boolean equals(Object o) {
      // Instrucciones para comparar
      // Debe devolver un valor booleano
  }
  ```
- La implementación de este método debe devolver cuando un objeto se considera igual a otro.
  ```

Requisitos del método equals

- Según la JSL (Java Language Specification), el método **equals** debería cumplir las siguientes características:
  - La **comparación con null** debe devolver siempre **falso**:

    ```
    x.equals(null) == false
    ```

  - **Reflexivo**:

    ```
    x.equals(x) == true
    ```

  - **Simétrico**:

    ```
    Si x.equals(y) == true entonces y.equals(x) == true
    ```

  - **Transitivo**:

    ```
    Si x.equals(y) == true && y.equals(z) == true entonces
    x.equals(z) == true
    ```

  - **Consistente**: múltiples invocaciones deben devolver siempre el mismo resultado.

Método hashCode

- `int hashCode()`
- Este método tal y como hemos visto al estudiar las Colecciones, debe devolver un **identificador único** para cada objeto.

Requisitos del método hashCode

- Según la JSL (Java Language Specification), el método hashCode debería cumplir las siguientes características:
- **Consistente con equals**:
  - `si x.equals(y) == true entonces`

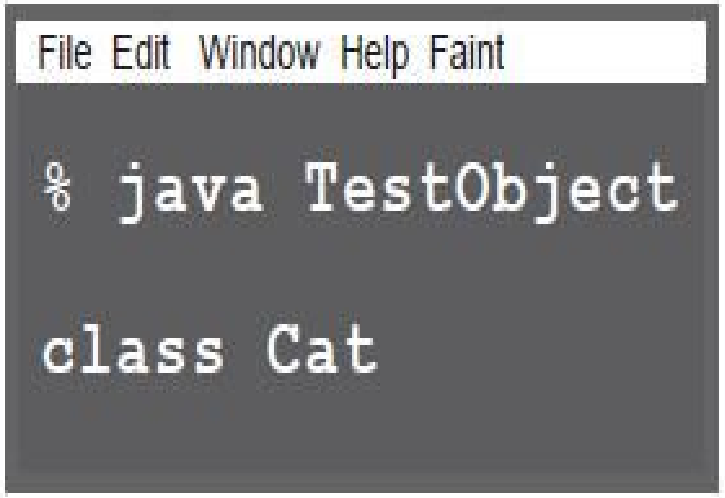
    `x.hashCode() == y.hashCode()`

    por tanto, siempre que sobreescribamos el método equals debemos sobreescribir también el método hashCode.
- **Consistente a lo largo de la ejecución**: es decir el hashCode generado a lo largo de la ejecución del programa debe ser el mismo. Posteriores ejecuciones del programa pueden dar resultados diferentes.

Método getClass

- `Class<?> getClass()`
- Éste método devuelve el nombre de la clase del objeto instanciado.

```
Cat c = new Cat();
System.out.println(c.getClass());
```



Nos devuelve el nombre de la clase del objeto instanciado

Método toString

- `String toString()`
- Este método devuelve una representación como texto del objeto.

Método clone

- `Object clone() throws CloneNotSupportedException`
- Si queremos que nuestras clases soporten clonación debemos implementar la interfaz **`Cloneable`**.
- Según la JSL (Java Language Specification), el método clone debería cumplir las siguientes características:
  - `x.clone() != x`

    es decir, no deben ser el mismo objeto (dirección de memoria)
  - `x.clone().getClass() == x.getClass()`

    original y clon deberían tener la misma clase
  - `x.clone().equals(x) == true`

    original y clon deberían considerarse iguales

Consideraciones del método clone

- Respecto al método `clone()` hay diversidad de opiniones, pero la opción más aceptada para implementar un mecanismo de copia de objetos es utilizar un **constructor copia**.
- Para crear un **constructor copia** simplemente debemos crear un constructor que reciba como parámetro un objeto de la misma clase que se está implementado.
- El constructor copia puede tener más parámetros, pero según la JSL, el objeto de la misma clase debe ser el primero.

```
public class Alumno {
    private final String nombre;
    // ...

    public Alumno(Alumno a) {
        this.nombre = a.nombre;
        // ...
    }
}
```

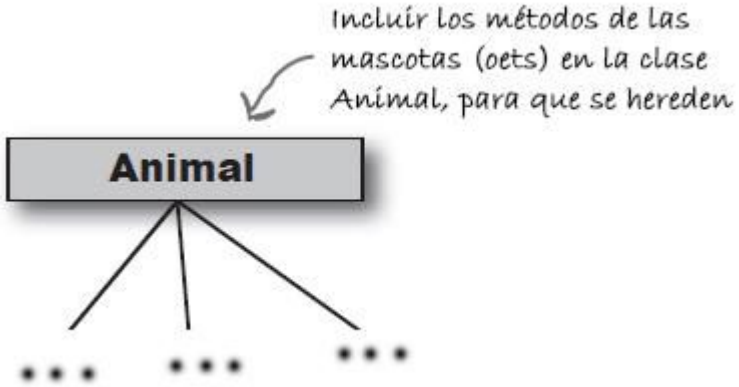
Interfaces

Comportamientos comunes

- Imaginemos que alguien quiere emplear las clases de animales que hemos diseñado para un programa de una tienda de mascotas (PetShop).
- Hay algunos comportamientos que son comunes a todas las mascotas, y por tanto servirían para los perros domésticos, ya que son mascotas, pero no para los perros salvajes.
- ¿Qué opciones tenemos para describir este comportamiento?

Opción 1: Comportamiento en la clase Animal

- Definimos los métodos de las mascotas en la clase `Animal`.



- Ventajas:
  - Todos los animales heredarán las características de las mascotas.
- Inconvenientes:
  - ¿Un `Lion` o un `Wolf` pueden ser una mascota? ¿Queremos permitirlo?

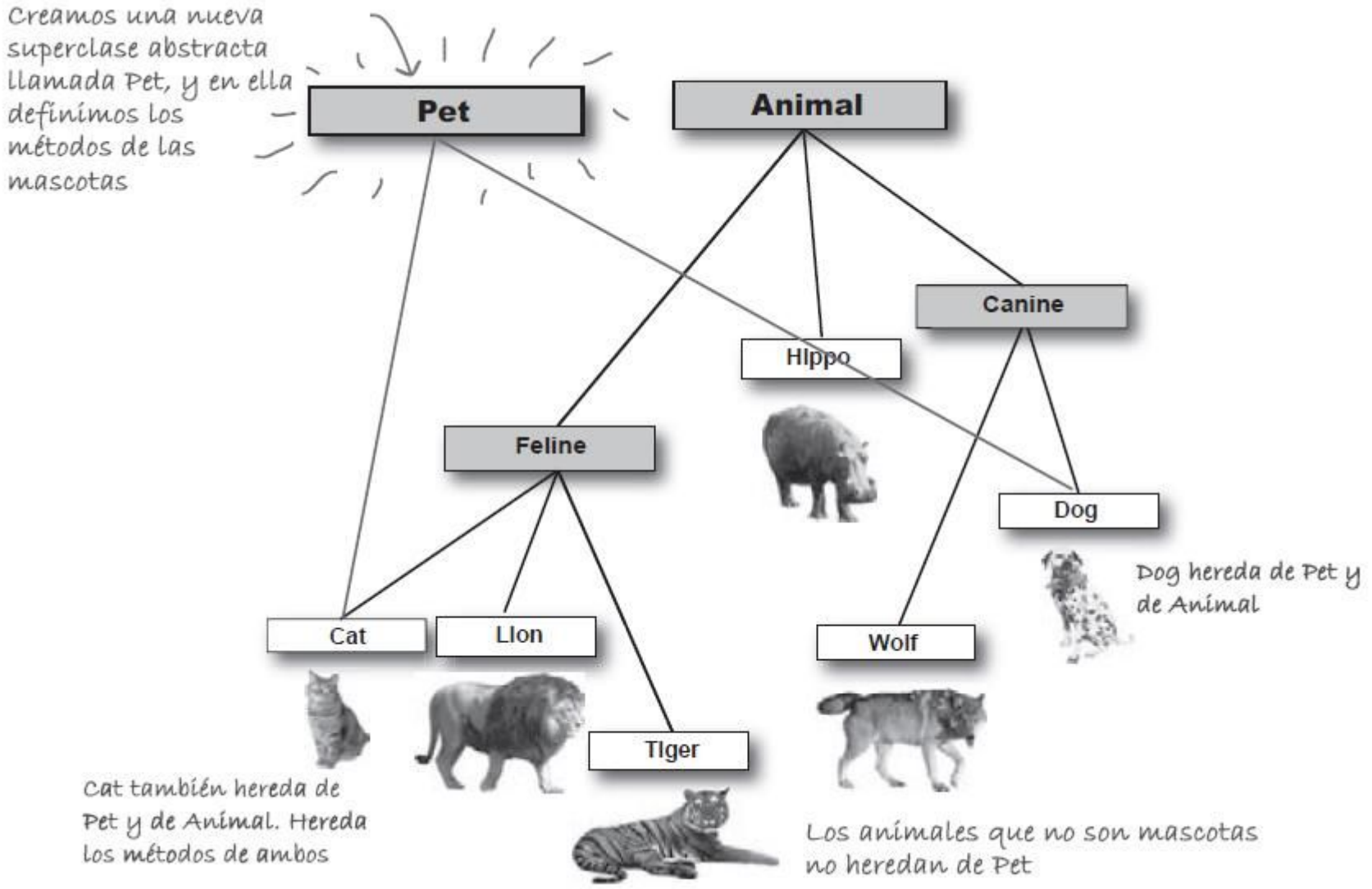
Opción 2: Métodos abstract en la clase Animal

- Definimos los métodos en la clase `Animal` como `abstract`, sin código, y obligamos a que las subclases los sobrescriban.
- Ventajas:
  - Las clases que modelan animales que no sean mascotas están obligadas a sobrescribir los métodos de la superclase (no hará nada).
- Inconvenientes:
  - Supone una pérdida de tiempo sobrescribir todos los métodos en todas las clases concretas.

Opción 3: Métodos en algunas clases

- Definimos los métodos de las mascotas en aquellas clases en las que tienen sentido: `Dog` y `Cat`.
- Ventajas:
  - No tendremos animales que no sean mascotas con un comportamiento de mascotas.
- Inconvenientes:
  - No podríamos emplear polimorfismo para las mascotas.

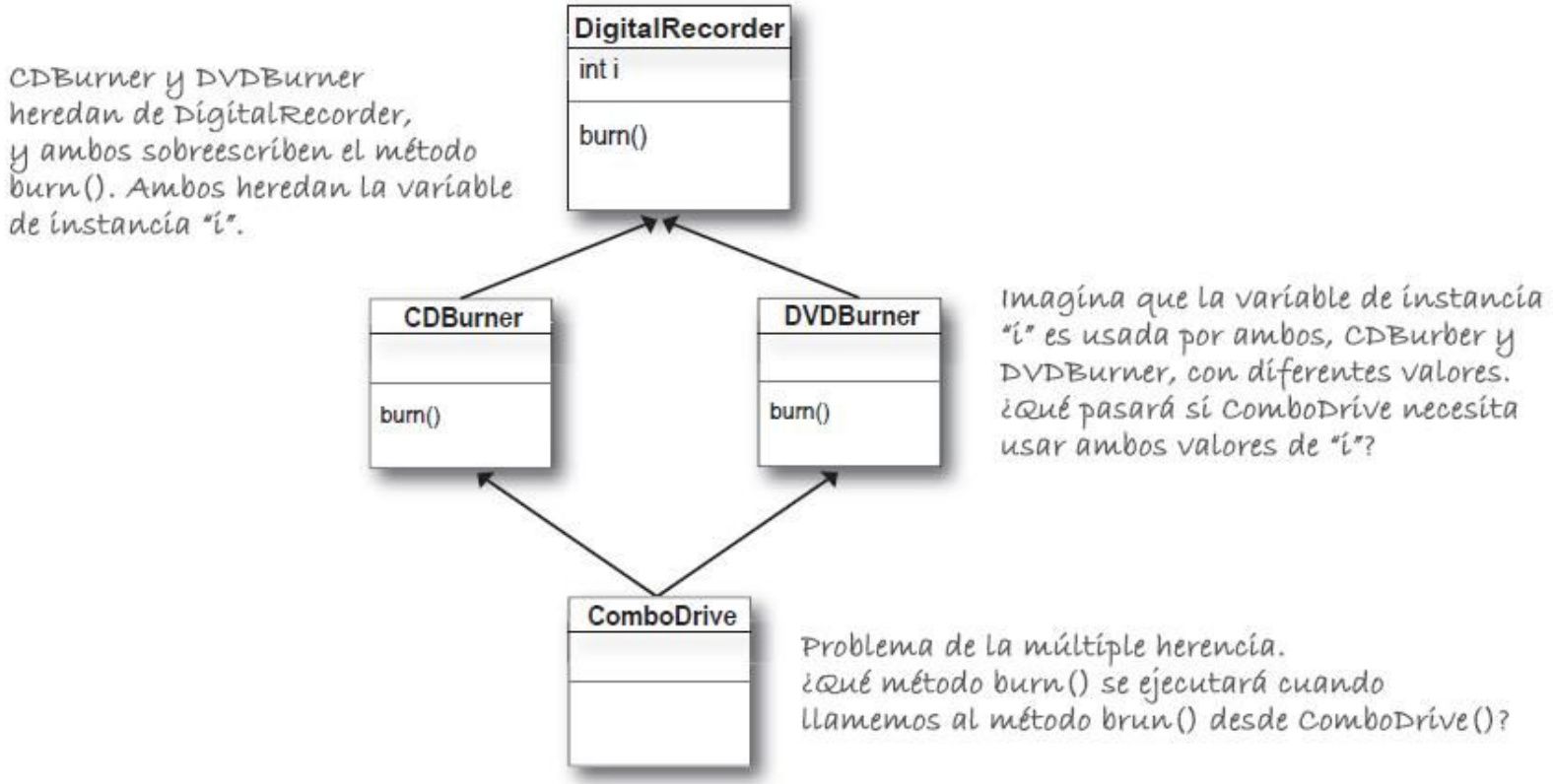
Lo que necesitaríamos sería ...



Pero hay un problema ...

- Java no permite la herencia múltiple
- La herencia múltiple provoca un problema:

Deadly Diamond of Death (Problema del Diamante de la muerte)



Solución: las interfaces

- Una `interface` (palabra reservada) es una clase 100% abstracta.
- Todos los métodos de una `interface` son **abstractos** y deben ser sobrescritos en las clases concretas.

Definir e implementar

Para DEFINIR una interface:

```
public interface Pet {...}
```

Usa la palabra reservada "interface" en vez de "class"

Para IMPLEMENTAR una interface:

```
public class Dog extends Canine implements Pet {...}
```

Usa la palabra reservada "implements" seguida del nombre de la interface. Fíjate que, además de implementar la interfaz Pet, estás heredando de la clase Canine.

Definir e implementar

Los métodos de la interface se definen como public y abstract, aunque no es necesario escribirlo, ya lo son.

Escribimos "interface" en lugar de "class"

```
public interface Pet {

    public abstract void beFriendly();

    public abstract void play();

}
```

Todos los métodos de la interface son abstractos, deben ir con punto y coma (;) Recuerda que no tienen cuerpo con el código.

Dog ES un Animal
y Dog es un Pet

Escribimos "implements" seguido del nombre de la interface

```
public class Dog extends Canine implements Pet {

    public void beFriendly() {...}

    public void play() {..}


    public void roam() {...}

    public void eat() {...}

}
```

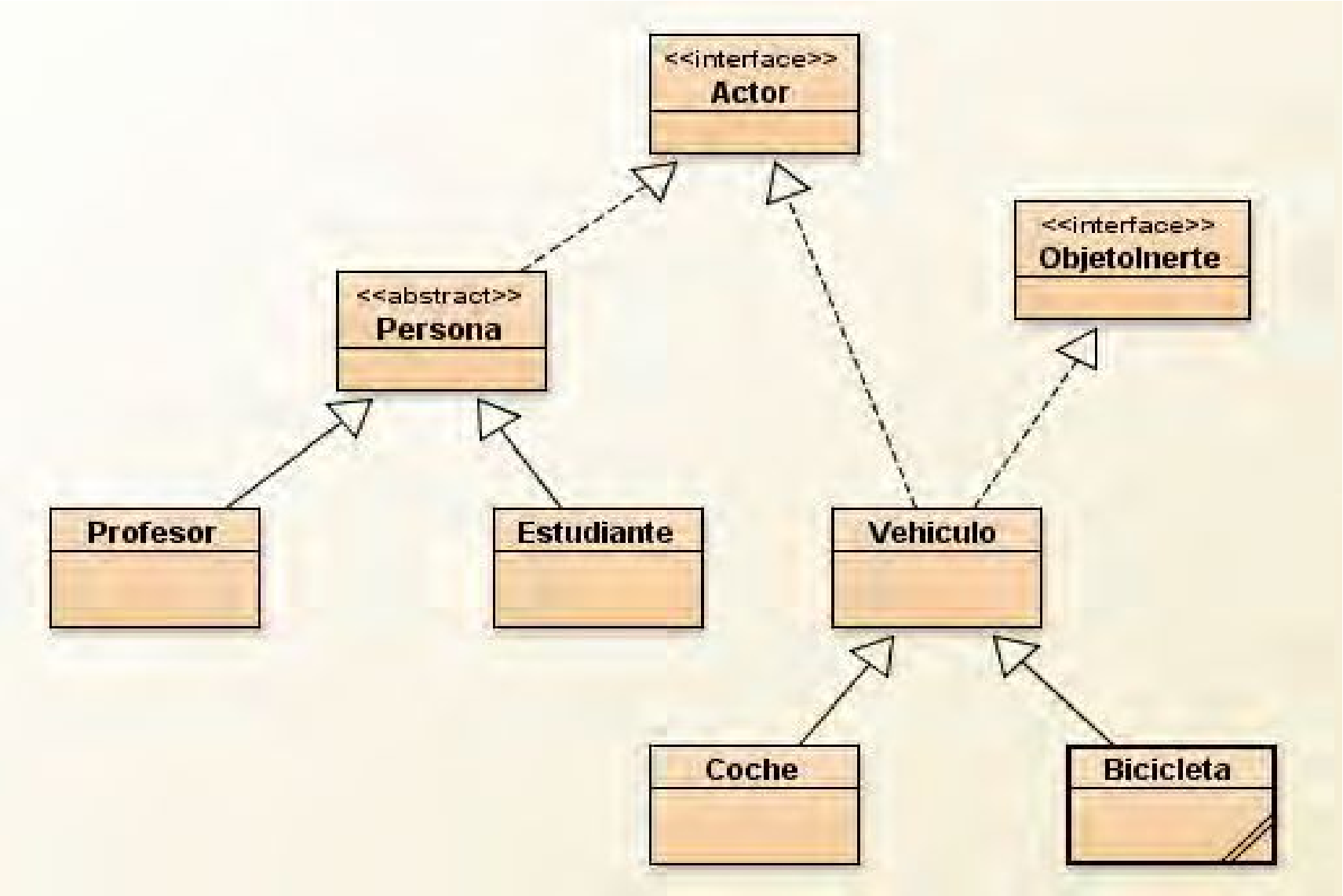
Aquí debemos implementar los métodos. Fíjate en los corchetes, no va con punto y coma.

Estos métodos son los métodos normales sobreescritos heredados de las superclases

Características de las interfaces

- Emplea la palabra reservada `interface`.
- Todos los métodos son **public** y **abstract.**
- No tienen constructores.
- Sólo pueden tener atributos de tipo "public static final", es decir, atributos de clase, públicos y constantes.
- Las clases implementan las interfaces (implements) en lugar de heredar o extender otras clases (extends).
- Las clases pueden implementar varias interfaces, pero sólo pueden heredar de una clase.
- Las interfaces pueden heredar entre ellas.

Ejemplo


<<interface>>
Actor
<<interface>>
ObjetoInerte
<<abstract>>
Persona
Profesor
Estudiante
Vehiculo
Coche
Bicicleta

Características de las interfaces

```
public interface Actor {...}

public abstract class Persona implements Actor {...}

public class Profesor extends Persona {...}

public class Estudiante extends Persona {...}

public interface ObjetoInerte {...}

public class Vehiculo implements Actor, ObjetoInerte {...}

public class Coche extends Vehiculo {...}

public class Bicicleta extends Vehiculo {...}
```

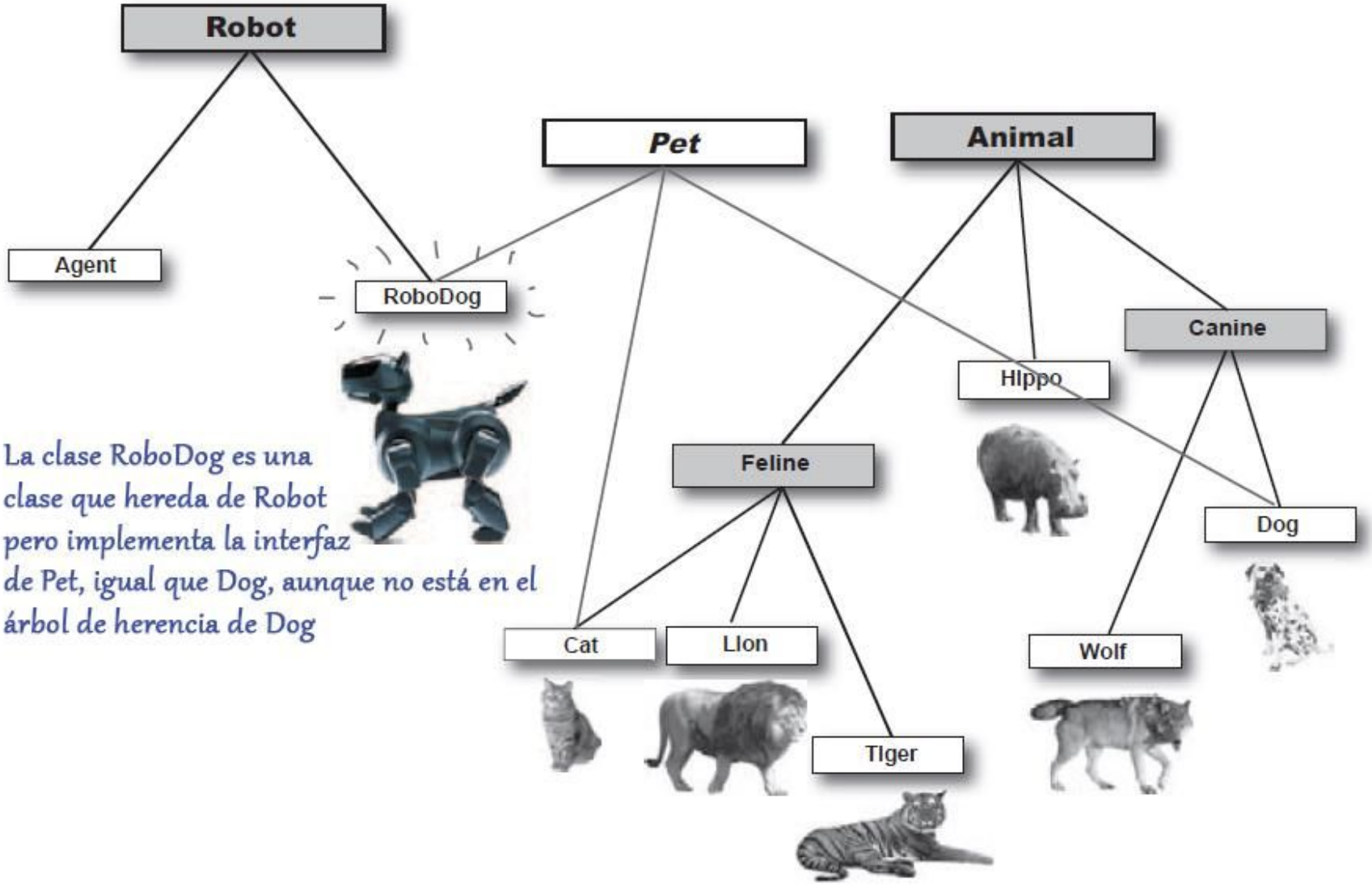
Herencia en las interfaces

```
interface E1 {
   void metodo1();
   void metodo2();
}
interface E2 {
   void metodo3();
}

interface E3 extends E1, E2 { // Herencia múltiple entre interfaces
   void metodo4();
   // metodo1, metodo2 y metodo3 se heredan de E1 e E2
}

class C implements E3 {
   // La clase C deberá implementar metodo1, metodo2, metodo3 y metodo4
}
```

Implementación en diferentes árboles de herencia



Herencia e interfaces

- **Herencia**
  - Las subclases están en el mismo árbol de herencia.
  - Una clase únicamente puede heredar de otra clase.
- **Interfaces**
  - No es necesario el mismo árbol de herencia.
  - Una clase puede implementar varias interfaces.
  - Las interfaces definen los roles de una clase.

Interfaces vs Clases abstractas

- Una interfaz sólo puede tener métodos abstractos o por defecto (desde Java 8). Una clase abstracta puede tener métodos abstractos y métodos no abstractos.
- Los atributos / variables declaradas a una interfaz deben ser static final, mientras que las de una clase abstracta pueden ser también dinámicas y no finales.
- Una interfaz no puede tener constructor, mientras que una clase abstracta si.
- Una interfaz puede extender sólo de otras interfaces (herencia en interfaces) mientras que una clase abstracta puede extender a otras clases Java y puede implementar múltiples interfaces.
- Todos los miembros de una interfaz son públicos, mientras que los miembros de una clase abstracta pueden tener varios niveles de visibilidad (private, protected, etc.).