

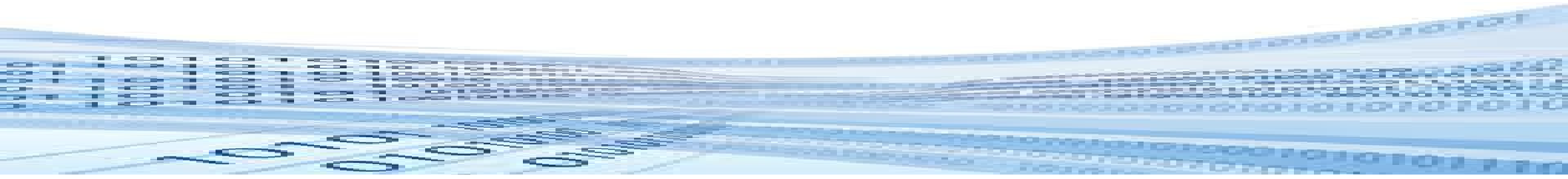
UD11.- Herencia y polimorfismo

Módulo: Programación
1º DAM



CONTENIDOS

- Herencia
- Polimorfismo
- Clases abstractas
- Interfaces



Propiedades de la POO

- **Encapsulamiento**

- Una clase está formada por propiedades o atributos (variables de instancia) y funciones y procedimientos (métodos).
- No se pueden definir variables ni métodos fuera de una clase, es decir, no hay variables ni métodos globales.

- **Ocultación**

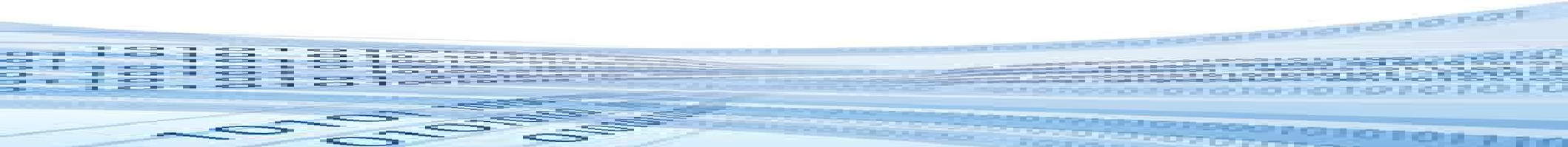
- Hay una zona privada al definir las clases que únicamente es utilizada por esa clase y alguna clase relacionada.
- Hay una zona pública que puede usarse en cualquier parte del código.

- **Herencia**

- Una clase puede heredar propiedades y métodos de otra.

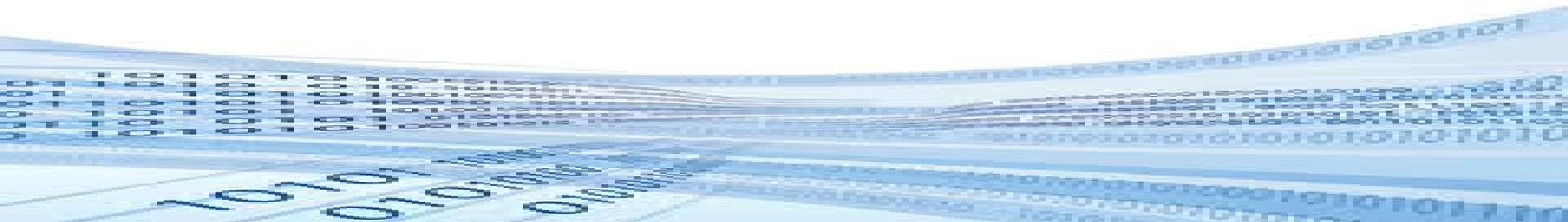
- **Polimorfismo**

- Un método de una clase puede tener varias definiciones diferentes.



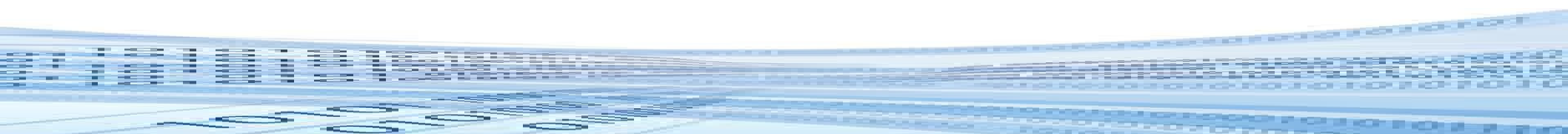
Concepto de herencia (I)

- 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.



Concepto de herencia (II)

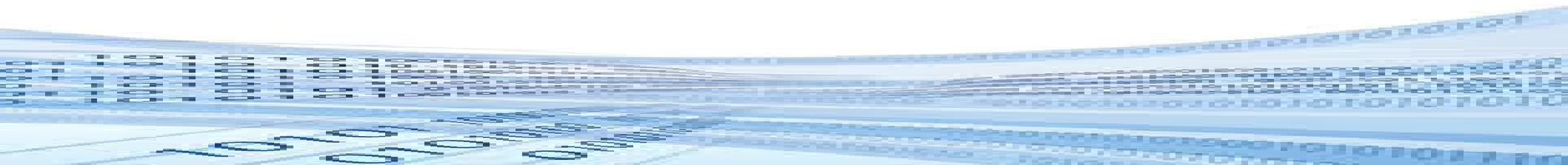
- En Java es especialmente relevante la herencia en dos aspectos:
 - La herencia se emplea (intrínsecamente) en el propio lenguaje a partir del conjunto de librerías que tiene.
 - El lenguaje apoya la definición de nuevas clases heredadas de las que ya están definidas.



Concepto de herencia (III)

- 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:

```
clase B extends A {...}
```



Herencia. Ejemplo 1 (I)

```
class Persona {
    public String nombre;
    public String apellidos;
    public int anyNaixement;

    public void mostrarInfo () {
        System.out.println ( "Datos personales:" + nombre + " "
+ apellidos + "(" + anyNaixement + ")");
    }
}

class Alumno extends Persona {
    private String grupo;
    private Horario horario;

    public void posarGrup (String grupo, Horario horario) {
        this.grup = grupo;
        this.horari = horario;
    }
}
```

Herencia. Ejemplo 1 (II)

- La clase Alumno hereda los atributos nombre, apellidos y anyNaixement de la clase Persona, así como el método mostrarInfo().
- Sobre un objeto de la clase Alumno se puede llamar al método mostrarInfo().

```
// Creación de un nuevo alumno
```

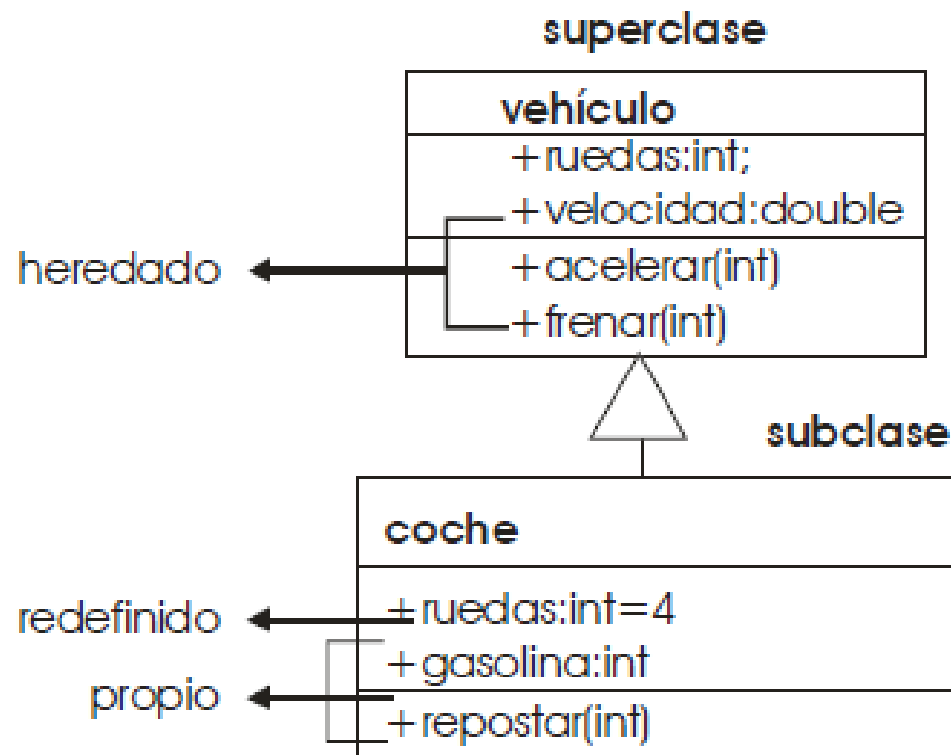
```
Alumno alu1 = new Alumno();  
// ...
```

```
// Llama al método que hereda de la clase Persona
```

```
alu1.mostrarInfo();
```


Herencia. Ejemplo 2 (I)

```
class coche extends vehiculo {  
  ...  
} //La clase coche parte de la definición de vehículo
```



Herencia. Ejemplo 2 (II)

```
class Vehiculo {
    protected int velocidad;
    protected int ruedas;
    public void parar() {
        velocidad = 0;
    }
    public void acelerar(int kmh) {
        velocidad += kmh;
    }
}

class Coche extends Vehiculo {
    private int gasolina;
    public Coche() {
        super();
        ruedas = 4;
    }
    public void repostar(int litros) {
        gasolina += litros;
    }
}

.....
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.acelerar(80); // Método heredado
        coche.repostar(12);
    }
}
```

Mecanismos de herencia (I)

- La subclase tiene (o hereda) **TODOS** los atributos y métodos de la superclase, aunque no todos los miembros tienen por qué ser accesibles:
 - Serán accesibles todos los métodos y propiedades *protected*, *public* y *"de paquete"*.
 - No serán accesibles los métodos y propiedades *private*.

Tipo de acceso	Palabra reservada	Ejemplo	Acceso desde una subclase del mismo paquete	Acceso desde una subclase de otro paquete
Privado	private	private int PPrivada;	No	No
Sin especificar		int PSinEspecificar;	Sí	No
Protegido	protected	protected int PProtegida;	Sí	Sí
Publico	public	public int PPublica;	Sí	Sí

Mecanismos de herencia (II)

- En la clase derivada pueden añadirse atributos (que generalmente serán privados) y métodos adicionales.
- La subclase **NO** hereda los constructores:
 - Cada nueva clase (incluso las derivadas), debe definir sus constructores.
 - Si no se implementa ningún constructor, se genera uno predeterminado sin argumentos.
 - Orden de ejecución de los constructores: desde el nivel más alto de la jerarquía de herencia, hasta el más específico.

Mecanismos de herencia (III)

- Podemos definir una propiedad en la subclase con el mismo nombre que en la superclase:
 - La propiedad de la superclase queda "oculta".
- 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 el atributo o método con su nombre (o `this.nombre`), estamos refiriéndonos al atributo o método de la subclase. Para referirnos a la superclase, tendremos que invocar con `super.nombre`.

Mecanismos de herencia (IV)

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

super

- La palabra reservada ***super*** nos 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.

super.Ejemplo

```
public class Vehiculo {
    protected double velocidad;
    ...
    public void acelerar(double kmh) {
        velocidad += kmh;
    }
}

public class Coche extends Vehiculo {
    protected double gasolina;
    ...
    public void acelerar(double kmh) {
        super.acelerar(kmh);
        gasolina*=0.9;
    }
}
```

- La llamada `super.acelerar` llama al método `acelerar` de la clase vehículo.

- Se puede llamar a un constructor de una superclase utilizando la sentencia `super()`.

```
public class Vehiculo {
    protected double velocidad;
    ...
    public Vehiculo(double v) {
        velocidad = v;
    }
}

class Coche extends Vehiculo {
    protected double gasolina;
    ...
    public Coche(double v, double g) {
        super(v); // Llama al constructor
        gasolina = g;
    }
}
```


Herencia y constructores (I)

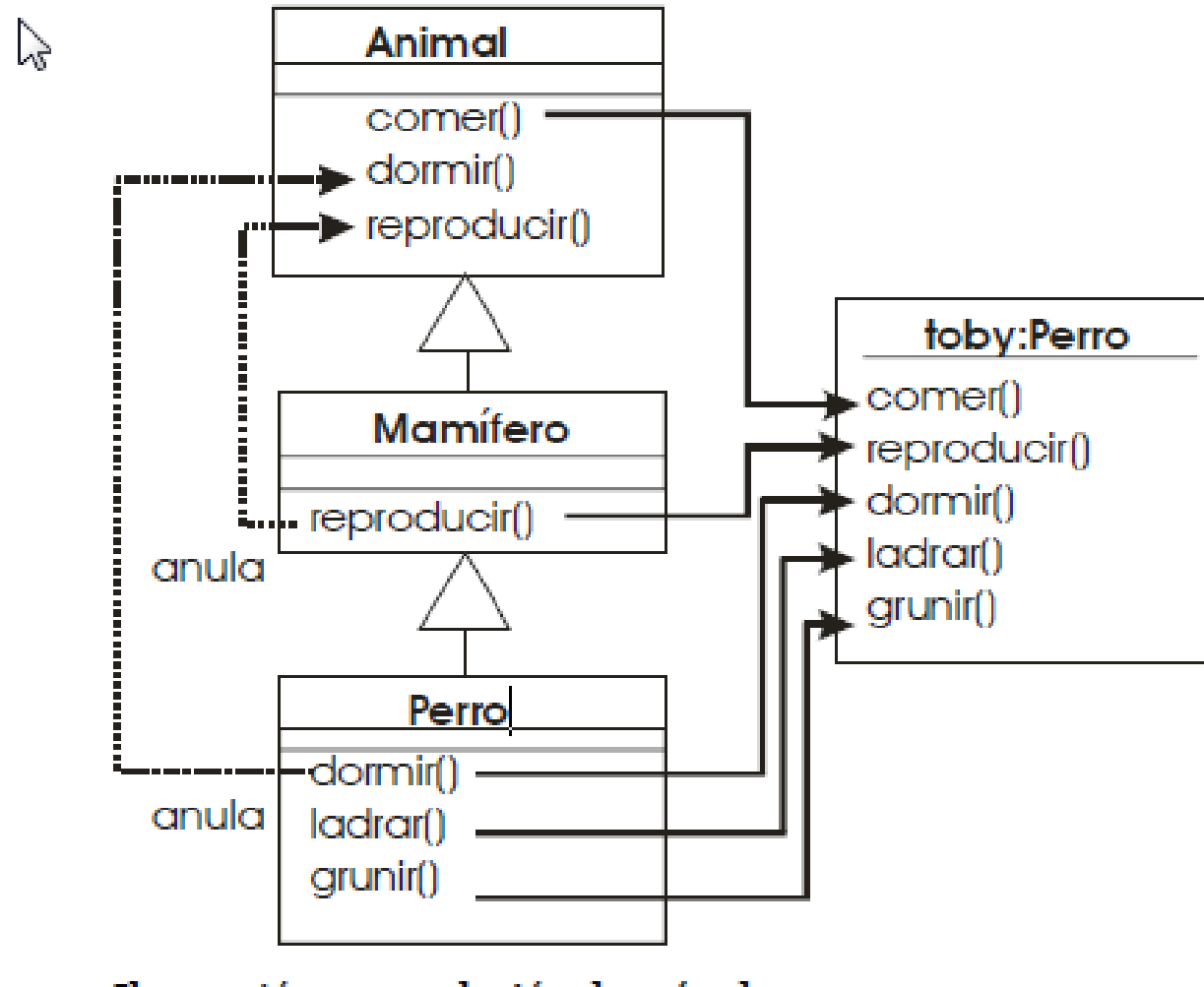
- Los constructores tienen la posibilidad de invocar a otro constructor de su propia clase con la sentencia **this(...)**.
- Los constructores de las subclases tienen la posibilidad de invocar a los constructores de las superclases con la sentencia **super(...)**.
- La llamada **super (...)** o **this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

Herencia y constructores (II)

Por defecto Java realiza estas dos 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.

Mecanismos de herencia. Ejemplo



Ejercicio práctico 1 (I)

```
public class Medico {  
    protected 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 cirugía  
    }  
  
    public void hacerIncision() {  
        // Código para realizar una incisión  
    }  
}
```

Ejercicio práctico 1 (II)

- 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 (I)

- 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 averiguar y agrupar.
- Utilizaremos herencia para evitar duplicar código en las subclases.

Ejercicio práctico 2 (II)

- 1** Busca objetos que tengan atributos y comportamientos comunes.

¿Qué tienen en común estos 6 tipos de objetos?
(lo veremos en el paso 3, los comportamientos)

¿Cómo están relacionados estos tipos?
(definiremos la herencia y sus relaciones,
pasos 4 y 5)



Ejercicio práctico 2 (III)

- Atributos:

- **picture**: Foto del animal.
- **food**: Comida (carne o hierba).
- **hunger**: Nivel de hambre (entero).
- **boundaries**: Medidas (alto x ancho) del espacio donde se encuentran.
- **location**: Coordenadas donde está el animal.

- Métodos:

- **makeNoise()**: Ruido del animal.
- **eat()**: Comportamiento al comer.
- **sleep()**: Comportamiento al dormir.
- **roam()**: Comportamiento cuando no come ni duerme.

2

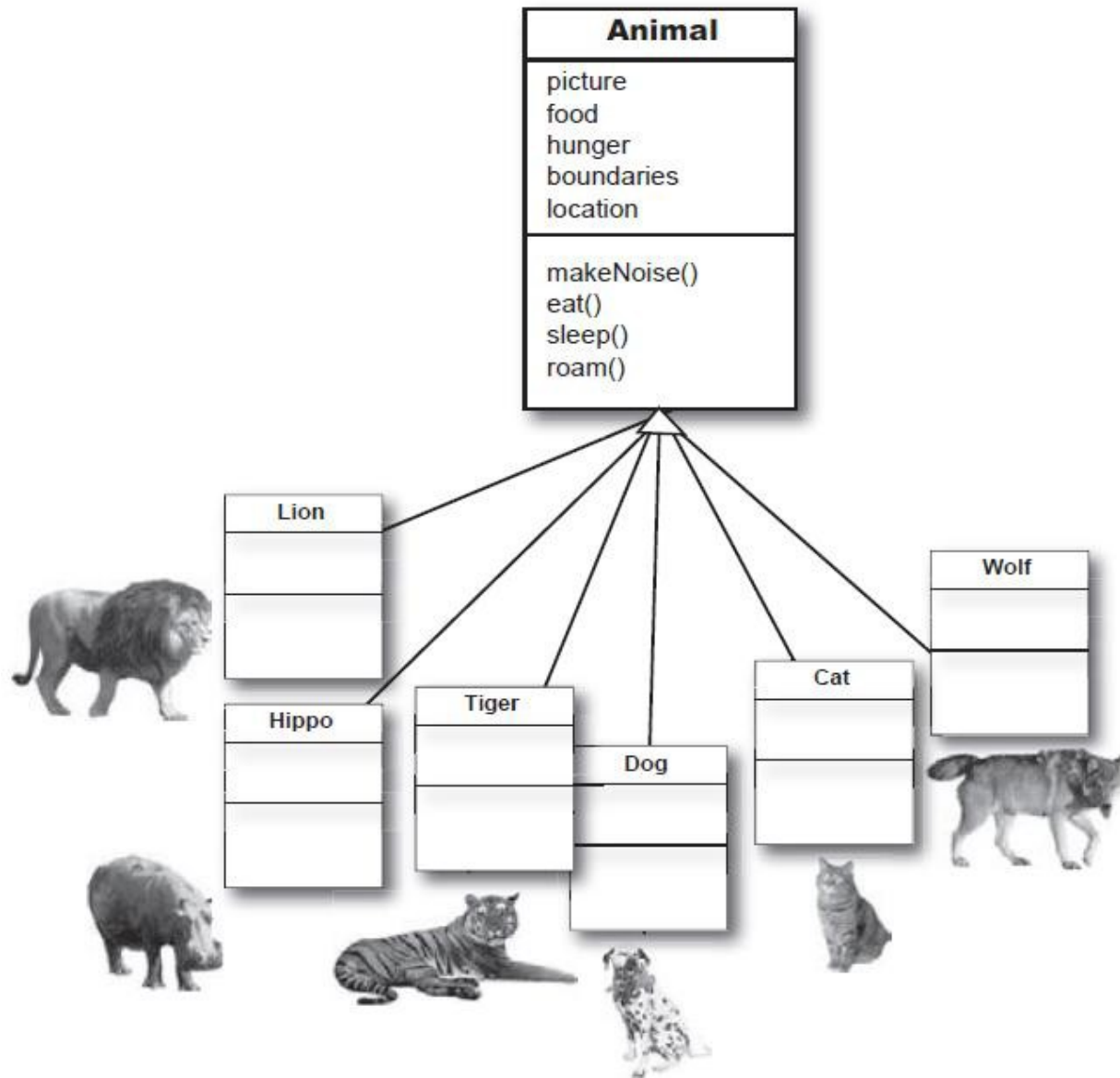
Diseñaremos una clase que represente el estado y comportamiento común de todos los animales

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

Y pondremos las variables y los métodos que todos los animales necesitan:

Animal
picture food hunger boundaries location
makeNoise() eat() sleep() roam()

Ejercicio práctico 2 (IV)



Ejercicio práctico 2 (V)

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

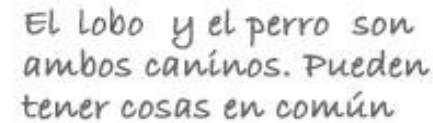
Observa la clase *Animal*. Decidiremos si los métodos `eat()` y `makeNoise()` serán sobrecargados de forma individual en las subclases.

Animal
picture food hunger boundaries location
makeNoise() eat() sleep() roam()



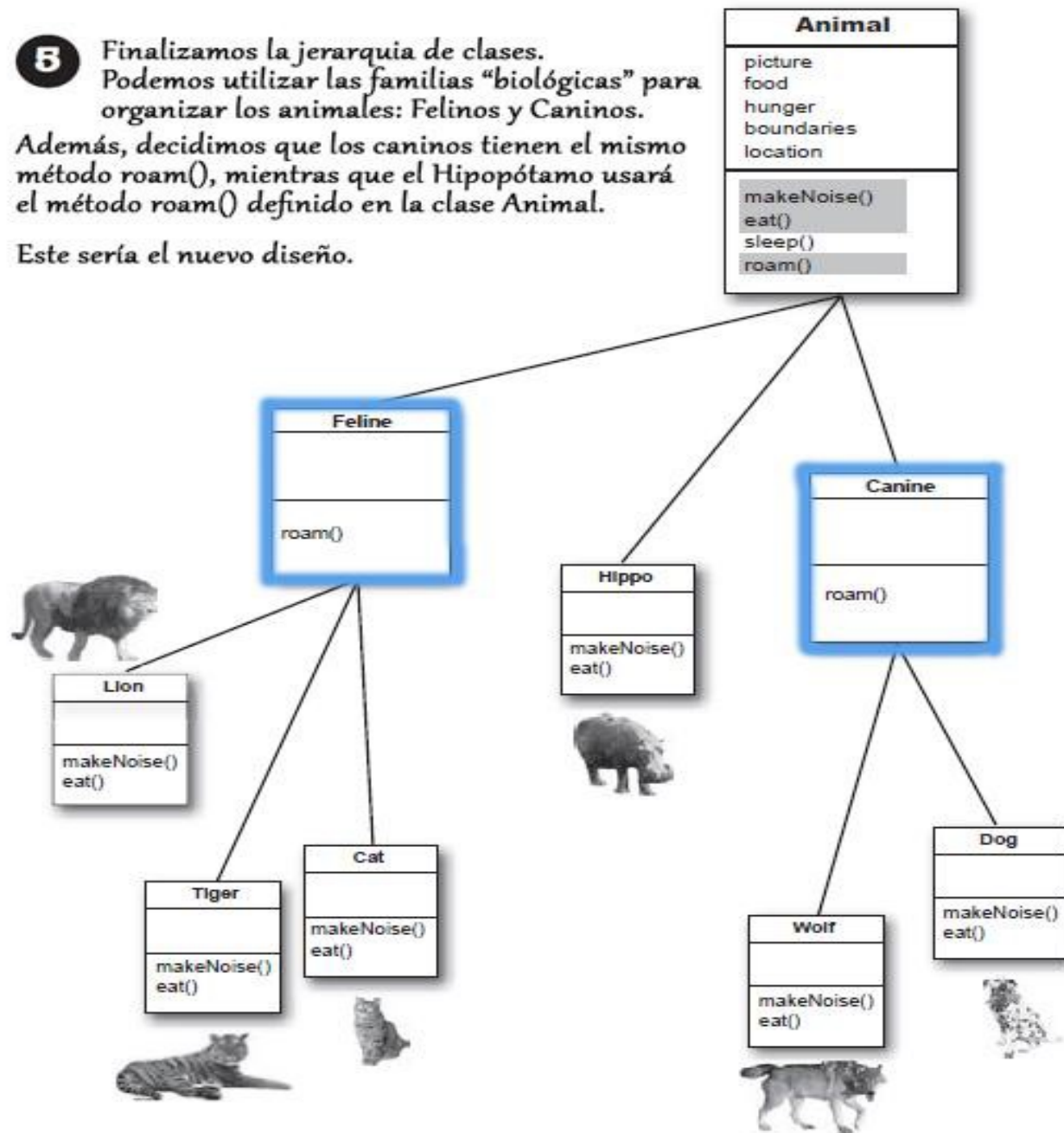
De momento sobreescribimos los métodos `makeNoise()` y `eat()`, que definirán el comportamiento específico de cada animal. Por ahora, parece que los métodos `sleep()` y `roam()` pueden permanecer genéricos.

El león, el tigre y el gato
podrían tener cosas en
común



Ejercicio práctico 2 (VII)

- 5** Finalizamos la jerarquía de clases. Podemos utilizar las familias “biológicas” para organizar los animales: Felinos y Caninos. Además, decidimos que los caninos tienen el mismo método `roam()`, mientras que el Hipopótamo usará el método `roam()` definido en la clase `Animal`. Este sería el nuevo diseño.



La JVM ejecutará el método más específico. Si no lo encuentra en la clase, se desplazará más arriba en la jerarquía de clases.

Ejercicio práctico 2 (VIII)

- Analizamos la clase Wolf ...
 - ¿Cuántos métodos tiene?
 - ¿Dónde está definido cada método?
 - ¿Qué pasará si ejecutamos las instrucciones que se muestran?
 - ¿A qué método se está invocando?

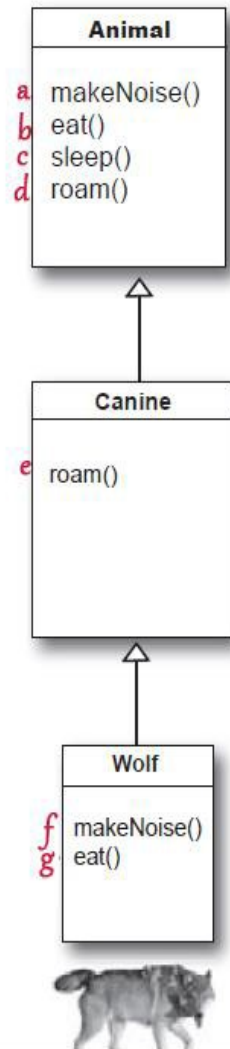
```
Wolf w = new Wolf();
```

```
1 w.makeNoise();
```

```
2 w.roam();
```

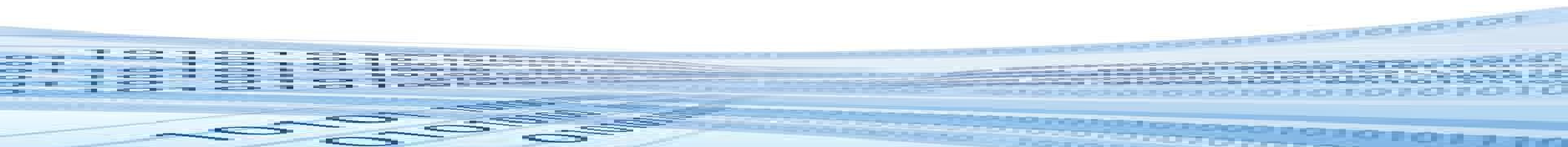
```
3 w.eat();
```

```
4 w.sleep();
```



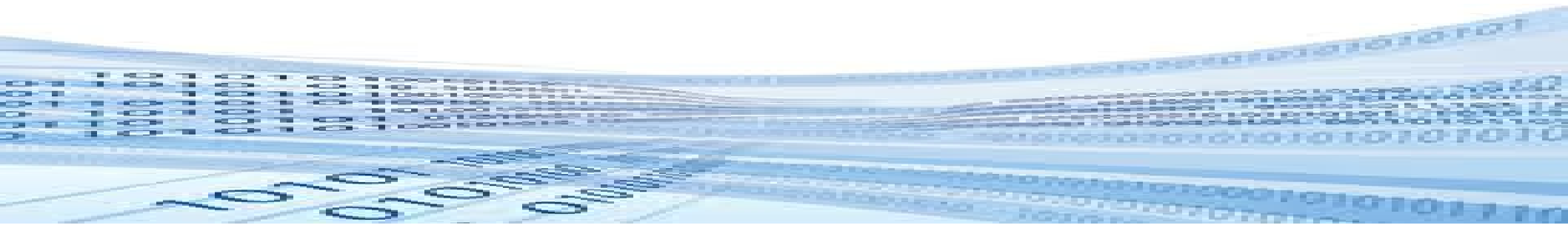
Test SER o NO SER

- 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.
 - Una bañera NO ES un baño, pero un baño TIENE una bañera.
- La subclase podrá hacer cualquier cosa que haga la superclase (o incluso más cosas).



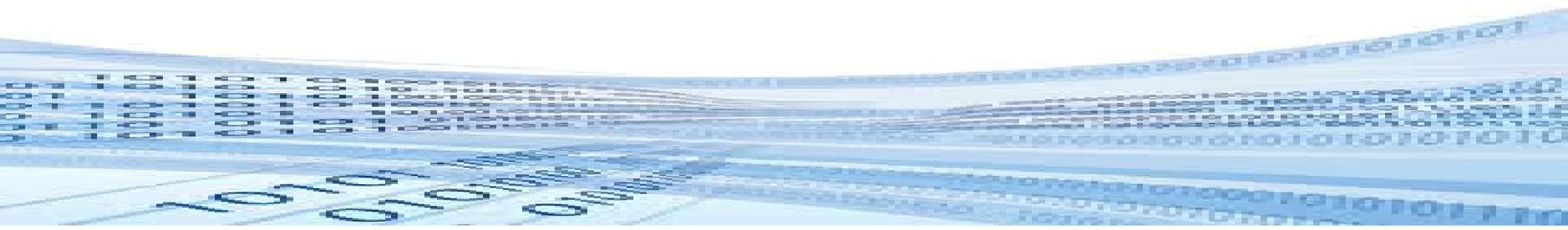
¿Cuándo hacer uso de la herencia?

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



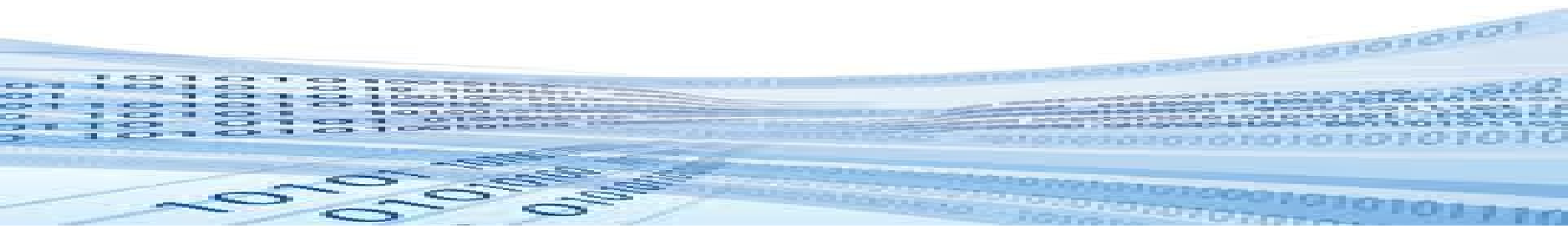
Ventajas de la herencia

- Evitamos código duplicado:
 - 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 de puntos importantes (I)

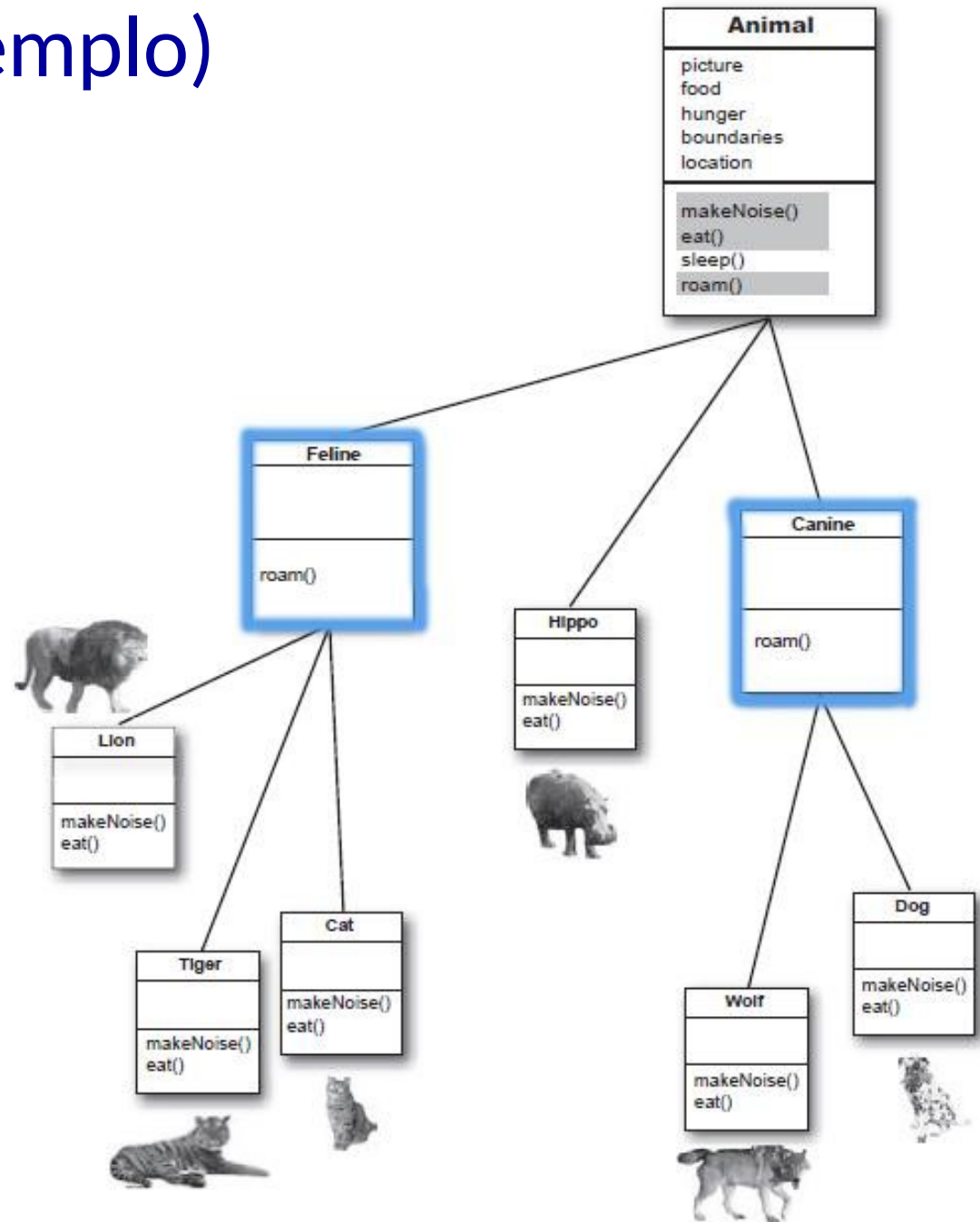
- Una subclase **EXTIENDE** a una superclase.
- Una subclase **hereda todos los miembros** de una superclase, pero dependiendo de su visibilidad, serán accesibles o no.
- Los **métodos** heredados **pueden ser sobreescritos** en la subclase.
- Los **atributos no se sobreescriben**; si se vuelven a definir, 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 de puntos importantes (II)

- 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 felino extends animal)
 - B es A (un felino es un animal)
 - Entonces ...
 - $C \text{ es } B \rightarrow C \text{ es } A$ (un gato es un animal)

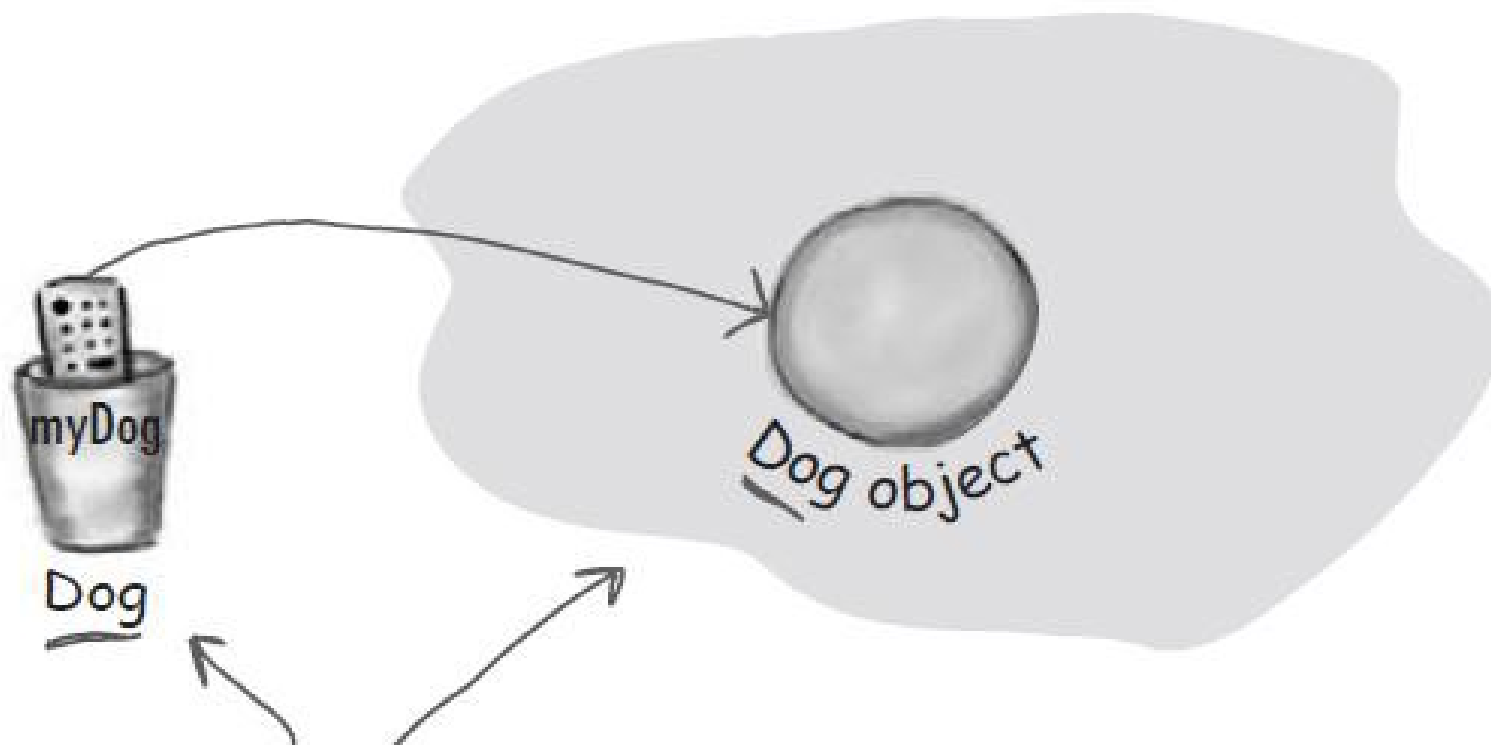
Clases de animales (Ejemplo)



Polimorfismo (I)

- Sin polimorfismo ...

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

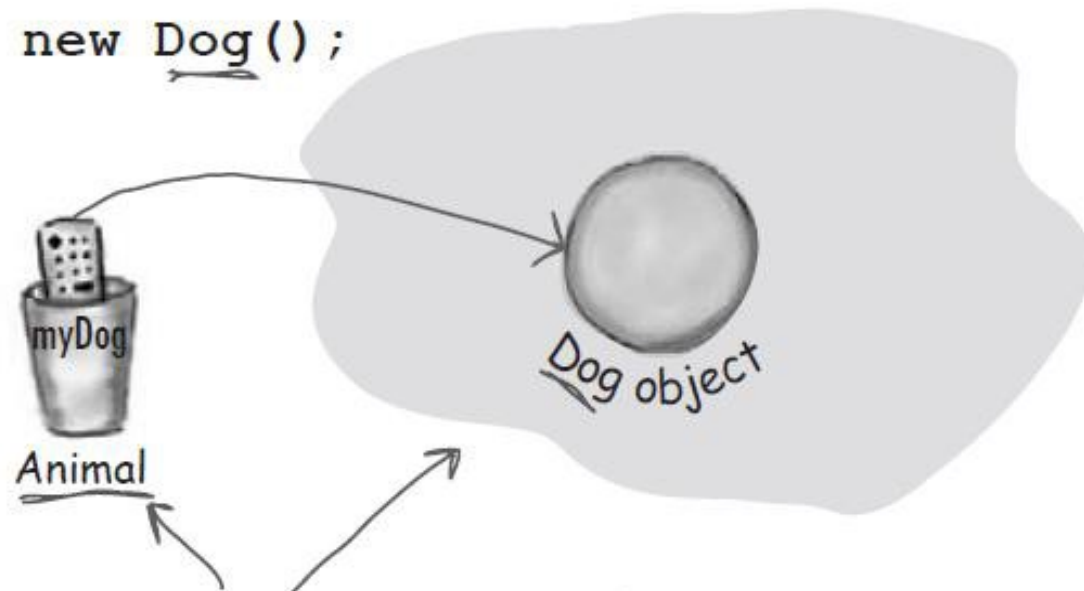


La variable de referencia myDog y el objeto de tipo Dog son del mismo tipo

Polimorfismo (II)

- Con polimorfismo: la referencia puede ser una superclase del objeto.

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



Estos dos NO son del mismo tipo. La variable de referencia está declarada como Animal pero el objeto es de tipo Dog.

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; // implícito
```

```
Persona p = (Persona) a; // explícito; no es necesario
```

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

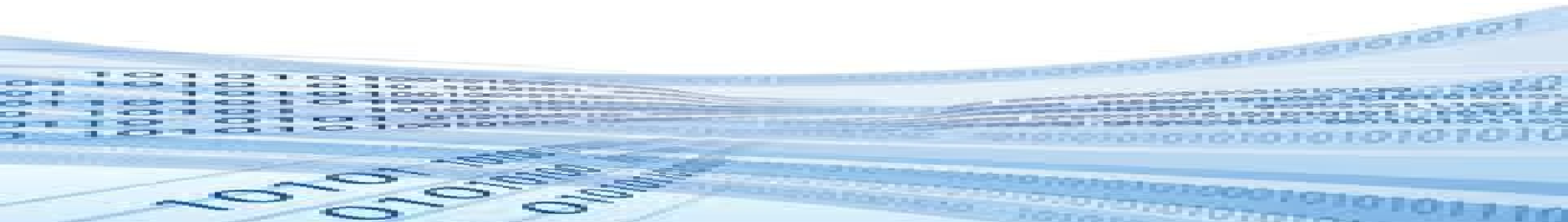
Polimorfismo. *Downcasting*

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

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

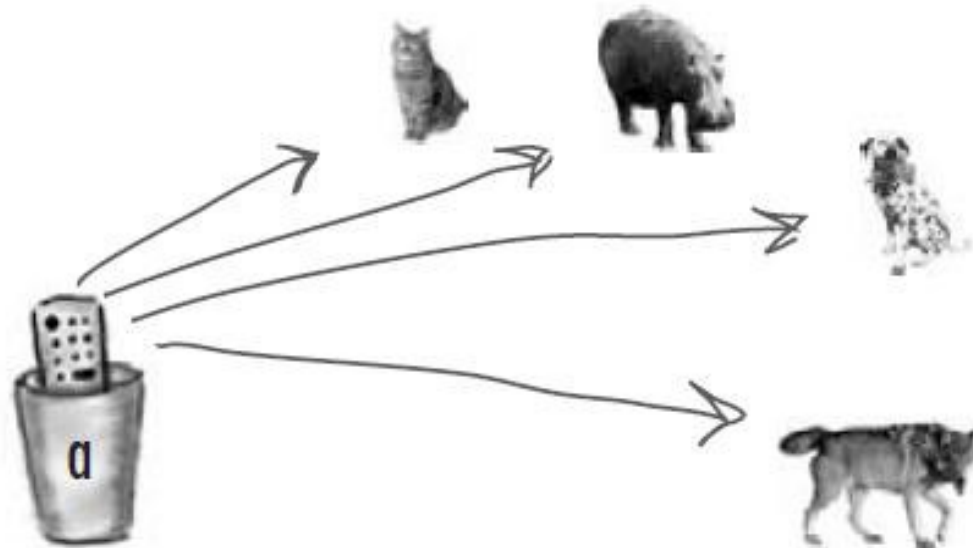
```
Alumno a = (Alumno) p;    // explícito; es necesario
```

- 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: parámetros y valor devuelto (I)

```
class Vet {  
    public void giveShot(Animal a) {  
        // Podemos hacer lo que  
        // queramos con el animal  
        a.makeNoise();  
    }  
}
```



El parámetro *a* de tipo *Animal* podría tomar cualquier valor de animal como argumento. El método *makeNoise()* se ejecutará según el tipo del animal pasado como argumento.

Polimorfismo: parámetros y valor devuelto (II)

```
class PetOwner {  
    public void start() {
```

```
        Vet v = new Vet();
```

```
        Dog d = new Dog();
```

```
        Hippo h = new Hippo();
```

```
        v.giveShot(d);
```

```
        v.giveShot(h);
```

```
    }
```

```
}
```

Creemos varias variables de referencia de diferentes clases. Las pasamos por parámetro al método giveShot().

Se ejecuta el método makeNoise() de Dog

Se ejecuta el método makeNoise() de Hippo

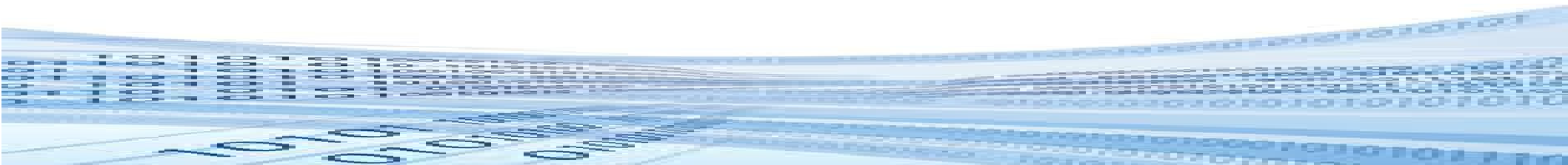
Dynamic Binding (Elección dinámica de método)

- Ejemplo:

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

```
Alumno a = (Alumno) p;
```

- Supongamos que la clase Persona dispone de un método imprime() y que este método se ha sobrescrito en la clase Alumno.
- Podemos escribir:
 - p.imprime()
 - a.imprime()
- Las dos invocaciones al método ejecutarán el código de la clase Alumno.



Polimorfismo (Arrays)

Se declara un vector de objetos de tipo Animal.
En otras palabras, un vector que almacenará
objetos de tipo Animal.

```
Animal[] animals = new Animal[5];
```

```
animals [0] = new Dog();
```

```
animals [1] = new Cat();
```

```
animals [2] = new Wolf();
```

```
animals [3] = new Hippo();
```

```
animals [4] = new Lion();
```

Pero mira qué podemos hacer. ¡Podemos incluir
cualquier subclase de Animal en el vector de
Animales!

Y aquí viene lo mejor del polimorfismo.
Podemos recorrer el vector e ir llamando
a los métodos, y cada uno de ellos, actúa
de forma correcta!!

```
for (int i = 0; i < animals.length; i++) {
```

```
    animals[i].eat();
```

Cuando i vale 0, el animal será un Dog y se ejecutará
el método eat() de Dog. Para i valor 1, será un Cat y comerá
como un gato.

```
    animals[i].roam();
```

Lo mismo para el método roam()

```
}
```

Polimorfismo y clases abstractas

Esto puede resultar extraño:

```
Animal anim = new Animal();
```

una variable de referencia
a un objeto de tipo *Animal*



Animal

Animal object

Son del mismo tipo, pero... ¿qué tipo de animal será?

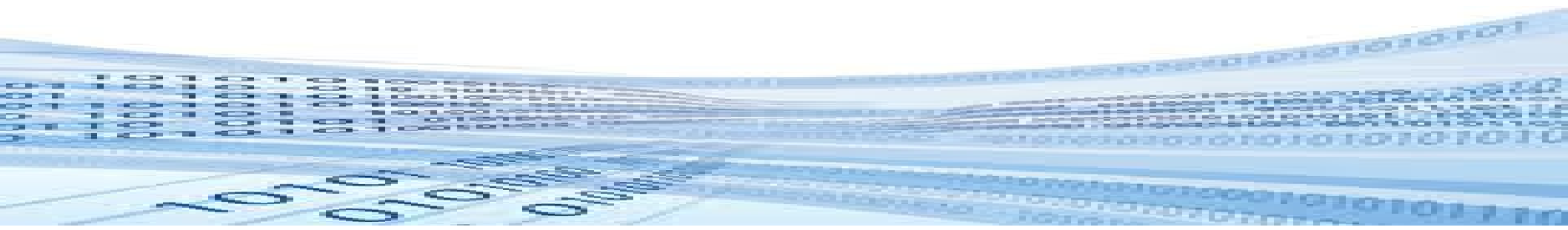
¿A qué animal se parece un new Animal()?

**Objetos
que dan
miedo**



Clases abstractas

- Tiene sentido crear un Tigre o un Lobo.
- 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 Lobo, un Tigre o un Perro.
- Marcando una clase como abstracta (*abstract*), evitamos que pueda ser instanciada, es decir que se creen objetos de esa clase.

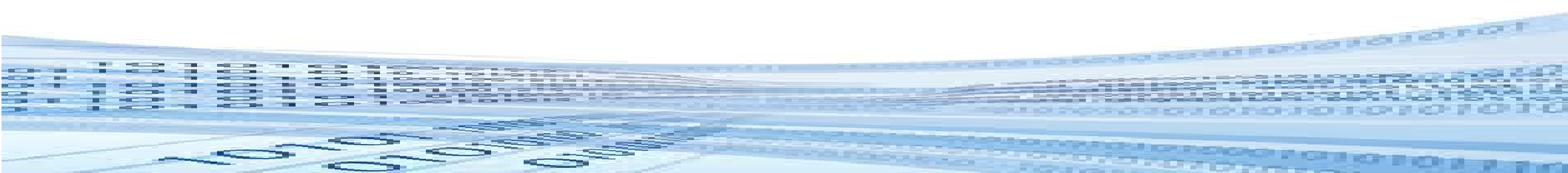


Clases abstractas

- 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 ...
```

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



clases abstractas

```
abstract public class Canine extends Animal
{
    public void roam() { }
}
```

```
public class MakeCanine {
```

```
    public void go() {
```

```
        Canine c;
```

```
        c = new Dog();
```

```
        c = new Canine();
```

```
        c.roam();
```

```
    }
```

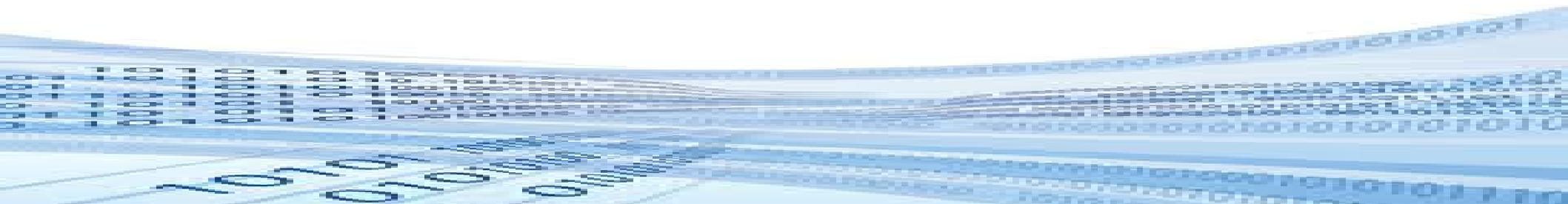
```
}
```

Esto es correcto. porque podemos asignar un objeto de una subclase a una variable de referencia de la clase, aunque sea abstracta

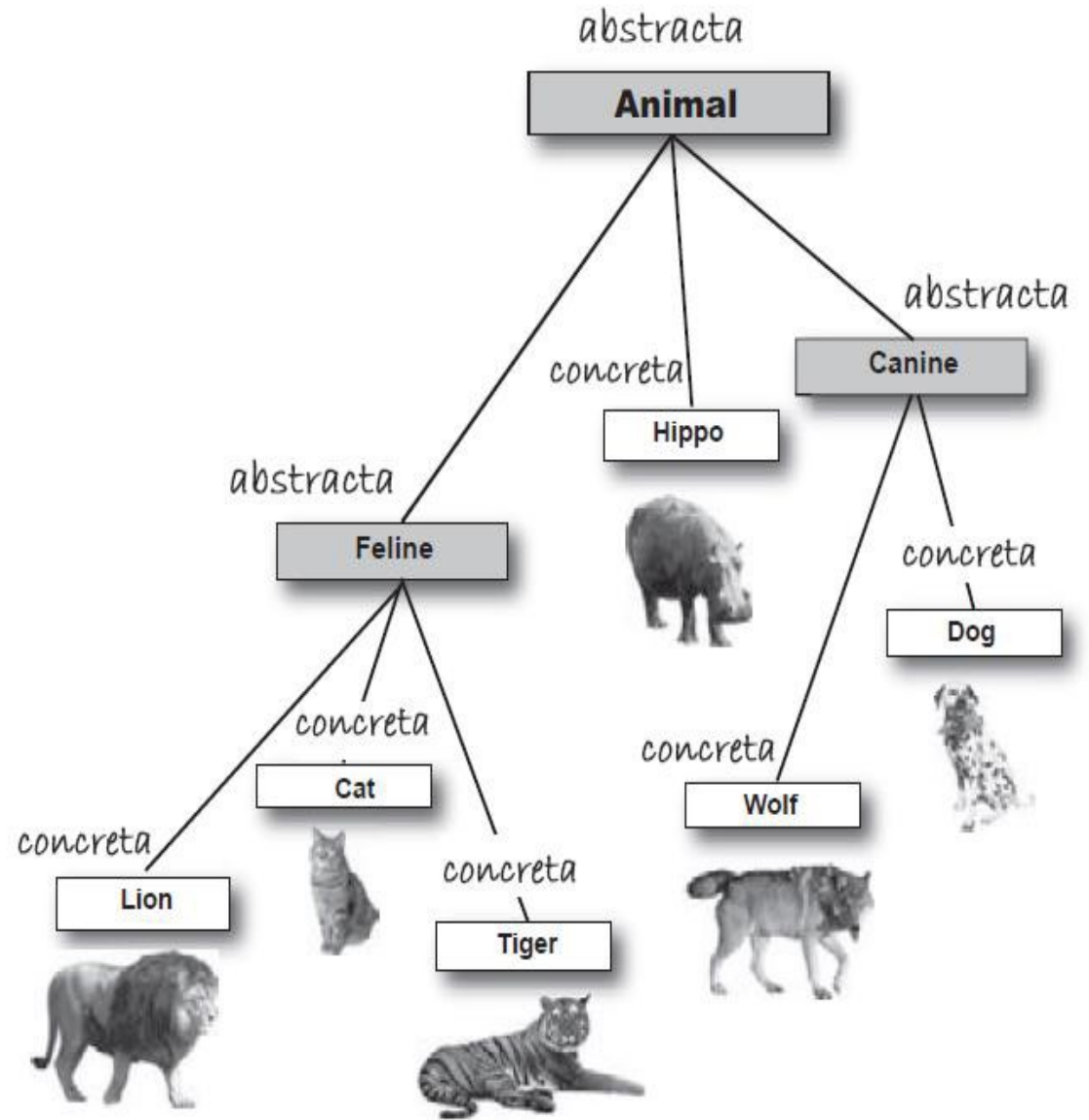
La clase Canine es asbtracta, por eso el compilador NO permite hacer esto

Clases abstractas

- Las clases abstractas, a excepción de las que tienen métodos no abstractos, no tienen uso, ni valores, únicamente sirven para heredar de ellas.
- Cuando utilizamos clases abstractas, se crean objetos de sus subclasses.
- En la API de Java encontraremos muchas clases abstractas. Por ejemplo, la clase *Component* es abstracta, pero la clase *JButton* es concreta.



Ejemplo: Animales




Métodos abstractos (abstract)

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

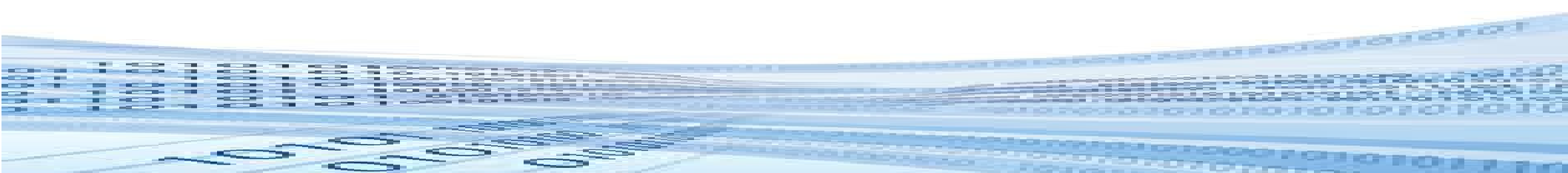
```
public abstract void eat();
```

El método no tiene
código, sin llaves y
acaba en punto y coma



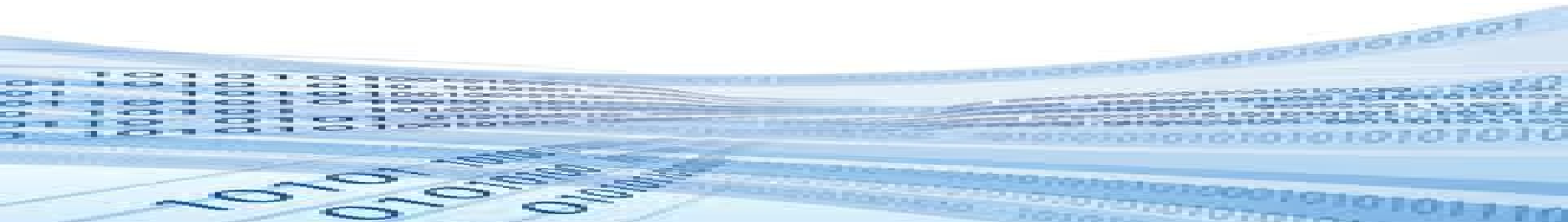
Métodos abstractos (*abstract*)

- Únicamente existen para ser heredados (polimorfismo).
- 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 de **Canine** y de **Animal**.



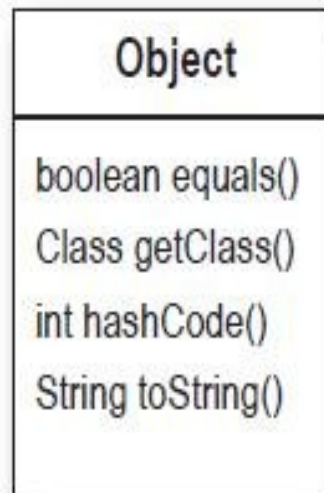
La clase Object (I)

- 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.

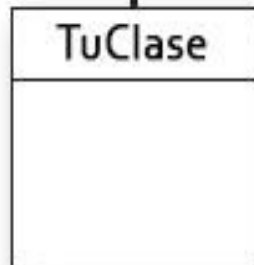


La clase Object (II)

- ¿Cuál es el comportamiento de cualquier objeto?
- Automáticamente, todos los objetos tienen los siguientes cuatro métodos:



Aquí hay algunos métodos de la clase Object



Cualquier clase que creamos hereda todos los métodos de la clase Object. Los heredan y se pueden utilizar, aunque no lo supieras

Método equals(Object o)

① equals(Object o)

```
Dog a = new Dog();  
Cat c = new Cat();
```

```
if (a.equals(c)) {  
    System.out.println("true");  
} else {  
    System.out.println("false");  
}
```

File Edit Window Help Stop

```
% java TestObject  
  
false
```

Nos dice si dos objetos
se consideran iguales
(en este caso, no lo son)

Método `getClass()`

② `getClass()`

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

File Edit Window Help Faint

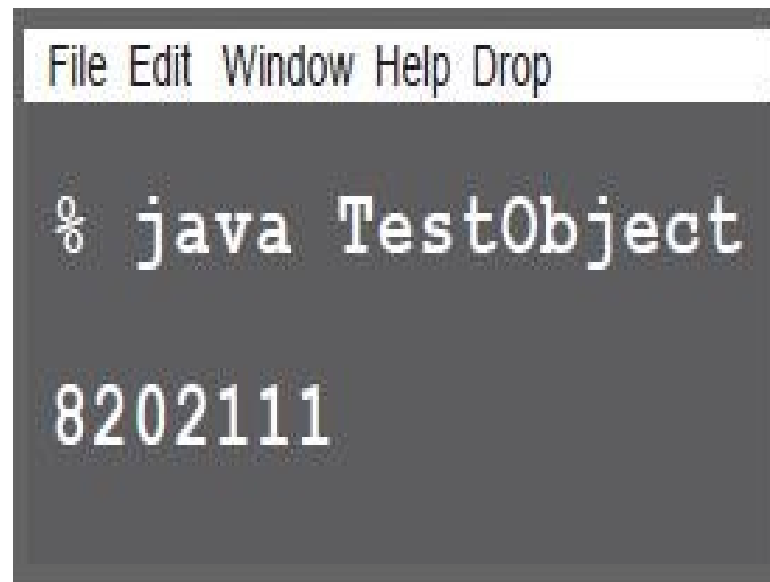
```
% java TestObject  
  
class Cat
```

Nos devuelve el
nombre de la
clase del objeto
instanciado

Método *hashCode()*

③ **hashCode()**

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



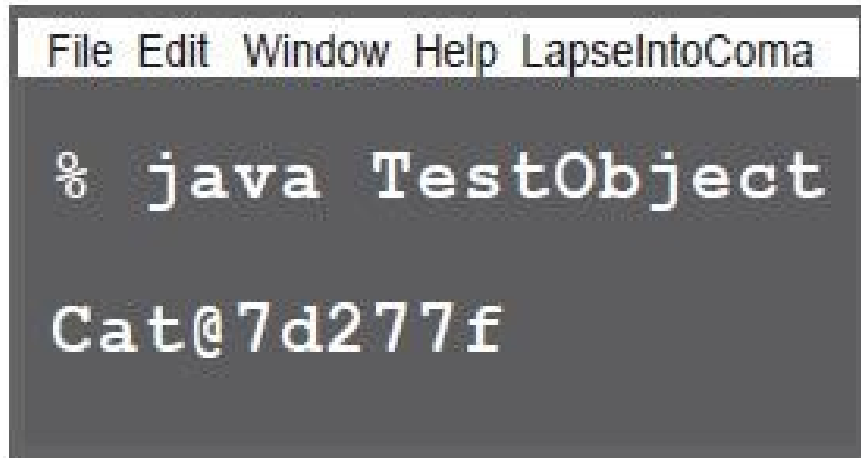
The screenshot shows a terminal window with a menu bar (File, Edit, Window, Help, Drop). The command `% java TestObject` has been executed, and the output `8202111` is displayed.

Devuelve el *hashCode* del objeto (por ahora, piensa que es un identificador único para cada objeto)

Método toString()

④ toString()

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

A screenshot of a Java IDE window titled 'LapseIntoComa'. The menu bar shows 'File', 'Edit', 'Window', 'Help', and 'LapseIntoComa'. The main editor area contains the following text:

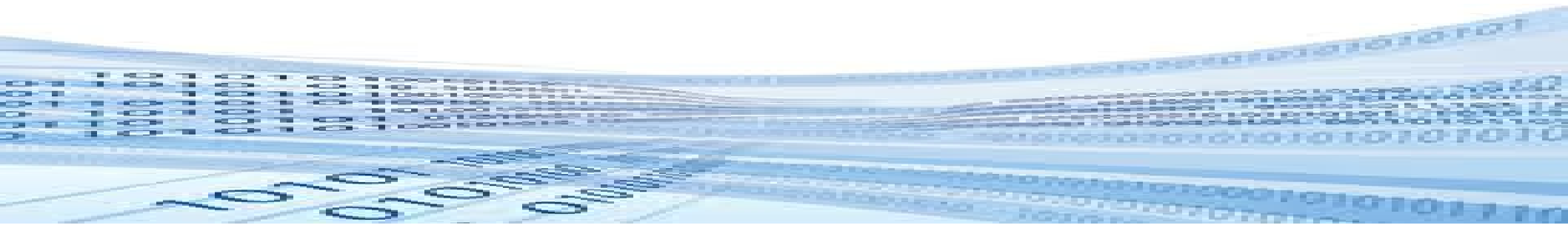
```
% java TestObject  
Cat@7d277f
```

The text is displayed in a monospaced font on a dark background.

Devuelve una cadena con el nombre de la clase y algún otro código del que de momento no nos preocuparemos

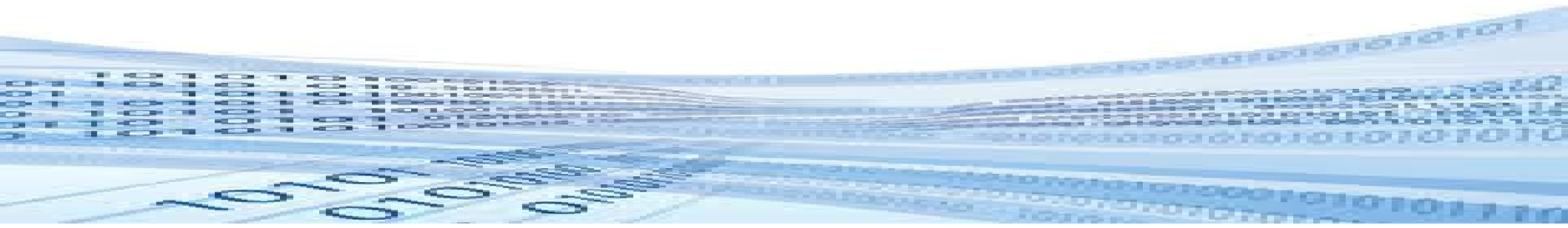
La clase Object (III)

- No es una clase abstracta, pero es especial.
- Todas las clases pueden heredar sus métodos sin necesidad de sobreescribirlos.
- Podemos sobreescribir algunos de sus métodos, otros están definidos como *final* y no podremos.
- Sería recomendable sobreescribir los 4 métodos anteriores.
- Es posible crear objetos de tipo Object, pero lo haremos más adelante para casos especiales.



La clase Object (IV)

- No es correcto emplear la clase Object para:
 - `Object o = new Ferrari();`
 - `o.irRapido();`
 - `// Esto no compila`
- El método `irRapido()` no pertenece a la clase Object



La clase Object (V)

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();  
Dog aDog = new Dog();  
myDogArrayList.add(aDog);  
Dog d = myDogArrayList.get(0);
```

← Creamos una ArrayList para almacenar objetos de tipo Dog

← Creamos un perro (Dog)

← Añadimos el perro a la lista

← Asignamos el perro de la lista a una nueva variable de referencia a Dog (d). Piensa que el método get() ahora devuelve un perro, al ser definido el ArrayList para almacenar perros

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();  
Dog aDog = new Dog();  
myDogArrayList.add(aDog);
```


← ArrayList para almacenar cualquier objeto

← Creamos un Dog

← Añadimos un Dog a la lista

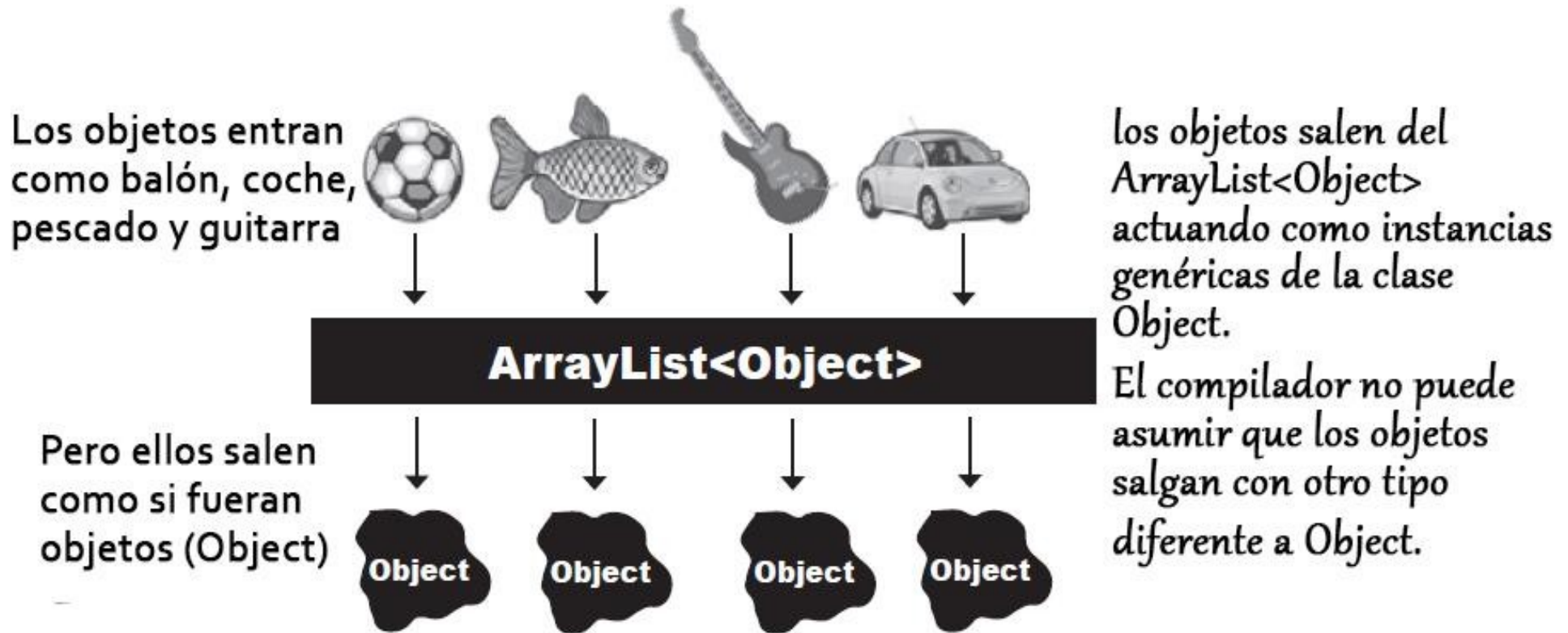
(Estos dos pasos son los mismos que antes)

Pero, ¿qué pasa cuando intentamos obtener un objeto tipo Dog y se lo asignamos a una variable de referencia?

 `Dog d = myDogArrayList.get(0);`

NO!! ¡¡No compila!! Cuando usamos un ArrayList<Object>, el método get() devuelve un tipo Object. El compilador conoce que el ArrayList hereda de Object pero no sabe si el elemento es un perro o no.

La clase Object (VI)



Cuando un *Dog* no actúa como un *Dog*... (I)

```
public void go() {
```

```
    Dog aDog = new Dog();
```

```
    Dog sameDog = getObject(aDog);
```

```
}
```

```
public Object getObject(Object o) {
```

```
    return o;
```

```
}
```

!!Esta línea no funciona!!

Incluso aunque el método devuelva una referencia al mismo objeto que se pasó por parámetro, el tipo de objeto devuelto será de tipo *Object*

MAL
☹

Se está devolviendo una referencia al mismo objeto, pero devolviendo un objeto de tipo *Object*. La llamada es legal, pero la asignación no

Cuando un *Dog* no actúa como un *Dog*... (II)

BIEN

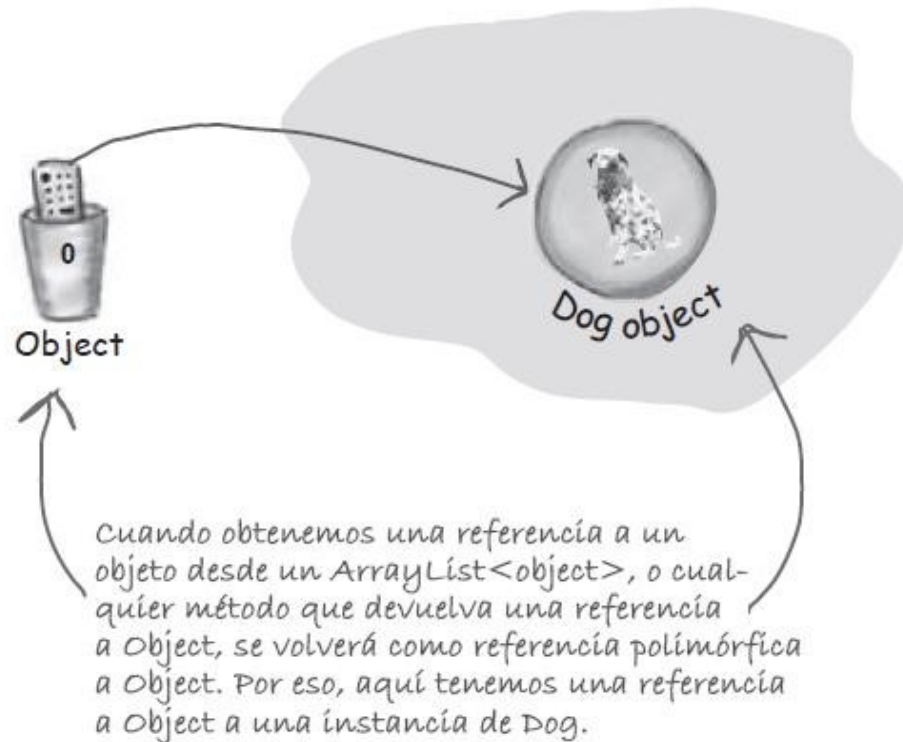


```
public void go() {  
    Dog aDog = new Dog();  
    Object sameDog = getObject(aDog);  
}  
  
public Object getObject(Object o) {  
    return o;  
}
```

Esto funciona porque es posible asignar cualquier cosa a una referencia de tipo *Object* y las clases pasan todas el test SER con el tipo *Object*.

Cualquier objeto en Java es una instancia de tipo *Object*, porque cualquier clase en Java tiene la clase *Object* en la parte superior de su árbol de herencia.

Cuando un *Dog* no actúa como un *Dog*... (III)



```
Object o = al.get(index);
```

```
int i = o.hashCode();
```

Esto es correcto. La clase `Object` tiene un método `hashCode()` que podremos llamar desde cualquier objeto de Java

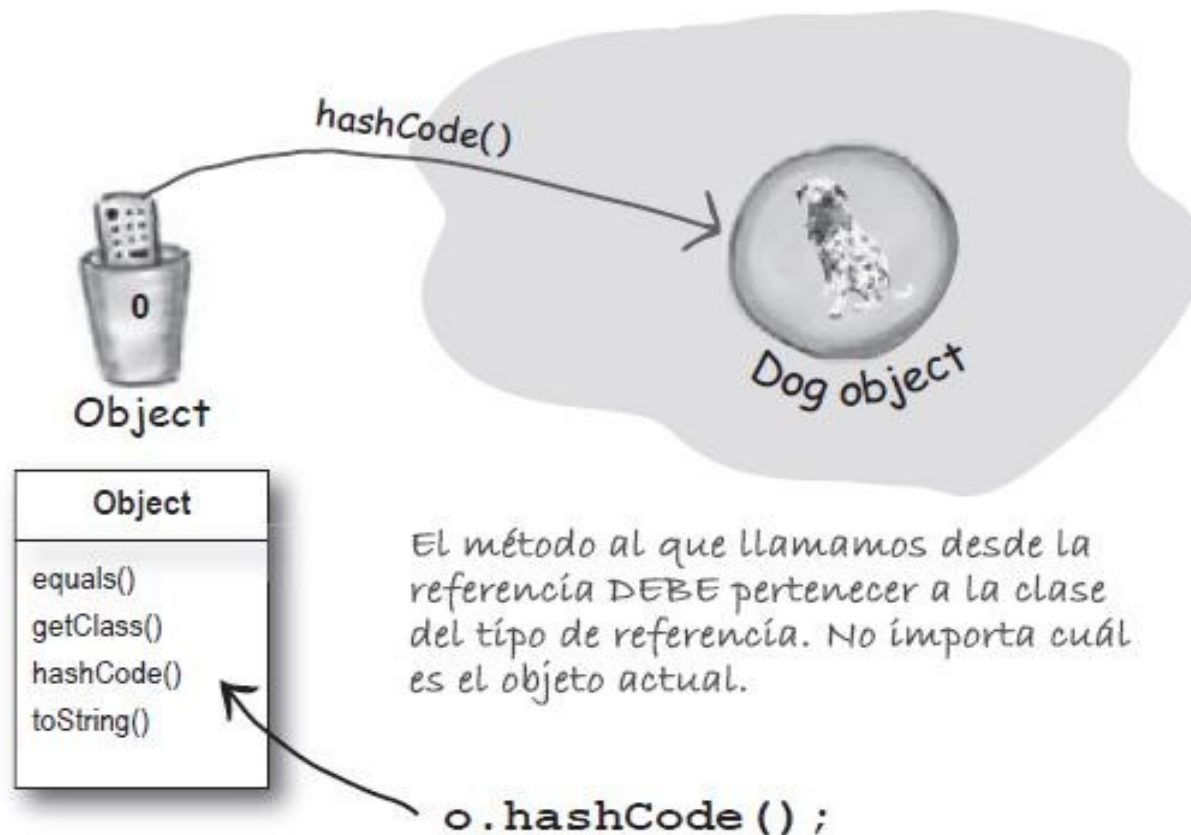
No compila!! →

```
o.bark();
```

No podemos hacer esto!! La clase `Object` no tiene ni idea de los que significa el método `bark()`. Incluso aunque tú sepas que es un `Dog`, el compilador no lo sabe.

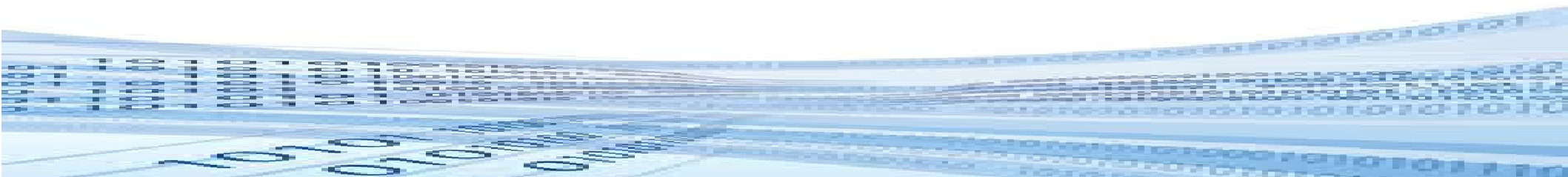
Cuando un *Dog* no actúa como un *Dog*... (IV)

El compilador decide si es posible llamar al método basándose en el tipo de referencia, no en el tipo de objeto.



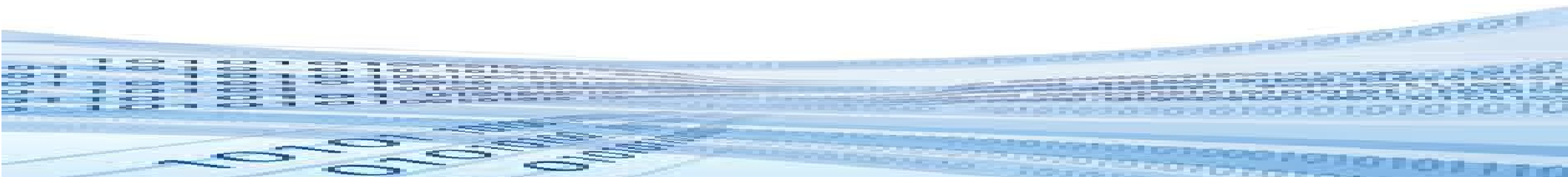
La variable de referencia *o* está declarada de tipo *Object*, por eso sólo es posible llamar a los métodos de la clase *Object*.

Interfaces



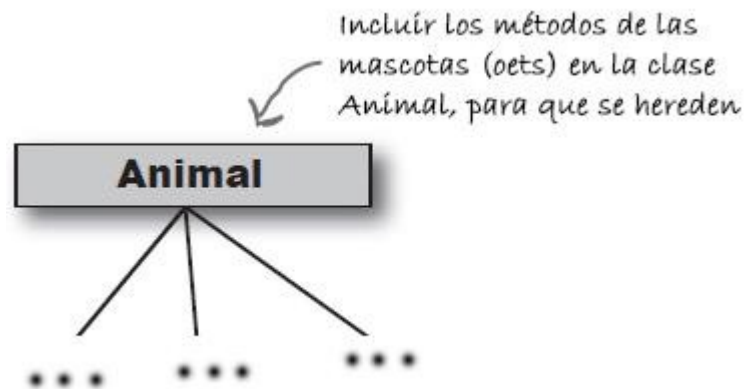
Comportamientos comunes

- 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?



Diseño 1: Comportamiento en la clase Animal

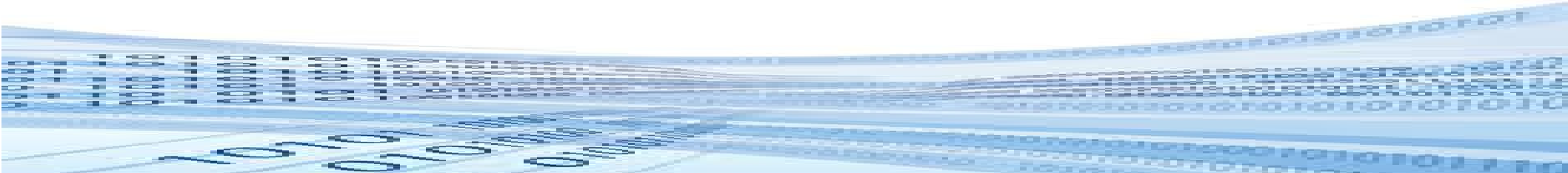
- Definimos los métodos de las mascotas en la clase Animal.



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

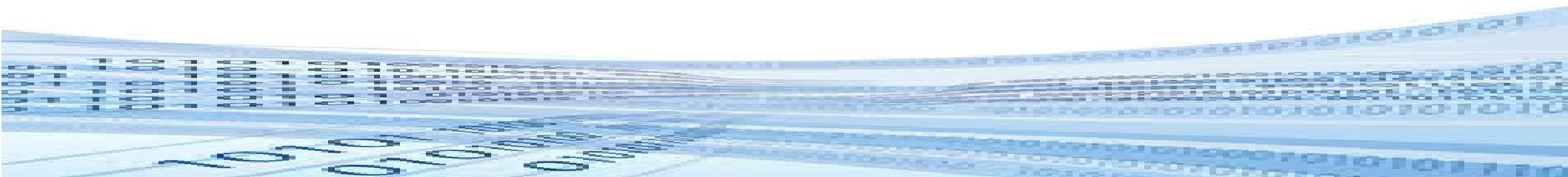
Diseño 2: Métodos *abstract* en la clase Animal

- 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.
 - No parece una buena solución.



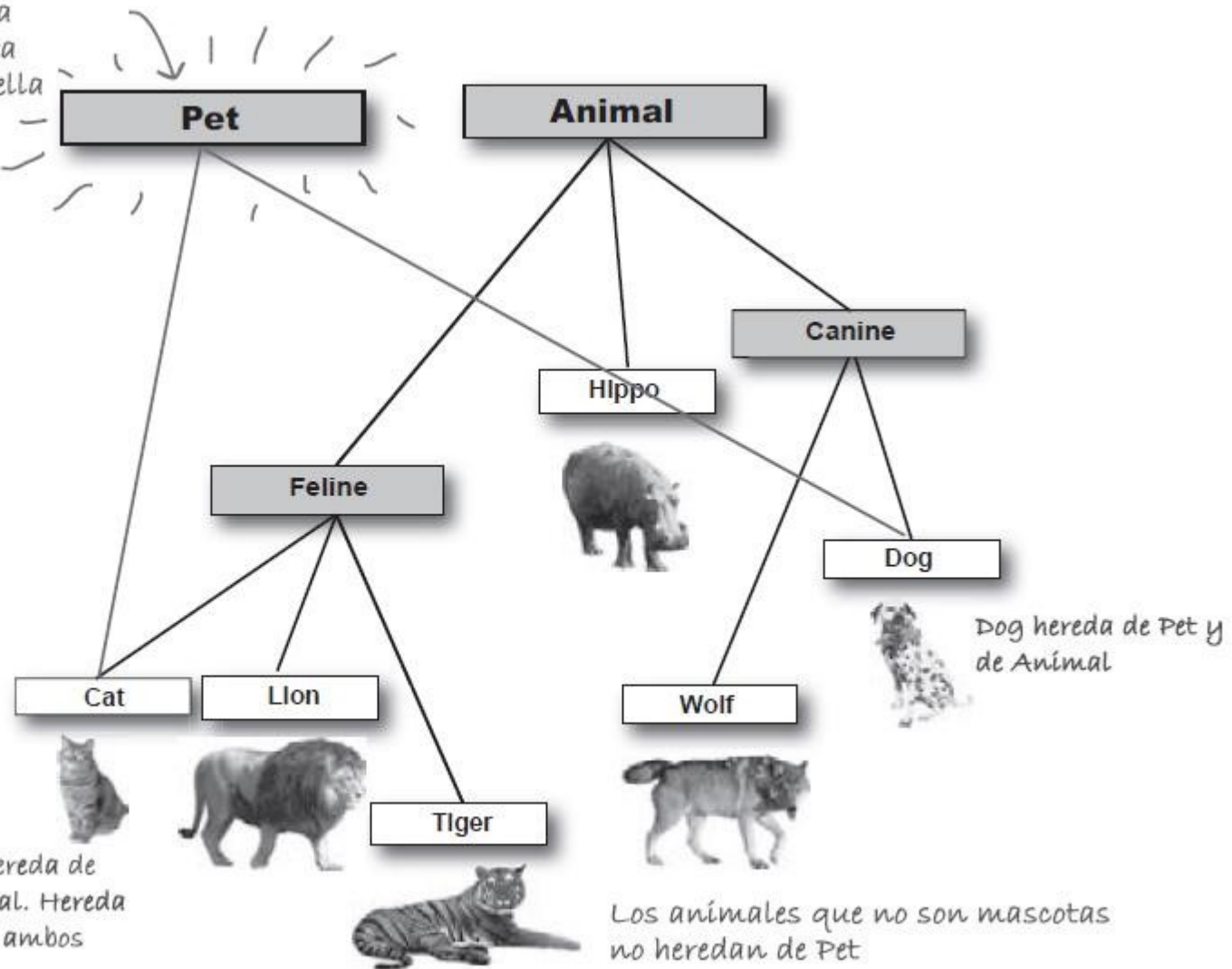
Diseño 3: Métodos en algunas clases

- 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 realmente necesitamos ...

Creamos una nueva superclase abstracta llamada Pet, y en ella definimos los métodos de las mascotas

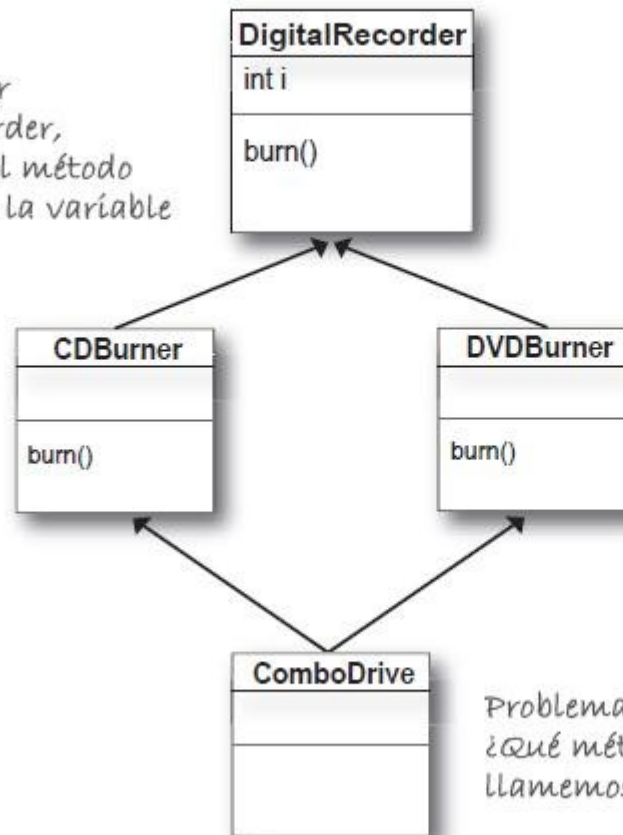


Pero hay un problema ...

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

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

CDBurner y DVDBurner heredan de DigitalRecorder, y ambos sobreescriben el método burn(). Ambos heredan la variable de instancia "i".



Imagina que la variable de instancia "i" es usada por ambos, CDBurber y DVDBurner, con diferentes valores. ¿Qué pasará si ComboDrive necesita usar ambos valores de "i"?

*Problema de la múltiple herencia.
¿Qué método burn() se ejecutará cuando llamemos al método brun() desde ComboDrive()?*

Solución: las interfaces

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

<i>Pet</i>
<u>abstract void beFriendly();</u> <u>abstract void play();</u>

Una interface es como una clase 100% abstracta

Todos los métodos en una interface son abstract. Cualquier clase que SEA una mascota, DEBE implementar (sobreescribir) los métodos de Pet.

Interface: Definir e implementar (I)

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.

interface: Definir e implementar (II)

Escribimos "interface"
en lugar de "class"

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

```
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

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

```
    public void beFriendly() {...}
```

```
    public void play() {...}
```

```
    public void roam() {...}
```

```
    public void eat() {...}
```

```
}
```

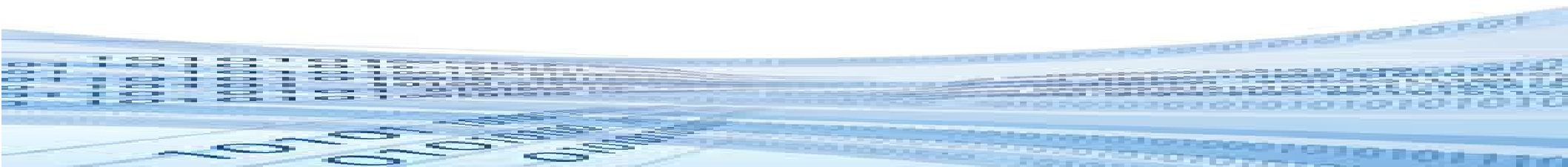
Escribimos "implements"
seguido del nombre de la
interface

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

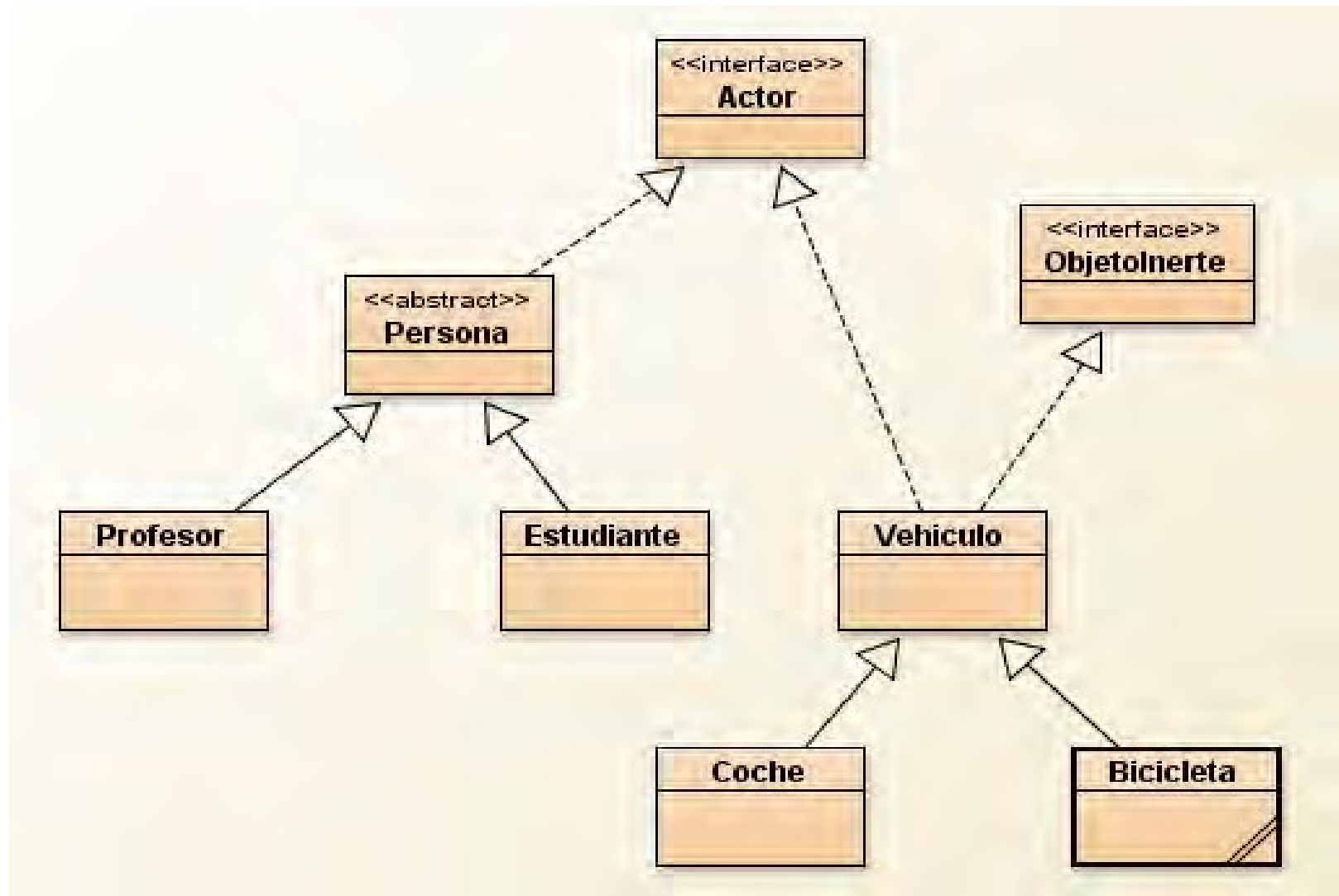
Estos métodos son los métodos normales
sobrescritos heredados de las superclases

Características de las interfaces

- Emplea la palabra reservada ***interface***.
- Todos los métodos son abstract y public.
- 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 (I)



Ejemplo (II)

```
public interface Actor {...}
```

```
public abstract class Persona implements Actor {...}
```

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

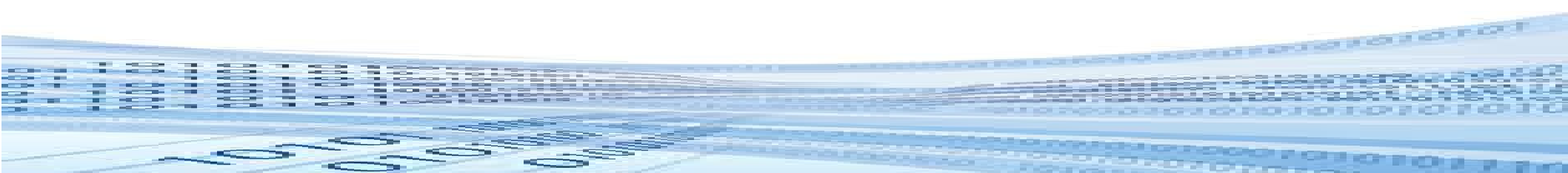
```
public class Estudiante extends Persona {...}
```

```
public interface ObjetoInerte {...}
```

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

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

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

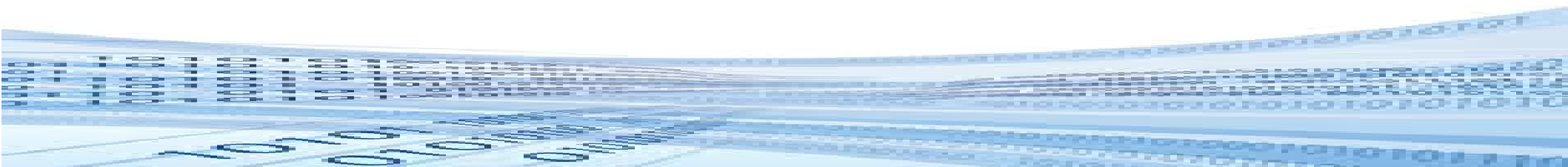


Herencia en las *interfaces*. Ejemplo

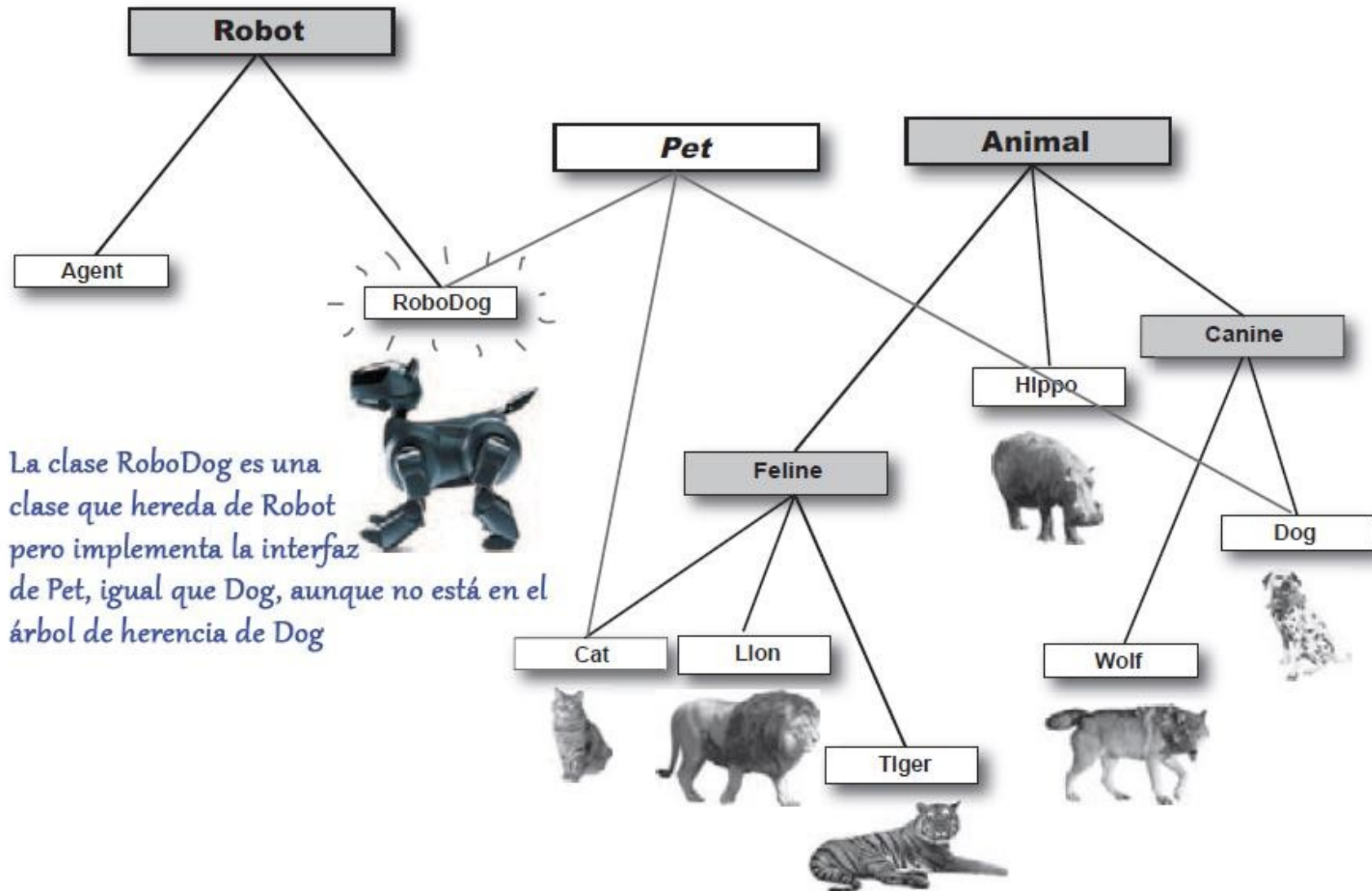
```
interfaz E1 {  
    void metodo1();  
    void metodo2();  
}  
interfaz E2 {  
    void metodo3();  
}
```

```
interfaz 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  
}
```



Clases que pertenecen a diferentes árboles de herencia pueden implementar la misma interfaz



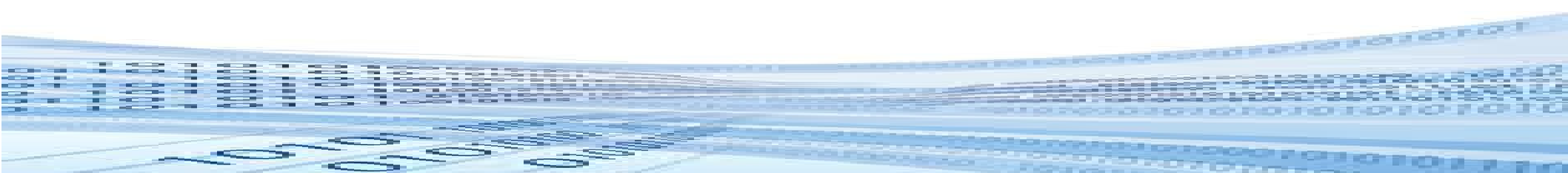
Herencia e interfaces

- Herencia

- Las subclases están en el mismo árbol de herencia.
- Una clase únicamente puede heredar de otra clase.

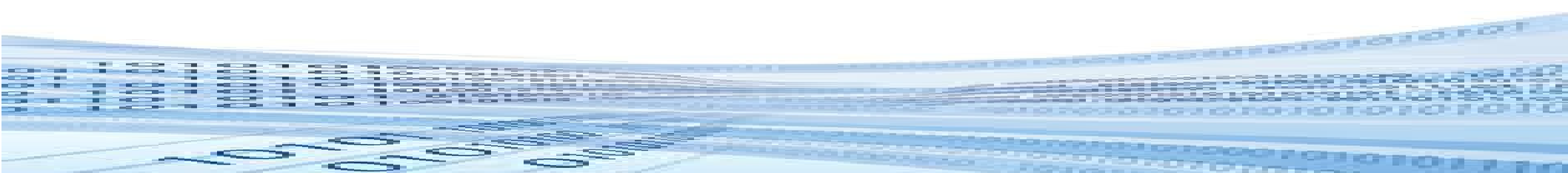
- Interfaces

- No es necesario el mismo árbol de herencia.
- Una clase puede implementar varias interfaces.
- Las interfaces definen los roles de una clase.



Interfaces vs Clase abstracta

- Una interfaz sólo puede tener métodos abstractos. 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 clase abstracta puede proporcionar la implementación de los métodos mientras que una interfaz sólo puede proporcionar la declaración (cabecera) de los métodos.
- 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.).



¿Qué definir en cada situación?

- Crea una clase que no herede de otra cuando no cumpla el test SER.
- Crea una subclase cuando se precise una versión más específica de la clase y se necesite sobrescribir o añadir nuevos comportamientos.
- Utiliza una clase abstracta para definir una "plantilla" para conjunto de subclases. Para garantizar que no se puedan crear objetos de este tipo.

