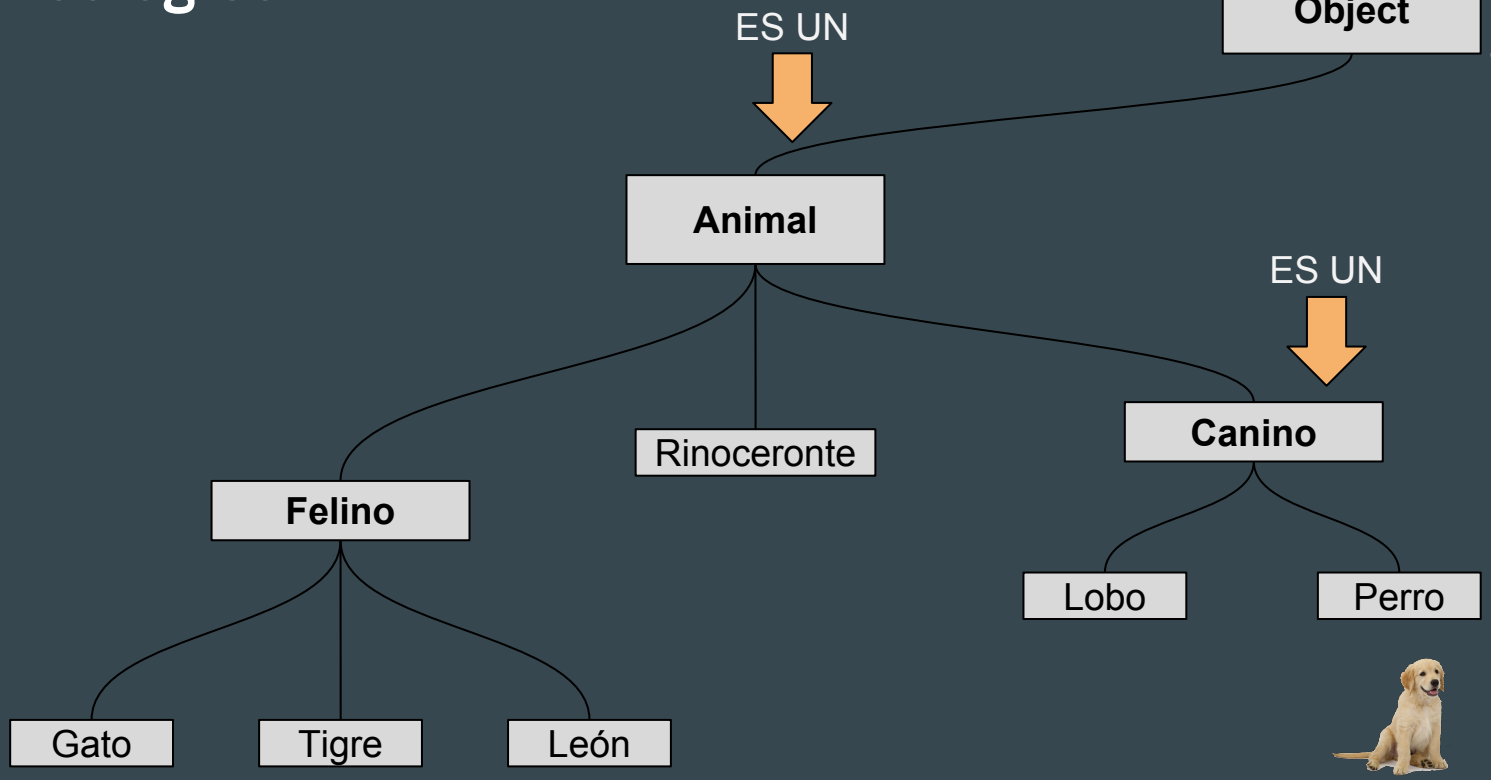
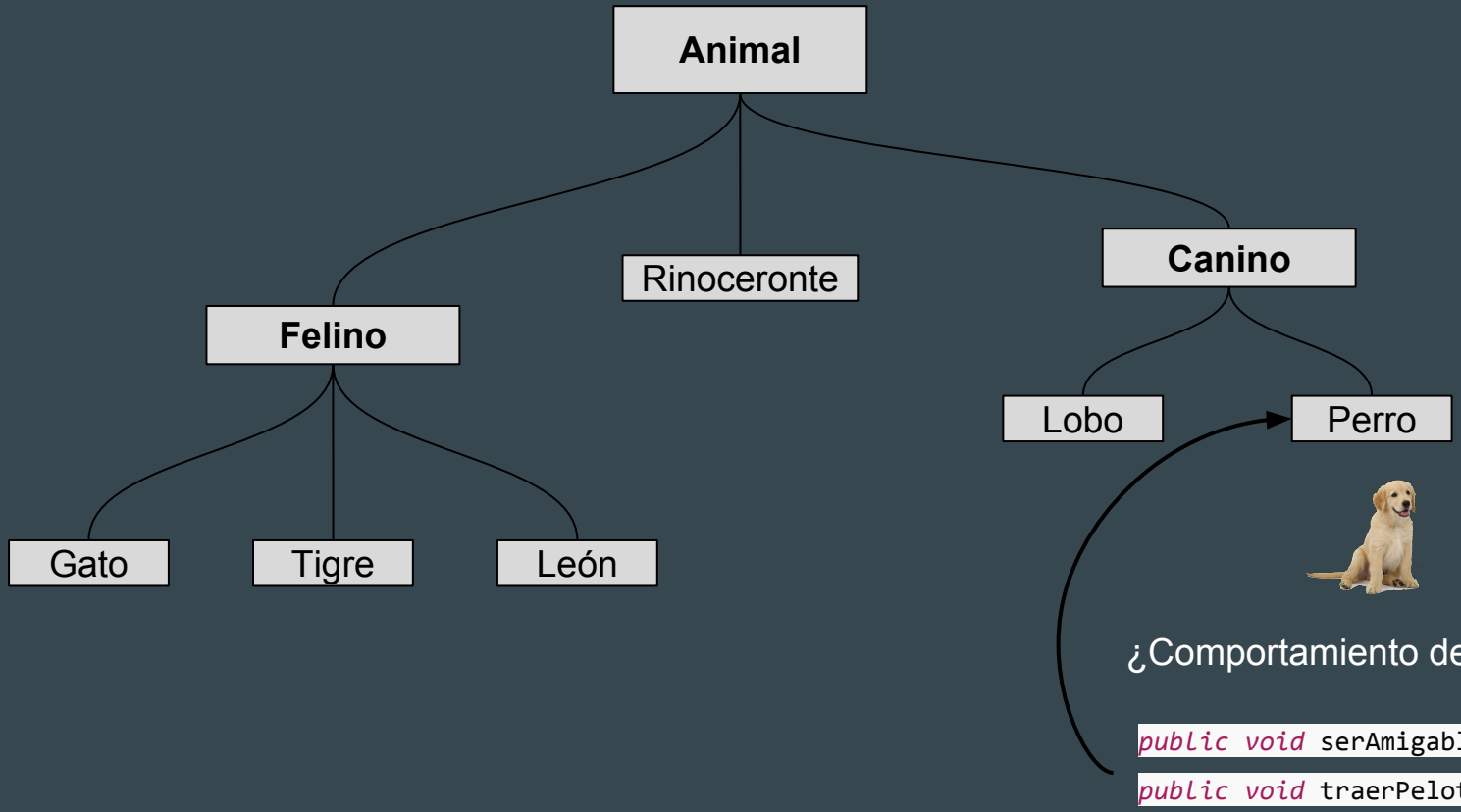
Interfaces

...

UML Zoológico

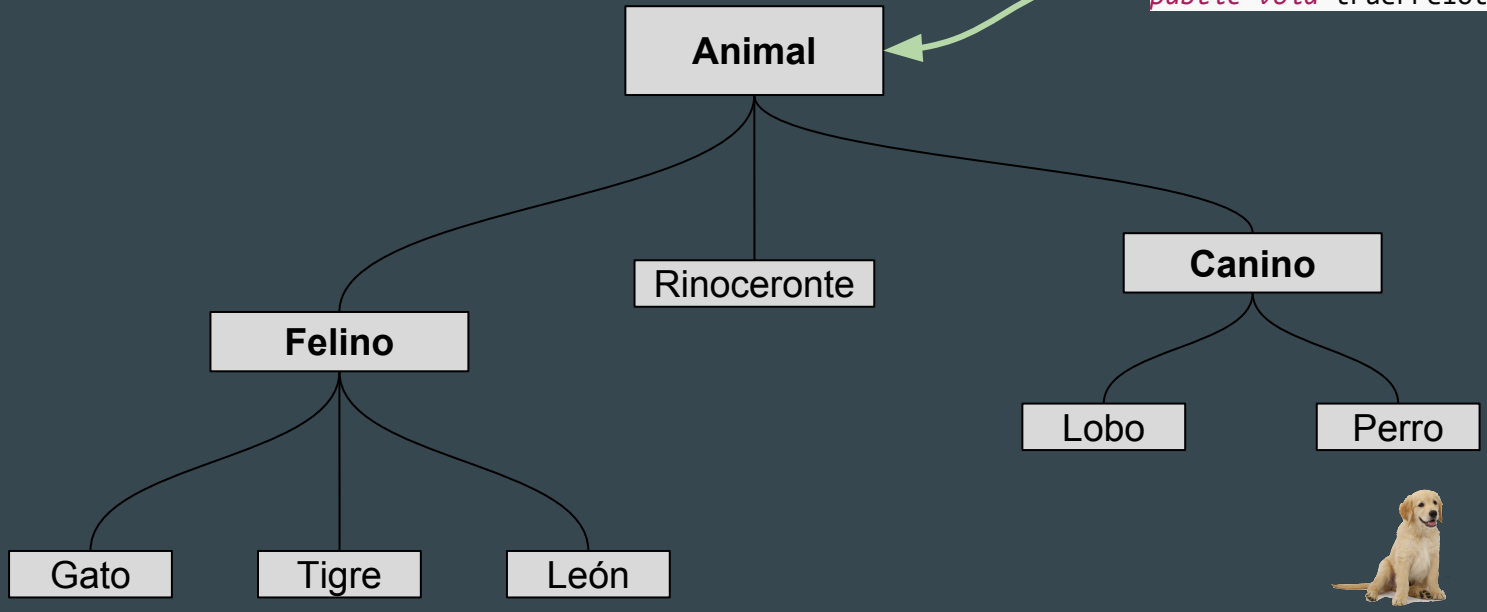


UML Veterinaria



Solución 1

```
public void serAmigable() {...}
public void traerPelota() {...}
```



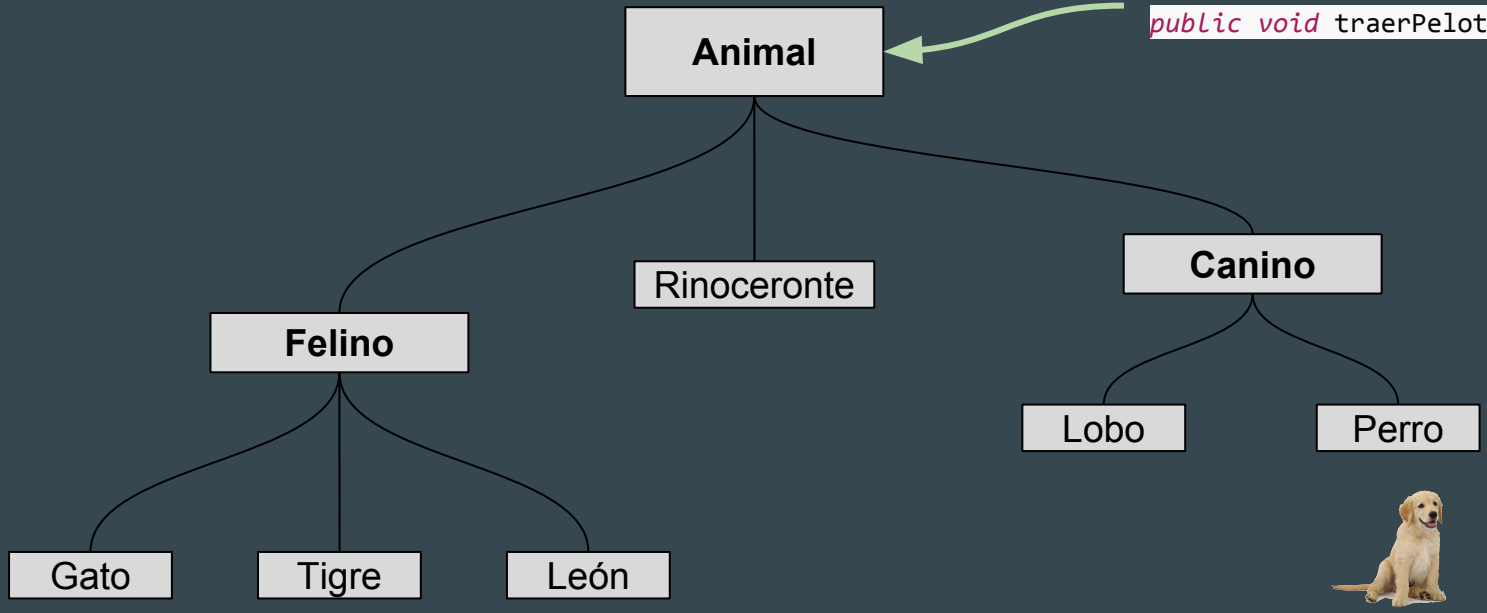
Pros:

- Todos los animales heredan el comportamiento de mascota.
- No hay que tocar el comportamiento de las subclases existentes.
- Las futuras clases implementadas podrán hacer uso de los nuevos métodos.
- Animal puede ser utilizado como tipo polimórfico.

Solución 2

¿Métodos abstractos en la superclase?

```
public void serAmigable() {...}
public void traerPelota() {...}
```

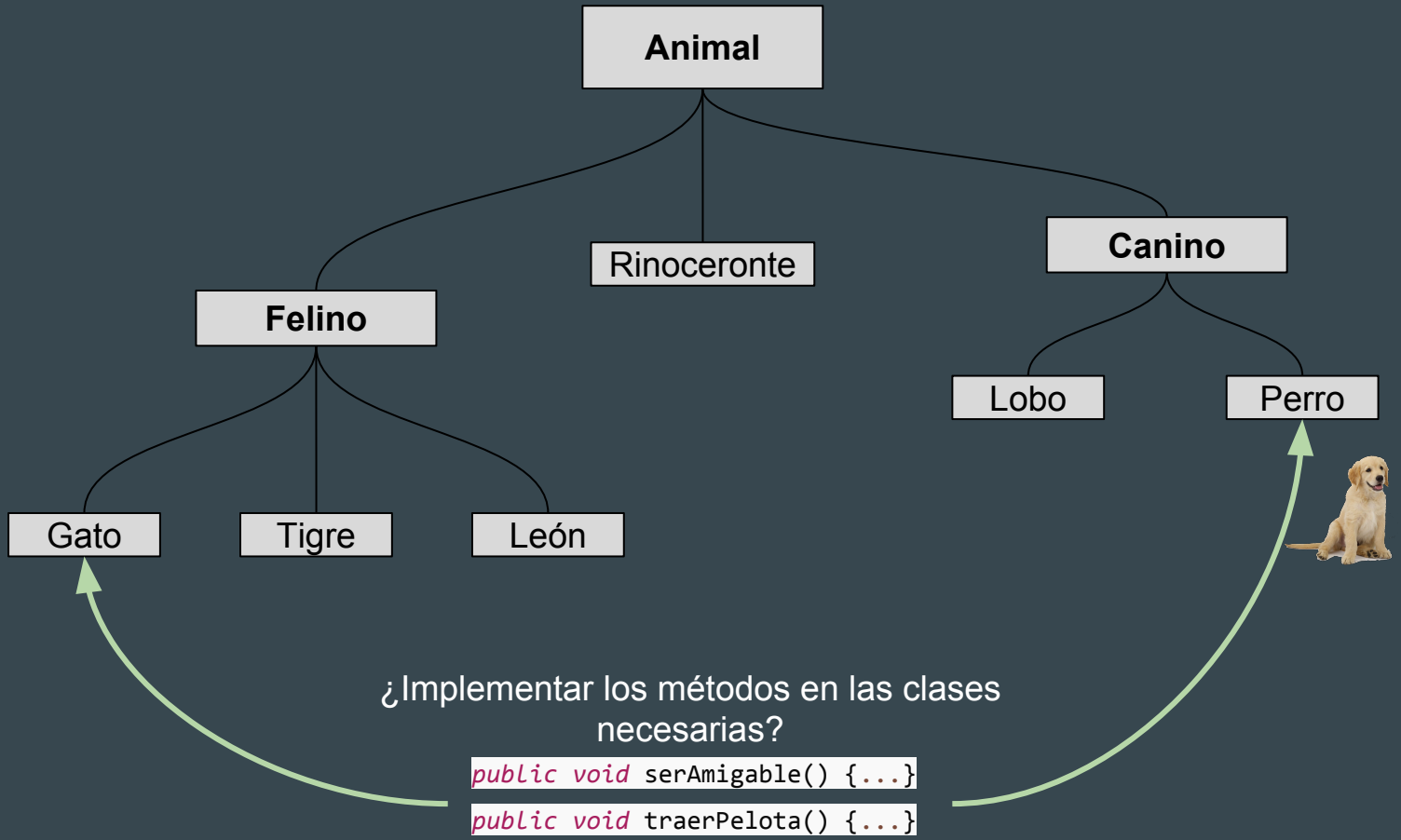


Pros:

- Todos los beneficios de la solución anterior.
- Se pueden implementar métodos que no hagan nada para los animales que no se consideran mascotas.


.- ¿Querés que lleve la pelota?
Tengo el método, sé cómo hacerlo...

Solución 3


Animal
Felino
Rinoceronte
Canino
Lobo
Perro
Gato
Tigre
León
¿Implementar los métodos en las clases necesarias?
public void serAmigable() {...}
public void traerPelota() {...}

Pros:

- No nos tenemos que preocupar por los rinocerontes o los lobos.
- Los métodos están en las clases a las que pertenecen.
- El gato y el perro implementan su comportamiento sin que los demás animales se enteren.

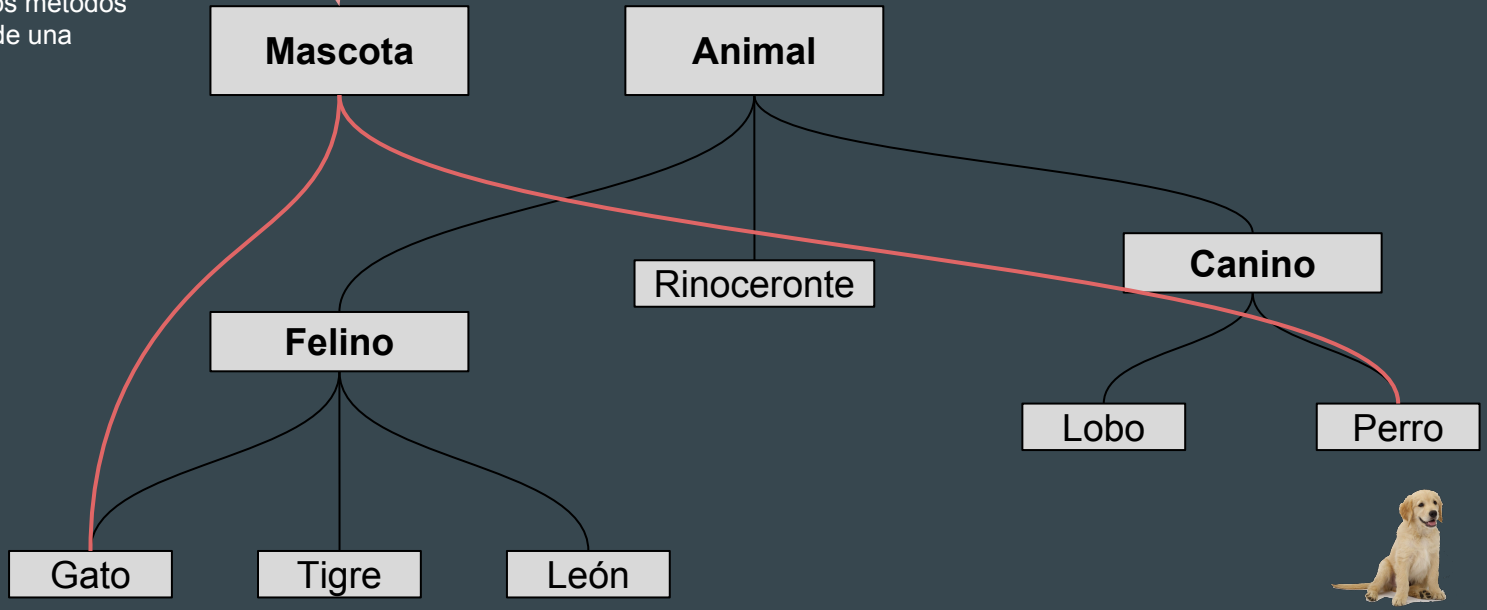
Contras:

- Hay que definir un contrato sobre las acciones que una mascota puede realizar.
- No estamos usando polimorfismo. No podemos usar Animal como tipo polimórfico. Porque no vamos a poder llamar un método de una mascota en un Animal.

¿Y ahora?

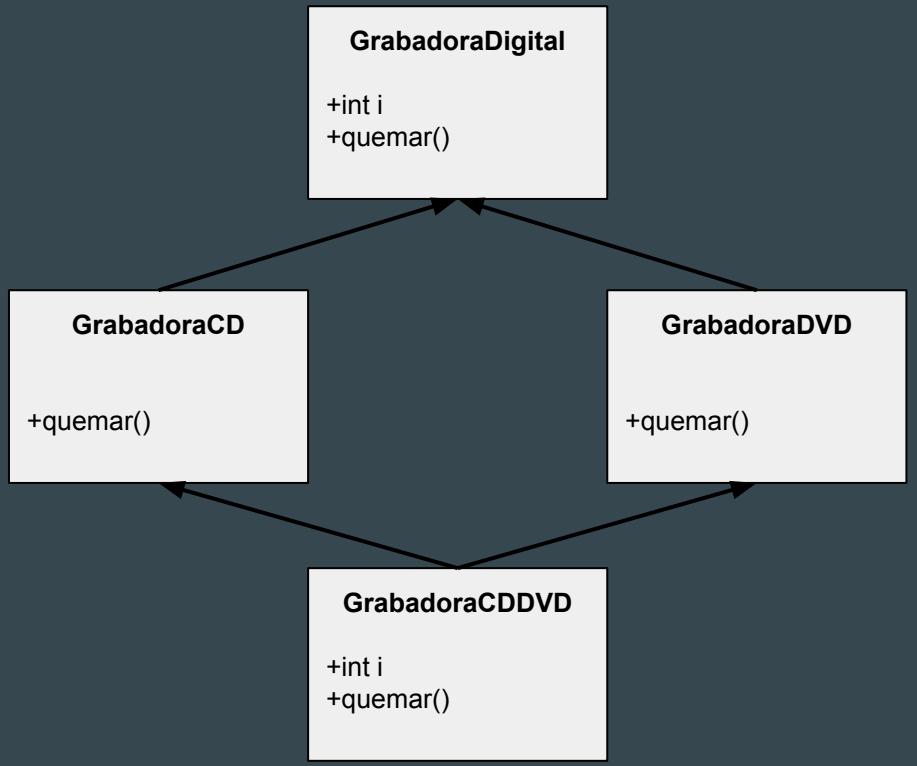
- Una forma de tener el comportamiento de mascotas solo en las clases necesarias.
- Una forma de garantizar que todos los métodos definidos poseen la misma signatura, un “contrato”.
- Una forma de tomar ventaja del polimorfismo.

Una nueva superclase abstracta, Mascota. Con todos los métodos necesarios de una mascota?

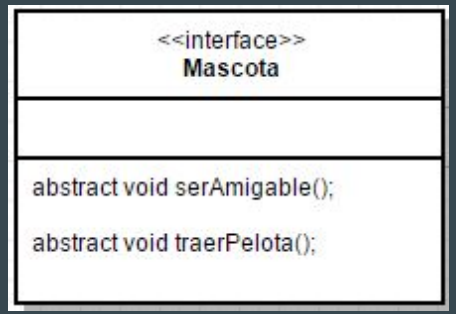


¿Herencia múltiple?

Deadly Diamond of Death (DDD)



Interfaces!!




```java
package com.utn.learning.interfaces;

public interface Mascota {

    abstract void serAmigable();
    abstract void traerPelota();
}
```

Todos los métodos dentro de una interfaz son públicos y abstractos por defecto.

De esta forma la clase que implemente dicha interfaz DEBE implementar los métodos.

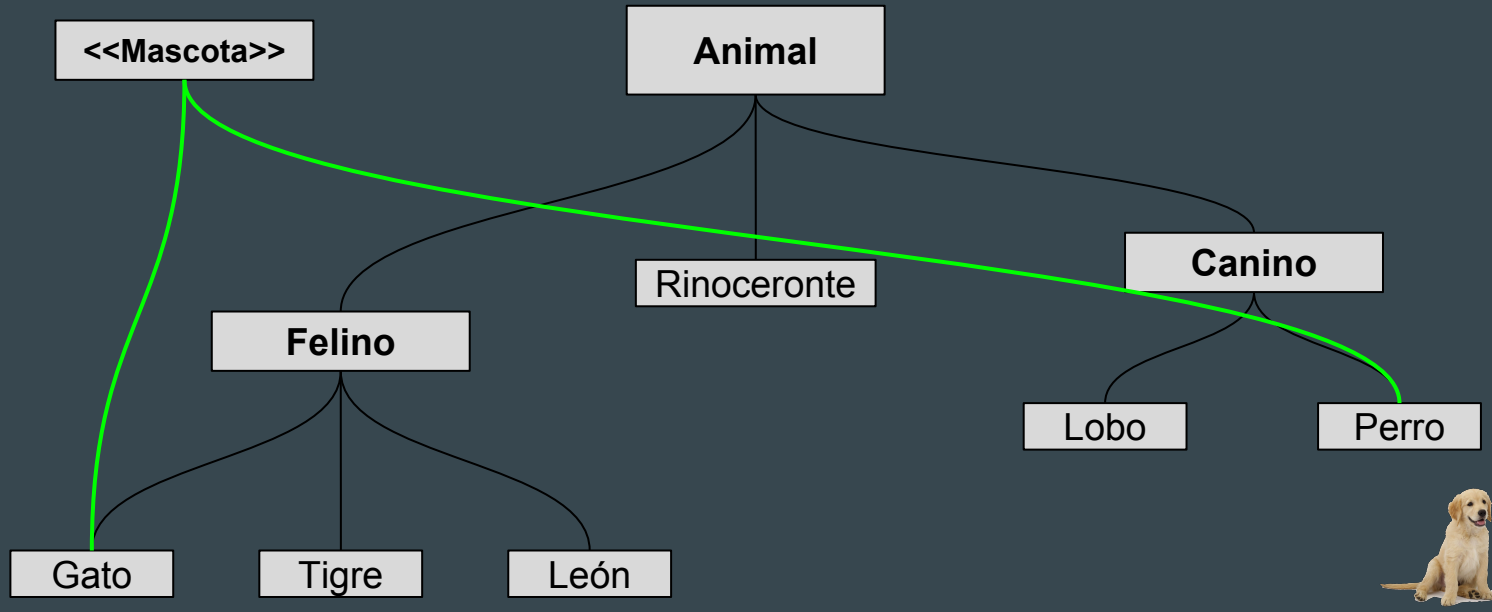
```java
package com.utn.learning.interfaces;

public class Perro extends Canino implements Mascota {

    @Override
    public abstract void serAmigable() {...};

    @Override
    public abstract void traerPelota() {...};
}
```

Diagrama con Interfaz



Consideraciones

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Nuevo requerimiento en DoIt:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);
}
```

Extensión de interfaces:

```
public interface DoItPlus extends DoIt {
   boolean didItWork(int i, double x, String s);
}
```

Método default, java 8 en adelante:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }
}
```

Ejemplo

```java
public interface Comparable {

    // this (el objeto que llama a esMayorQue)
    // y otro deben ser instancias de la misma
    // clase devuelve 1, 0, -1 si this
    // es mayor que, igual a, o menor a otro.

    public int esMayorQue(Comparable otro);
}
```

```java
public Object encontrarMayor(Object o1, Object o2) {
   Comparable obj1 = (Comparable) o1;
   Comparable obj2 = (Comparable) o2;

   if ((obj1).esMayorQue(obj2) > 0)
      return obj1;
   else
      return obj2;
}
```

```java
public class Rectangulo implements Comparable {
    public int base = 0;
    public int altura = 0;

    public int getArea() {
        return base * altura;
    }

    // Método requerido de la interface Comparable
    public int esMayorQue(Relatable otro) {
        Rectangulo otroRect = (Rectangulo) otro;

        if (this.getArea() < otroRect.getArea())
            return -1;
        else if (this.getArea() > otroRect.getArea())
            return 1;
        else
            return 0;
    }
}
```