

## Interfaces

- 1) Saber definir una interfaz
- 2) Saber qué es una interfaz funcional
- 3) Herencia de interfaces
- 4) Implementación de interfaces
- 5) Variables de tipo una interfaz
- 6) Dependencias entre clases

## Clases abstractas y anónimas

- 1) Saber definir clases abstractas
- 2) Herencia de clases abstractas
- 3) Variables de tipo una clase abstracta
- 4) Saber qué es una clase anónima
- 5) Saber definir un clase anónima a partir de una interfaz
- 6) Saber definir una clase anónima a partir de una clase abstracta
- 7) Saber definir una clase anónima a partir de una clase

## Interfaces y clases genéricas

- 1) Saber qué es una interfaz genérica
- 2) Herencia de interfaces genéricas
- 3) Implementar interfaces genéricas
- 4) Saber qué es una clase genérica
- 5) Crear objetos de tipo una clase genérica
- 6) Herencia de clases genéricas

## Abstracción y generalización en UML

- 1) Interfaces en UML
- 2) Herencia de interfaces en UML
- 3) Implementación de interfaces en UML
- 4) Dependencias de clases en UML
- 5) Clases abstractas en UML
- 6) Herencia de clases abstractas en UML

# Interfaces

- 1) Saber definir una interfaz
- 2) Saber qué es una interfaz funcional
- 3) Herencia de interfaces
- 4) Implementación de interfaces
- 5) Variables de tipo una interfaz
- 6) Dependencias entre clases



# Interfaces en Java

Definir constantes comunes para ser utilizadas por múltiples clases

Una interfaz es una caratula de constantes y métodos

```
package clases.interfaces;
public interface Alimentarse
{
    int GRAMOS_POR_KILO = 1000;
    float KILOS_POR_GRAMO = 0.001F;
    Object comer();
    Object comer(float kg);
    Object beber();
    Object beber(float litros);
}
```

De forma implícita las variables definidas tienen los modificadores de **final static public**, es decir, son constates estáticas públicas

De forma implícita los métodos definidos tienen el modificador de **abstract public**, es decir, son métodos de objeto públicos abstractos

Tales métodos se denominan **métodos abstractos**

[Interfaces en UML](#)

# Métodos de una interfaz

En una interfaz se pueden definir los siguientes tipos de métodos:

- **Métodos abstractos**
- **Métodos predeterminados**
- **Métodos estáticos**
- **Métodos privados de objeto**

# Interfaces en Java 8

En una interfaz se pueden añadir métodos con código (métodos predeterminados)

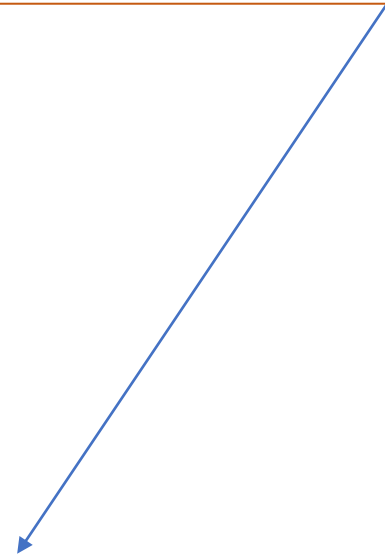
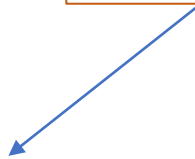
```
package clases.interfaces;  
public interface Alimentarse  
{
```

```
...
```

```
default String gordo(){  
    return "Estás gordo";  
}
```

```
default Object alimentarse(float kg, float litros){  
    comer(kg);  
    beber(litros);  
    return this;  
}  
}
```

Son métodos de objetos públicos con código



# Interfaces en Java 8

En una interfaz se pueden añadir métodos estáticos

```
package clases.interfaces;  
public interface Alimentarse  
{
```

...

```
    static float cantidadEnGramos(float kg){  
        return kg*GRAMOS_POR_KILO;  
    }
```

```
    static float cantidadEnKilogramos(float gramos){  
        return gramos*KILOS_POR_GRAMO;  
    }  
}
```

Se puede acceder en una aplicación a los miembros estáticos definidos en una interfaz del siguiente modo:


```
float a = Alimentarse.cantidadEnGramos(200);  
int b = Alimentarse.GRAMOS_POR_KILO;
```

# Interfaces en Java 9

En una interfaz se pueden definir métodos privados de objeto

```
package clases.interfaces;  
public interface Alimentarse  
{  
    ...  
    private void saludar(String mensaje)  
    {  
        System.out.println(mensaje);  
    }  
}
```

Los métodos privados sólo pueden invocarse en métodos predeterminados de la interfaz, permitiendo la reutilización de código.



# Interfaces funcionales

Una interfaz funcional es aquella que sólo tiene una caratula de un método (puede tener constantes estáticas, métodos predeterminados, estáticos y privados)

```
package clases.interfaces;
```

```
@FunctionalInterface
```

```
public interface Copiable
```

```
{
```

```
    Object copiar(Object o);
```

```
    default void alimentarse(float kg){
```

```
        System.out.println("Has comido "+kg+" kilos");
```

```
    }
```

```
}
```

Anotación para indicar que la interfaz sólo tiene un método sin implementar

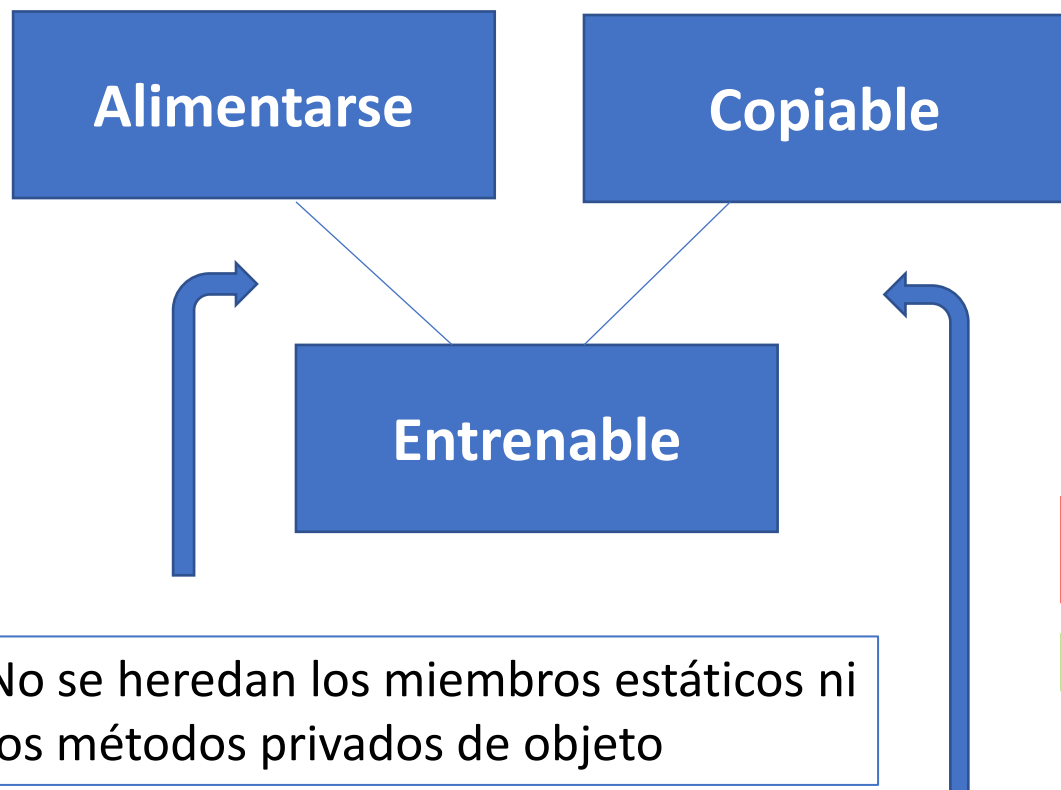
Se pueden definir uno o mas métodos predeterminados , estáticos y privados





# Herencia de interfaces

Las interfaces permiten la herencia múltiple.



No se heredan los miembros estáticos ni los métodos privados de objeto

No se pueden heredar de interfaces que tengan un método predeterminado con el mismo nombre y parámetros

```

package classes.interfaces;
public interface Entrenable
extends Alimentarse, Copiable
{
    void entrenamiento(int tiempo);
    void lesionarse(int tiempo);
}
  
```

Métodos predeterminados heredados

Métodos abstractos heredados

Object alimentarse(float kg, float litros)

default String gordo()

void alimentarse(float kg)

Object comer();

Object comer(float kg);

Object beber();

Object beber(float litros);

Object copiar(Object o);

# Implementación de interfaces

Una clase puede implementar una o más interfaces.

La implementación de una interfaz en una clase consiste en definir el cuerpo de todos los métodos abstractos.

...

```
import clases.interfaces.Alimentarse;
```

```
import clases.interfaces.Carrera;
```

```
public class Persona implements Alimentarse, Carrera
```

```
{
```

```
    ...
```

```
}
```

Hay que definir cada uno de los siguientes métodos anteponiendo la palabra reservada de **public**

- Object comer()      Object comer(float kg)
- Object beber()      Object beber(float litros)
- Object correr()      Object correr(float km)

En todos los métodos se puede devolver Persona en vez de Object



# Variables de tipo interfaz

Se pueden declarar variables, atributos y parámetros de tipo una interfaz. Los valores que se pueden asignar a tales elementos son el valor “null” o cualquier objeto que implemente dicha interfaz.

```
Persona p = new Persona("PEDRO",34,1.8,67);  
Alimentarse ali = p;
```

Desde la variable “ali” sólo se pueden acceder a los métodos de la interfaz  
**ali.comer();**

```
if(ali instanceof Persona)  
    p = (Persona) ali;
```

Operador de  
conversión

Con el operador “instanceof” se puede comprobar si una variable interfaz apunta a un determinado objeto, y convertir dicha variable a tal objeto

```
if(p instanceof Alimentarse)  
    ...
```

Con el operador “instanceof” se puede comprobar si un objeto implementa una interfaz



# Dependencia entre clases

En la definición de la clase “Persona” hay un atributo de objeto “Mascota”, de manera que si no se ha definido la clase “Mascota”, no se puede definir la clase “Persona”. Como consecuencia, la clase “Persona” depende de la existencia de la clase “Mascota”



Definir el atributo de tipo objeto “Mascota” como una interfaz “IMascota” en la que se tuvieran declarados los métodos, sin implementar, de la clase “Mascota”

Atributo  
IMascota  
en Persona



```
private IMascota mascota;
```

```
public IMascota getMascota(){  
    return mascota.clone();  
}
```

```
public Persona setMascota(IMascota mascota)  
{  
    if(mascota==null)this.mascota = null;  
    else this.mascota = (IMascota) mascota.clone();  
    return this;  
}
```



Al invocar el método setMascota se le pasará un objeto que implemente IMascota

# Clases abstractas y anónimas

- 1) Saber definir clases abstractas
- 2) Herencia de clases abstractas
- 3) Variables de tipo una clase abstracta
- 4) Saber qué es una clase anónima
- 5) Saber definir una clase anónima a partir de una interfaz
- 6) Saber definir una clase anónima a partir de una clase abstracta
- 7) Saber definir una clase anónima a partir de una clase

# Clases abstractas



Una clase abstracta es una clase en la que algunos de sus métodos sólo tienen la firma. Dichos métodos serán de objeto, públicos o protected y no finales, y se les anteponen la palabra reservada “abstract”, al igual que a la clase.

```
package clases.abstractas;  
abstract public class Animal
```

```
{  
    abstract public Animal comer();  
    abstract public Animal comer(int kilos);  
    abstract protected int setNumeroDePatas();  
}
```

Se definen los atributos y métodos no abstractos  
Los constructores se definirán como **protected** ya que no es posible crear objetos de tipo una clase abstracta

El método “setNumeroDePatas” es abstracto ya que se desconoce el tipo del animal y no se sabe si el animal tiene 0, 2 o 4 patas. También dependiendo del tipo de animal, el aumento de peso por la cantidad de comida varía de un animal a otro

# Herencia de clases abstractas



Una clase se puede derivar de una clase abstracta.

- Una clase que herede de una clase abstracta e implemente todos sus métodos abstractos dejará de ser abstracta.
- Si sólo implementa algunos métodos abstractos, entonces seguirá siendo abstracta y deberá declarar los métodos abstractos no implementados.
- En una clase que derive de una clase abstracta, puede tener nuevos métodos abstractos.

```
package clases;
```

```
public abstract class Cuadrupedo extends Animal
```

```
{
```

Los cnstructores siguen siendo protegidos

```
...
```

En vez de Animal se puede poner Cuadrupedo

```
abstract public Cuadrupedo comer();
```

```
abstract public Cuadrupedo comer(int kilos);
```

```
abstract public Cuadrupedo clone();
```

```
}
```

Se implementa uno de los tres métodos abstractos de Animal

Se ha añadido un nuevo método abstracto

# Variables de tipo una clase abstracta

No es posible crear objetos de una clase abstracta, pero si declarar variables de tipo una clase abstracta y asignarles un objeto de una clase que implemente todos los métodos abstractos de dicha clase abstracta

```
Animal p = new Perro("Leon",20F);  
Cuadrupedo p1 = new Perro("Leon",20F);
```

Desde la variable “p” sólo se puede acceder a los miembros “públicos” definidos en la clase “Animal”

Desde la variable “p1” sólo se puede acceder a los miembros “públicos” definidos en la clase “Cuadrupedo”

```
//pe de tipo Perro  
if(p instanceof Perro)  
    pe = (Perro) p;
```

Con el operador “instanceof” se puede comprobar si un objeto de una clase abstracta almacena un objeto deriva dicha clase

Operador de conversión



# Clases anónimas

Una clase anónima es una clase que no tiene nombre. Las clases anónimas sólo podrán reemplazar los métodos de la clase que deriva (que puede ser abstracta) o de la interfaz que implementa.

- Clases anónimas que implementan una interfaz
- Clases anónimas que derivan de una clase abstracta
- Clases anónimas que derivan de una clase padre

# Clases anónimas que implementan una interfaz



## Parámetro de tipo ICoche

```
p.setCoche(new ICoche)
{
}
});
```

Se implementan todos los métodos de la interfaz "Icoche"

## Variable de tipo ICoche

```
ICoche co = new ICoche()
{
}
};
```

//p y p1 de tipo Coche

```
p.setCoche(co);
p1.setCoche(co);
```

Es posible definir varios objetos de tipo dicha clase anónima

# Clases anónimas que derivan de una clase abstracta



## Parámetro de tipo Cuadrupedo

```
p.setPerro(new Cuadrupedo()  
{  
  
});
```

Se reemplazan todos los métodos abstractos de “Cuadrupedo” y , opcionalmente, se sobrescriben métodos que ya están definidos en “Cuadrupedo”

## Variable de tipo Cuadrupedo

```
Cuadrupedo cu = new Cuadrupedo()  
{  
  
};
```

//p y p1 es de tipo Perro

```
p.setPerro(cu);  
p1.setPerro(cu);
```

Es posible definir varios objetos de tipo dicha clase anónima

# Clases anónimas que derivan de una clase



## Parámetro de tipo Mascota

```
p.setMascota(new Mascota("peque"))  
{  
  
});
```

Se sobreescriben métodos que ya están definidos en "Mascota"

## Variable de tipo Mascota

```
Mascota ma = new Mascota("peque")  
{  
  
};
```

//p y p1 de tipo Persona

```
p.setMascota(ma);  
p1.setMascota(ma);
```

Es posible definir varios objetos de tipo dicha clase anónima

# Interfaces y clases genéricas

- 1) Saber qué es una interfaz genérica
- 2) Herencia de interfaces genéricas
- 3) Implementar interfaces genéricas
- 4) Saber qué es una clase genérica
- 5) Crear objetos de tipo una clase genérica
- 6) Herencia de clases genéricas

# Interfaz genérica

Una interfaz genérica es una interfaz que tiene un tipo no definido como parámetro de un método.

```
package clases.interfaces.genericas;  
public interface Acumutable<T>  
{  
    double PI = 3.1416;  
  
    T acumular();  
    T acumular(T o);  
    boolean esNulo();  
}
```

```
package clases.interfaces.genericas;  
public interface Calculo<T>  
{  
    T sumar(T o);  
    T restar(T o);  
}
```

Después del nombre de la interfaz, se escribe entre "< >" y separados por comas los tipos no definidos T, S, ...

# Herencia de interfaces genéricas

Se pueden crear interfaces genéricas que deriven de otras interfaces genéricas

```
package clases.interfaces.genericas;  
public interface Aritmetica<T> extends Acumulable<T>, Calculo<T>  
{  
    T multiplicar(T o);  
}
```

# Implementación de interfaces genéricas



Cuando una clase implementa una interfaz debe indicarse los tipos concretos

```
public class Figura implements Acumulable<Figura> {
```

```
}
```

Los parámetros dejan de ser de tipo Object, para ser de un tipo concreto

```
@Override  
boolean esNulo()  
{  
}
```

```
@Override  
Figura acumular()  
{  
}
```

```
@Override  
Figura acumular(Figura o)  
{  
}
```

También para “sumar”,  
“restar” y “multiplicar”



# Clases genéricas

Una clase genérica es una clase que al menos tiene un atributo de tipo no definido

```
package clases.genericas;
```

```
public class Intercambiar<T>
```

```
{  
    Constructor  
    public Intercambiar(T primero, T segundo)  
{
```

```
    this.primer = primero;
```

```
    this.segundo = segundo;
```

```
}  
Métodos get
```

```
public T getSegundo()  
{
```

```
    return segundo;  
}
```

```
public T getPrimero()  
{
```

```
    return primero;  
}
```

Después del nombre de la clase, se escribe entre "< >" y separados por comas los tipos no definidos T, S, ...

## Atributos

```
private T primero;
```

```
private T segundo;
```

## Método de cambio de estado

```
public void intercambiar()  
{
```

```
    T aux = primero;
```

```
    primero = segundo;
```

```
    segundo = aux;
```

```
}
```

# Clases genéricas



Hágase un programa que cree un objeto de tipo Intercambiar Personas. Se intercambiarán los dos objetos personas y se mostrarán los valores del nuevo primero y nuevo segundo.

```
Intercambiar<Persona> in = new Intercambiar<Persona>(  
    new Persona("PEDRO",23,1.5,45),  
    new Persona("JUAN",21,1.7,55)  
);  
in.intercambiar();  
System.out.println(in.getPrimero().toString());  
System.out.println(in.getSegundo().toString());
```

Salida  
consola



```
Nombre: JUAN  
Edad: 21  
Altura: 1.7  
Peso: 55.0  
Nombre: PEDRO  
Edad: 23  
Altura: 1.5  
Peso: 45.0
```

# Herencia de clases genéricas

Una clase se puede derivar de una clase genérica

```
package clases;
import clases.genericas.Intercambiar;
public class IntercambiarPersona extends Intercambiar<Persona>
{
    public IntercambiarPersona(Persona primero,Persona segundo)
    {
        super(primer,segundo);
    }
}
```

# Herencia de clases abstractas II

Resuélvase el ejercicio anterior utilizando la clase IntercambiarPersona

```
IntercambiarPersona in = new IntercambiarPersona(  
    new Persona("PEDRO",23,1.5,45),  
    new Persona("JUAN",21,1.7,55)  
);  
in.intercambiar();  
System.out.println(in.getPrimero().toString());  
System.out.println(in.getSegundo().toString());
```

Salida  
consola

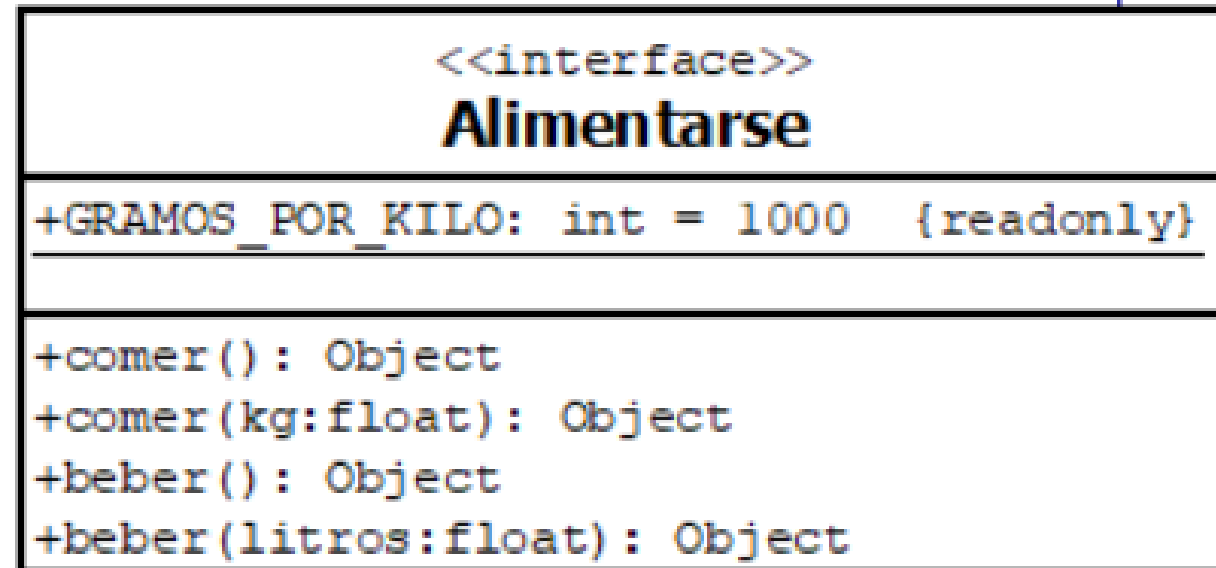
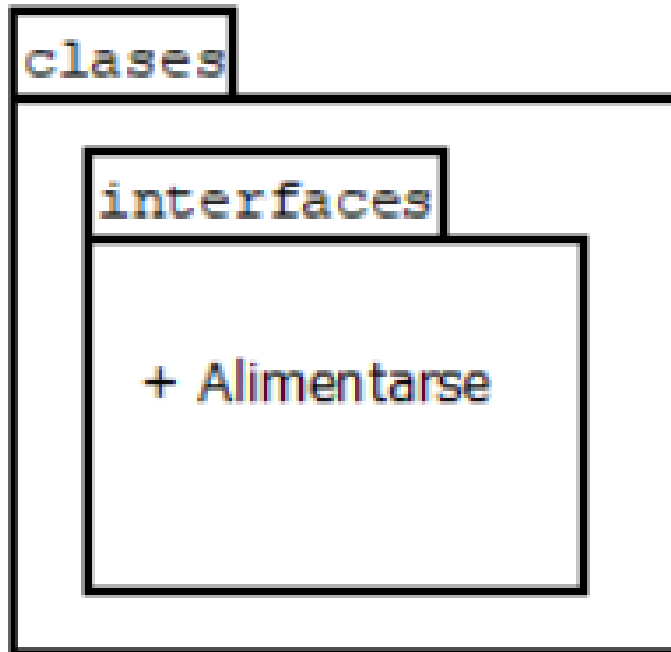


```
Nombre: JUAN  
Edad: 21  
Altura: 1.7  
Peso: 55.0  
Nombre: PEDRO  
Edad: 23  
Altura: 1.5  
Peso: 45.0
```

# Abstracción y generalización en UML

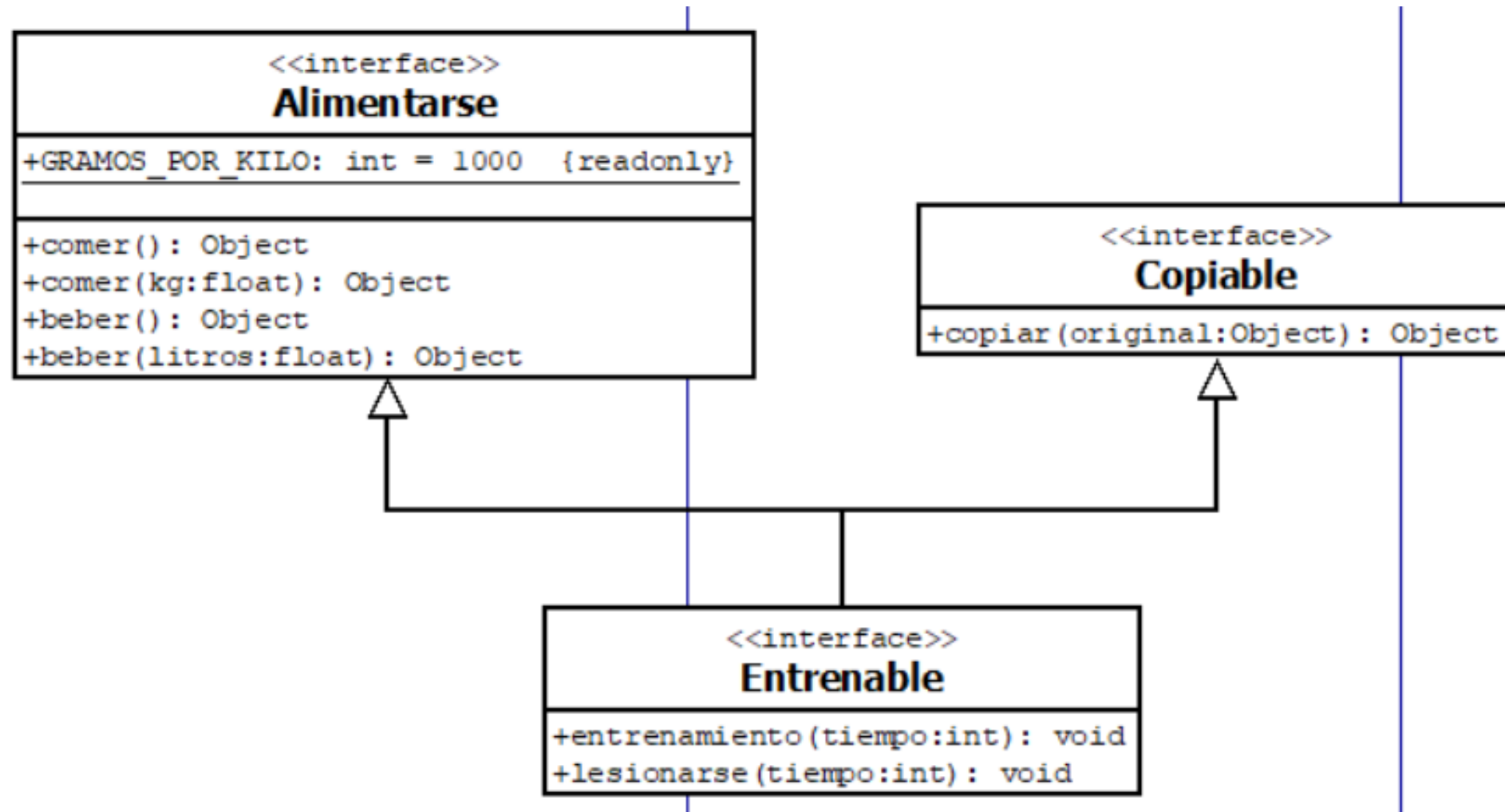
- 1) Interfaces en UML
- 2) Herencia de interfaces en UML
- 3) Implementación de interfaces en UML
- 4) Dependencias de clases en UML
- 5) Clases abstractas en UML
- 6) Herencia de clases abstractas en UML

# Interfaces en UML

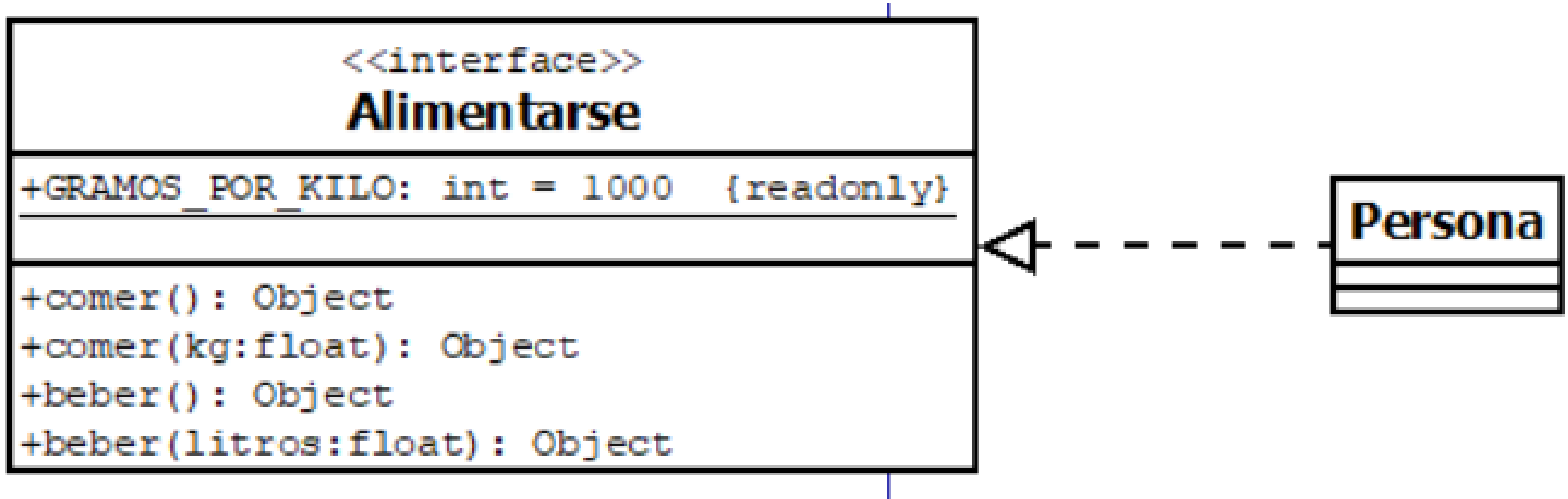


# Herencia de interfaces en UML

[Herencia de interfaces](#)

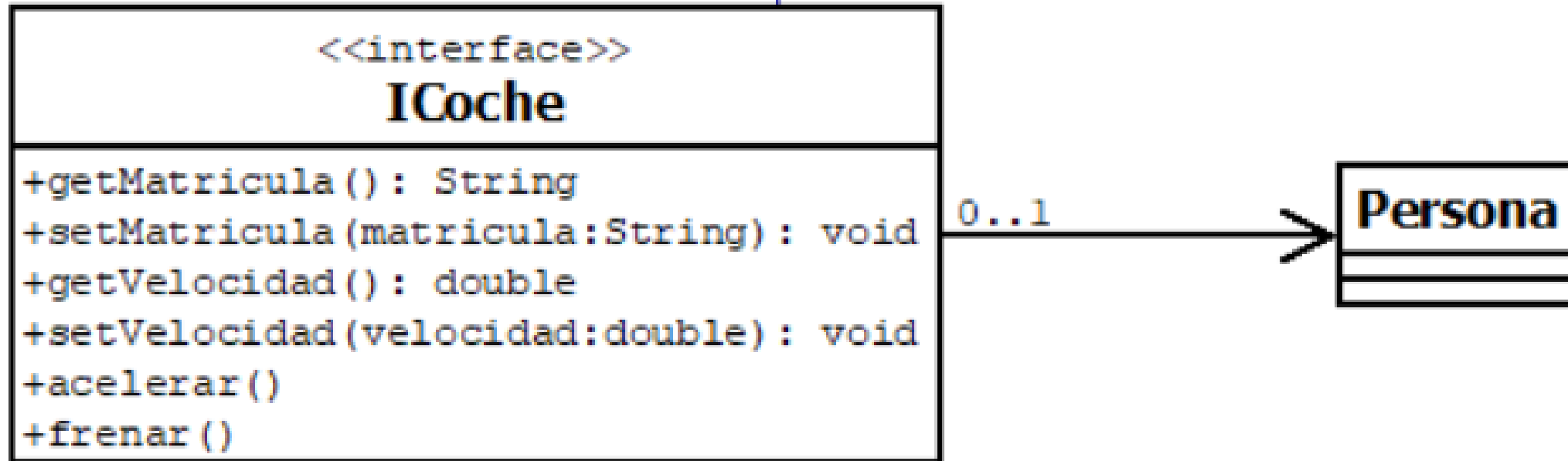


# Implementación de interfaces en UML



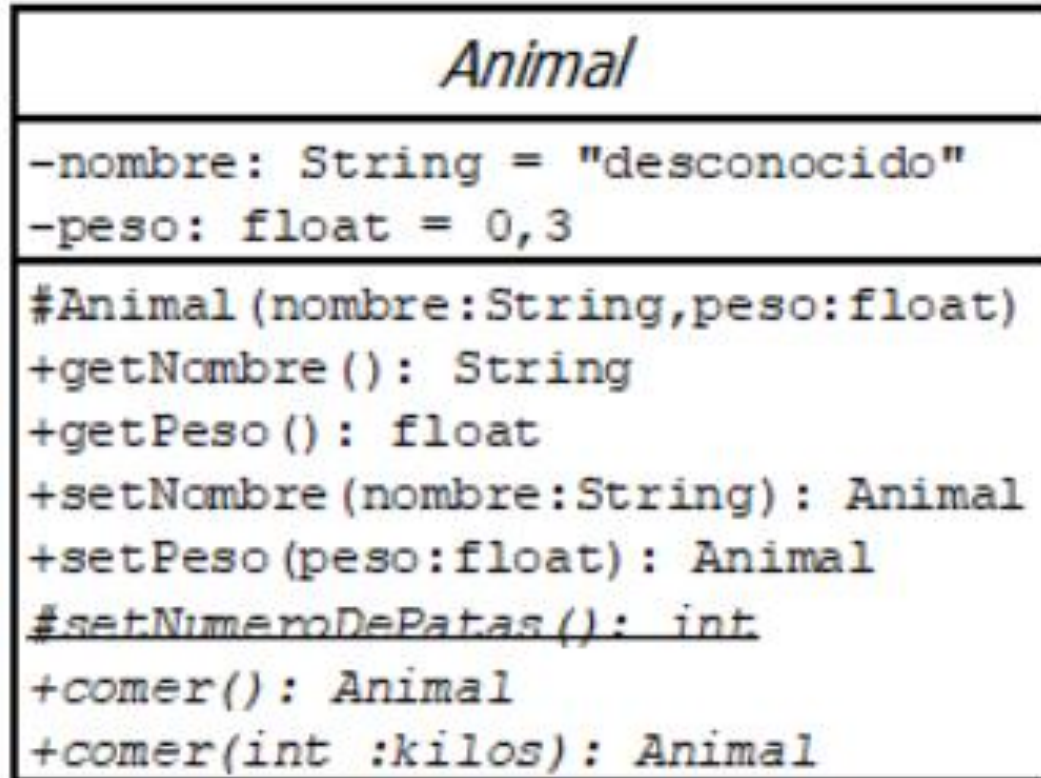


# Dependencias entre clases

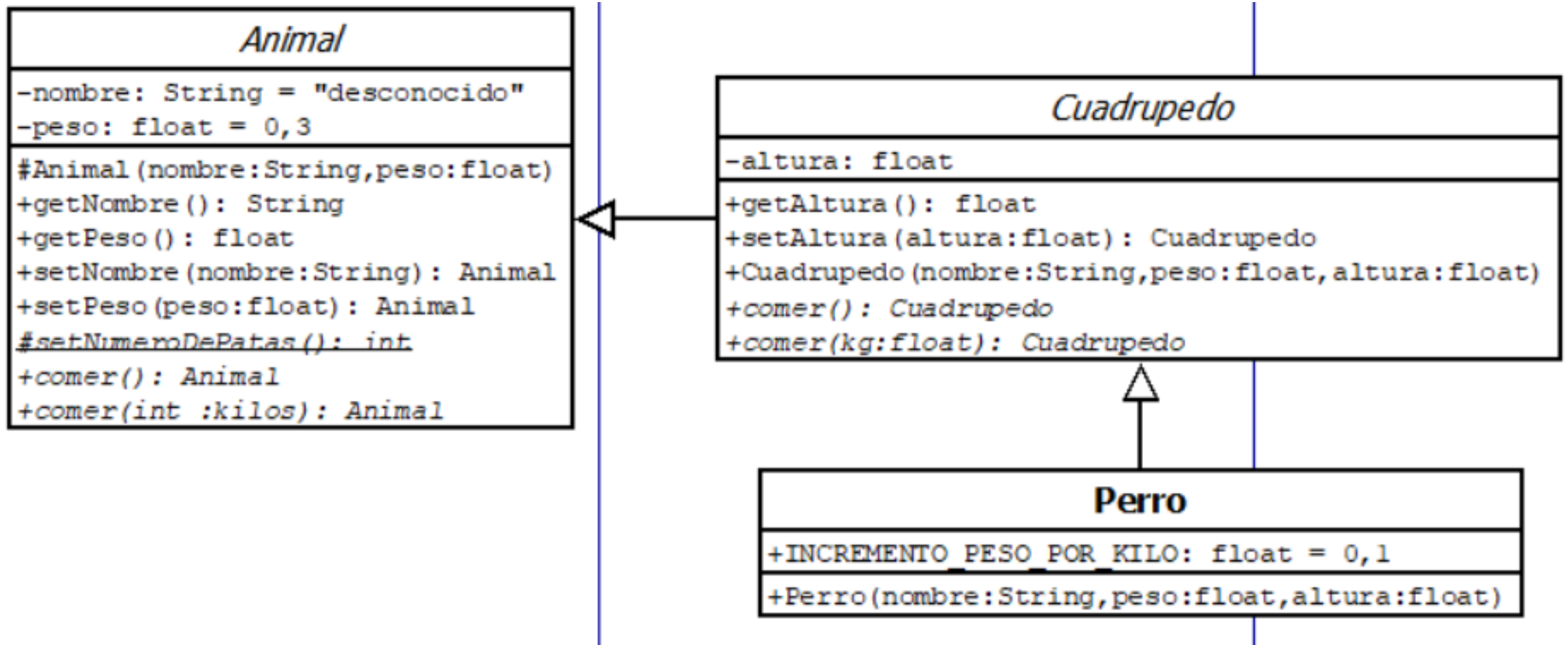


# Clases abstractas en UML

[Clases abstractas](#)



# Herencia de clases abstractas en UML



[Herencia de clases abstractas](#)