



# Java Básico

---

## Polimorfismo

## Copyright

- Copyright (c) 2004  
José M. Ordax
- Este documento puede ser distribuido solo bajo los términos y condiciones de la Licencia de Documentación de javaHispano v1.0 o posterior.
- La última versión se encuentra en  
<http://www.javahispano.org/licencias/>

# Herencia

Mediante la herencia garantizábamos que todas las subclases de una superclase concreta tienen todos los métodos que tiene dicha superclase.

Es decir, definimos una especie de interfaz (API) para un grupo de clases relacionados mediante la herencia.

Ejemplo:

Animal
■ foto: String
■ tipo_comida: String
■ localizacion: String
■ tamaño: String
● hacerRuido()
● comer()
● dormir()
● rugir()

Aquí estamos diciendo que cualquier Animal puede hacer esas cuatro cosas (incluyendo los parámetros y tipos de retorno). Cualquier Animal significa cualquier clase que en la jerarquía de clases herede de Animal.

# Polimorfismo

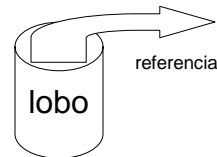
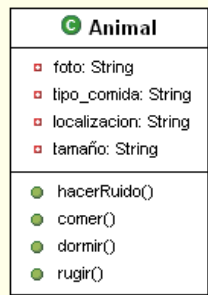
Es otro de los paradigmas de la Orientación a Objetos.

Consiste en que una vez se ha definido una superclase para un grupo de subclases, cualquier instancia de esas subclases puede ser usada en el lugar de la superclase.

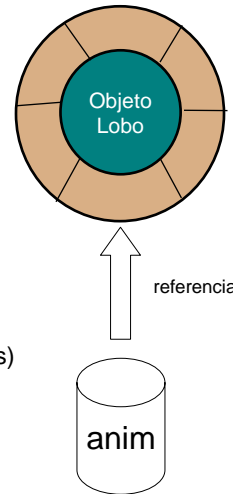
Significa que podemos referenciar un objeto de una subclase mediante una referencia declarada como una de sus superclases.

Object o = new String("Hola");

## Ejemplo



```
public class Test
{
    public static void main(String[] args)
    {
        Lobo lobo = new Lobo();
        Animal anim = lobo;
    }
}
```



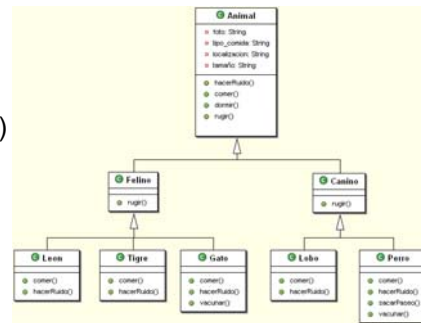
## Polimorfismo

- Por tanto mediante el polimorfismo podemos asignar a una referencia de un tipo superior en la jerarquía de herencia, una instancia de un tipo inferior (que herede).
- Ahora bien, que la referencia sea de otro tipo no significa que los métodos que se ejecuten sean otros. Siguen siendo los de la instancia.
- Algunos usos habituales del polimorfismo en Java son:
  - Implementación de colecciones genéricas.
  - Implementación de métodos genéricos.

# Polimorfismo

Ejemplo de colección genérica:

```
public class TestPolimorfismo
{
    public static void main(String[] args)
    {
        Animal[] animales = new Animal[4];
        animales[0] = new Lobo();
        animales[1] = new Perro();
        animales[2] = new Leon();
        animales[3] = new Tigre();
        for(int i=0; i<animales.length; i++)
        {
            animales[i].dormir();
            animales[i].comer();
        }
    }
}
```

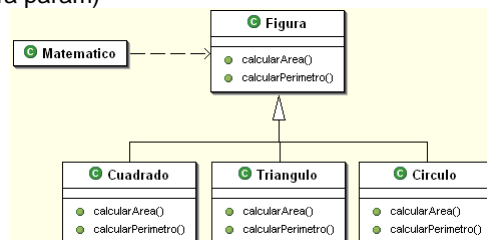


# Polimorfismo

Ejemplo de método genérico:

```
public class Matematico
{
    public double calcularArea(Figura param)
    {
        return param.calcularArea();
    }
}
```

```
public class TestPolimorfismo
{
    public static void main(String[] args)
    {
        Matematico m = new Matematico();
        m.calcularArea(new Circulo(1.2,2.4,13.0));
        m.calcularArea(new Triangulo(1.1,1.1,2.3,2.3,4.1,4.1));
    }
}
```



# Polimorfismo

- Con el polimorfismo podemos desarrollar código que no tiene que ser modificado por la introducción en el programa de nuevas subclases o tipos debido a:
  - Cambio en las especificaciones.
  - Rediseño.
- Ejemplo: la clase Matematico seguirá funcionando aunque desarrollemos nuevas figuras como Cuadrado, Ameba, etc.... siempre y cuando hereden de la superclase Figura.

## Ejemplo

- Supongamos que necesitamos implementar una clase para almacenar dos Lobos en nuestro proyecto.

```
public class MiLista
{
    Lobo l1 = null;
    Lobo l2 = null;

    public boolean add(Lobo param)
    {
        if(l1 == null)
        {
            l1 = param;
            return true;
        }
        else if(l2 == null)
        {
            l2 = param;
            return true;
        }
        else
            return false;
    }
}
```

## Ejemplo

¡Opps! Ahora nos dicen que en el mismo proyecto también necesitamos almacenar dos Gatos.

Tenemos distintas alternativas:

Crear una clase nueva MiLista2.

Añadir a MiLista dos atributos nuevos del tipo Gato y otro método add() que reciba un Gato.

Modificar MiLista para que maneje el tipo genérico Animal y así nos valga tanto para Lobos como para Gatos e incluso otros animales en el futuro.

```
public class MiLista
{
    Animal l1 = null;
    Animal l2 = null;

    public boolean add(Animal param)
    {
        if(l1 == null)
        {
            l1 = param;
            return true;
        }
        else if(l2 == null)
        {
            l2 = param;
            return true;
        }
        else
            return false;
    }
}
```

## Ejemplo

Hablando con un colega de otro proyecto en la máquina de café, nos comenta que en su proyecto necesita implementar una clase para almacenar dos Triangulos.

Le podríamos pasar nuestra clase si la hubiéramos hecho más genérica.

¿No heredaba en Java todo de la clase Object?

En Java encontraremos multitud de ejemplos que usen el Polimorfismo con este fin.

```
public class MiLista
{
    Object l1 = null;
    Object l2 = null;

    public boolean add(Object param)
    {
        if(l1 == null)
        {
            l1 = param;
            return true;
        }
        else if(l2 == null)
        {
            l2 = param;
            return true;
        }
        else
            return false;
    }
}
```

# Castings

- El casting es una forma de realizar conversiones de tipos.
- Hay dos clases de casting:
  - UpCasting: conversión de un tipo en otro superior en la jerarquía de clases. No hace falta especificarlo.
  - DownCasting: conversión de un tipo en otro inferior en la jerarquía de clases.
- Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.

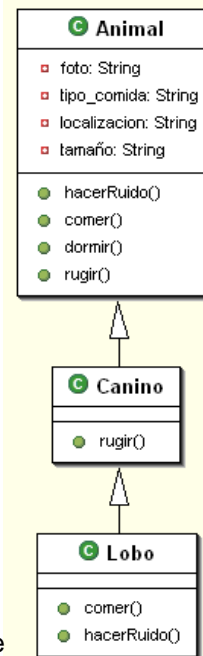
## Ejemplo

```
public class Test
{
    public static void main (String[] args)
    {
        Lobo lobo = new Lobo();

        // UpCastings
        Canino canino = lobo;
        Object animal = new Lobo();
        animal.comer();

        // DownCastings
        lobo = (Lobo)animal;
        lobo.comer();
        Lobo otroLobo = (Lobo)canino;
        Lobo error = (Lobo) new Canino();
    }
}
```

**No compila.** No puedes llamar al método comer() sobre un Object. No puedes convertir un Canino en un Lobo.



## Clases abstractas

- A menudo existen clases que sirven para definir un tipo genérico pero que no tiene sentido instanciar (crear objetos de ella).
- Por ejemplo:
  - Puede tener sentido instanciar un Circulo pero a lo mejor no instanciar una Figura, porque... ¿qué figura es? ¿cuál es su área? ¿y su perímetro?
- Estas clases pueden estar siendo usadas simplemente para agrupar bajo un mismo tipo a otras clases, o para contener código reutilizable, o para forzar un API a sus subclases.....

## Clases abstractas

- Las clases se definen como abstractas mediante la *keyword*: `abstract`.
- Declaración de una clase abstracta:
  - `modificador_acceso abstract class nom_clase`  
`{`  
`}`
- Ejemplo:
  - `public abstract class MiClase`  
`{`  
`}`



## Ejemplo

```
public abstract class Animal
{
    .....
}

public abstract class Canino extends Animal
{
    .....
}

public class Perro extends Canino
{
    .....
}

public class Test
{
    public static void main(String[] args)
    {
        Canino c;
        c = new Canino();
        c.rugir();
    }
}
```

No compila. Canino es una clase abstracta.

```
classDiagram
    Animal <|-- Canino
    Canino <|-- Perro
    class Animal {
        +String foto
        +String tipo_comida
        +String localizacion
        +String tamaño
        +hacerRuido()
        +comer()
        +dormir()
        +rugir()
    }
    class Canino {
        +rugir()
    }
    class Perro {
        +comer()
        +hacerRuido()
        +sacarPaseo()
        +vacunar()
    }
```

## Métodos abstractos

- Además de clases abstractas, también podemos tener métodos abstractos.
- Una clase abstracta significaba que tenía que ser heredada. No podía ser instanciada.
- Un método abstracto significa que tiene que ser sobrescrito. No está implementado.
- Una clase con uno o varios métodos abstractos tiene que ser declarada abstracta.
- No obstante una clase abstracta no tiene porque tener métodos abstractos.

# Métodos abstractos

- Los métodos se definen como abstractos mediante la *keyword*: `abstract`.
- Declaración de un método abstracto:  
`modif_acceso abstract tipo_retorno nombre([tipo param,...]);`
- Ejemplo:  
`public abstract void miMetodo();`
- El objetivo de un método abstracto es forzar una interfaz (API) pero no una implementación.

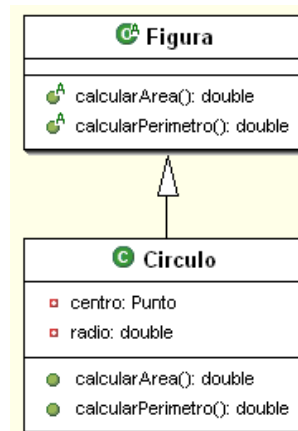
## Ejemplo

```
public abstract class Figura
{
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

public class Circulo extends Figura
{
    private Punto centro = null;
    private double radio = 0.0;

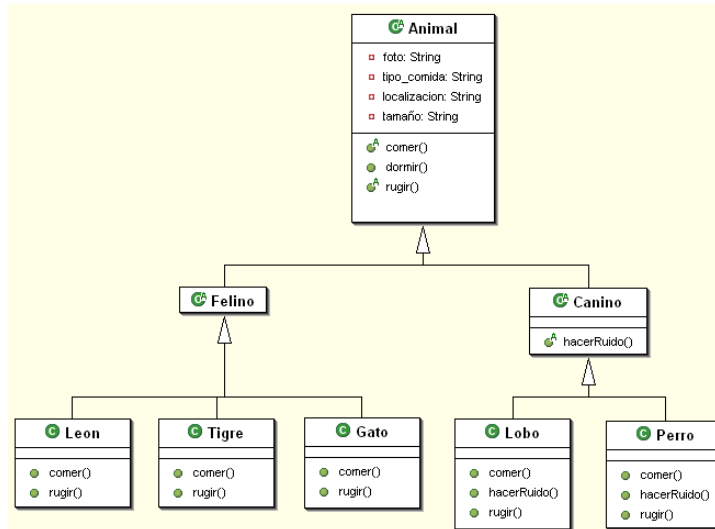
    public double calcularArea()
    {
        return Math.PI*radio*radio;
    }

    public double calcularPerimetro()
    {
        return 2*Math.PI*radio;
    }
}
```





# Mas Polimorfismo


 Diseño del aplicativo SimAnimal 2004.




# Mas Polimorfismo

 ¿Qué ocurre si queremos reusar el diseño para un aplicativo de Tienda de Mascotas?

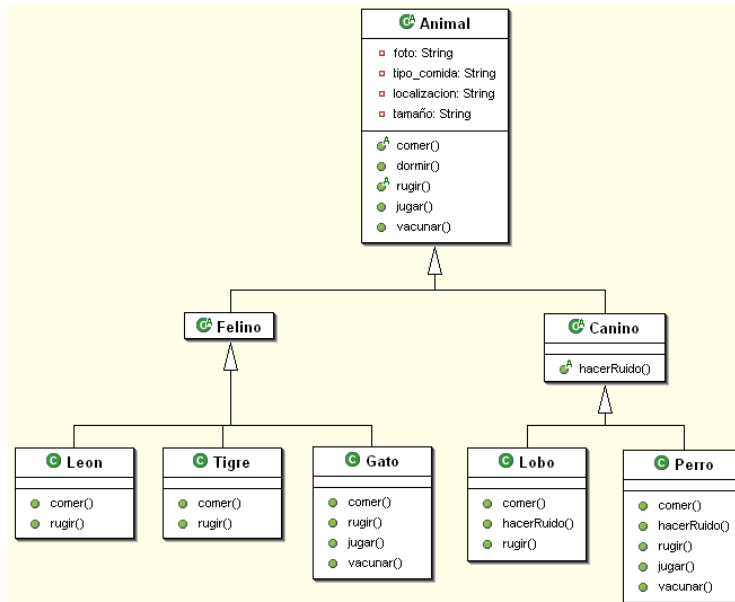
 Una primera aproximación sería añadir a la clase **Animal** todos los métodos específicos de una mascota.

 Automáticamente todas las mascotas tendrán los métodos necesarios.

 Pero también los tendrán las no mascotas.

 Y seguro que hay que retocar cada mascota reescribiendo sus métodos porque tengan alguna peculiaridad.

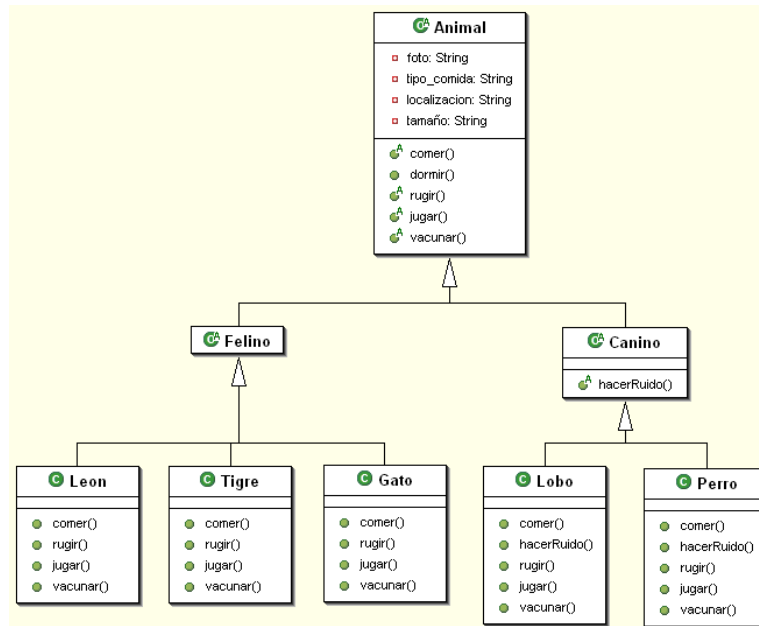
# Mas Polimorfismo



# Mas Polimorfismo

- Modificamos la primera aproximación definiendo los métodos de las mascotas en la clase Animal como abstractos de manera que cada mascota los implemente.
- Así todas las mascotas heredan el interfaz e implementan su comportamiento dependiendo de la mascota en concreto.
- Pero no solo el resto de animales heredarán también el interfaz si no que tienen que implementarlo aunque sea vacío.

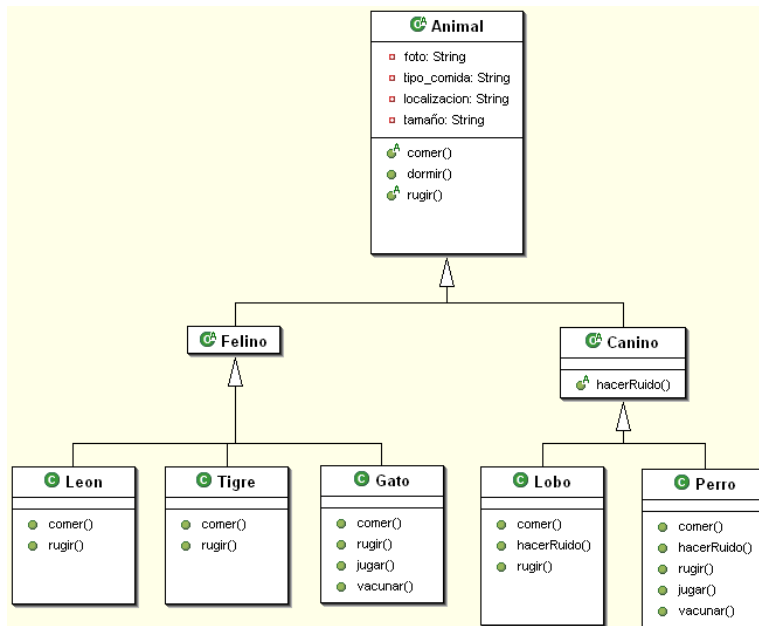
# Mas Polimorfismo



# Mas Polimorfismo

- Otra aproximación sería introducir los nuevos métodos solo en las mascotas.
- Así ya no nos tenemos que preocupar de que haya clases que sin ser mascotas tengan métodos de estas.
- Sin embargo esto implica otro tipo de problemas como que los programadores de mascotas tendrán que ponerse de acuerdo en el interfaz de estas y siempre llevarlo a raja tabla puesto que ahora no se hereda y el compilador no nos ayuda con los posibles errores.
- Otro inconveniente muy importante es que no tenemos posibilidad de usar el polimorfismo con las mascotas.

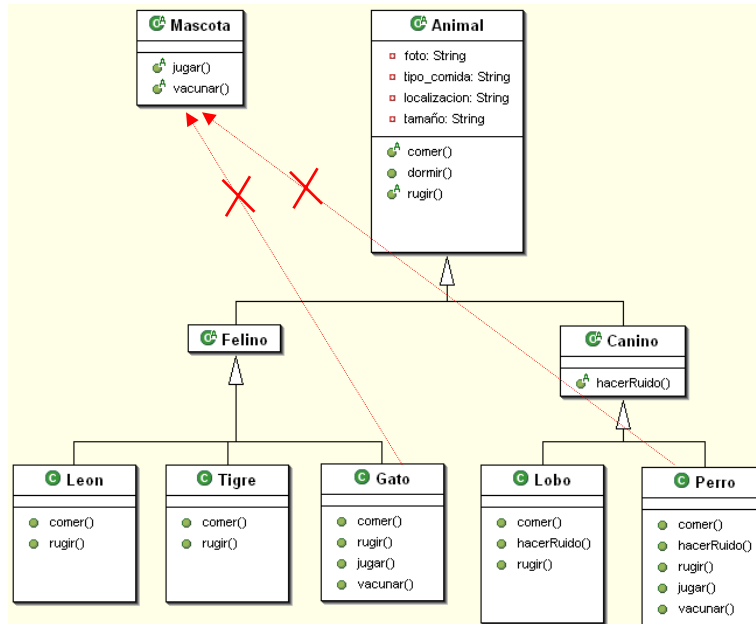
# Mas Polimorfismo



# Mas Polimorfismo

- La solución que parece óptima, sería tener otra clase abstracta llamada Mascota con los métodos abstractos de las mascotas. Y que todas las mascotas heredasen de ella.
- Así, ya no nos tenemos que preocupar de que haya clases que sin ser mascotas tengan métodos de estas.
- Todas las mascotas cumplirán forzosamente el API de las mascotas y el compilador nos ayudará a asegurarlo.
- Y también tendremos la posibilidad de usar el polimorfismo con las mascotas.
- Pero eso significa que habrá clases que heredarán de dos clases a la vez y en Java no existe la herencia múltiple.

# Mas Polimorfismo



## Interfaces

- Los interfaces en Java nos solucionan en parte la no existencia de la herencia múltiple; habilitando así las posibilidades del polimorfismo en la herencia múltiple sin los problemas que esta conlleva.
- Los interfaces son un tipo de clase especial que no implementa ninguno de sus métodos. Todos son abstractos. Por tanto no se pueden instanciar.
- La declaración de un interface Java se realiza mediante la *keyword*: `interface` seguido de su nombre.

# Interfaces



Declaración de un interface:



```
modificador_acceso interface nombre_interface  
{  
}
```



Ejemplo:



```
public interface MiInterface  
{  
}
```



Siguen siendo clases Java por lo que su código fuente se guarda en un fichero texto de extensión \*.java y al compilarlo se generará un \*.class

# Interfaces



Los métodos se definen como abstractos mediante la *keyword*: *abstract*.



Declaración de un método abstracto:



```
modif_acceso abstract tipo_retorno nombre([tipo param,...]);
```



Ejemplo:



```
public abstract void miMetodo();
```



El objetivo de un método abstracto es forzar una interfaz (API) pero no una implementación.



# Interfaces

- De los interfaces también se hereda, aunque se suele decir implementa. Y se realiza mediante la *keyword*: `implements`.
- Declaración de la herencia:  

```
modif_acceso class nom_clase implements nom_interface[,nom_int....]  
{  
}
```
- Ejemplo:  

```
public class MiClase implements MiInterface  
{  
}
```

# Interfaces

- Una clase puede heredar de múltiples interfaces.
- Una clase puede heredar de otra clase y a la vez heredar de múltiples interfaces.
- Un interface puede también definir constantes.
- Si una clase que hereda de un interface, no implementa todos los métodos de este, deberá ser definida como abstracta.

# Ejemplo

```
public interface Mascota
{
    public abstract void jugar();
    public abstract void vacunar();
}
```

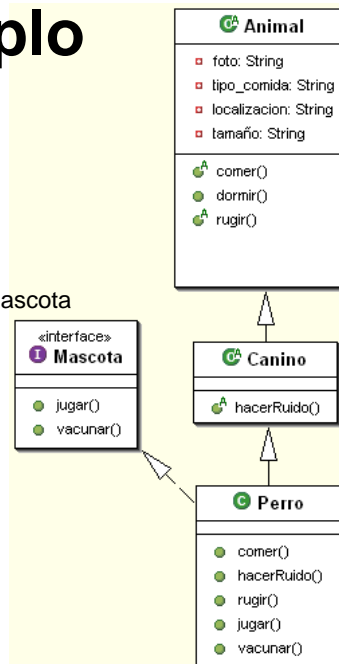
```
public class Perro extends Canino implements Mascota
{
    public void comer() { ... }

    public void hacerRuido() { ... }

    public void rugir() { ... }

    public void jugar() { ... }

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

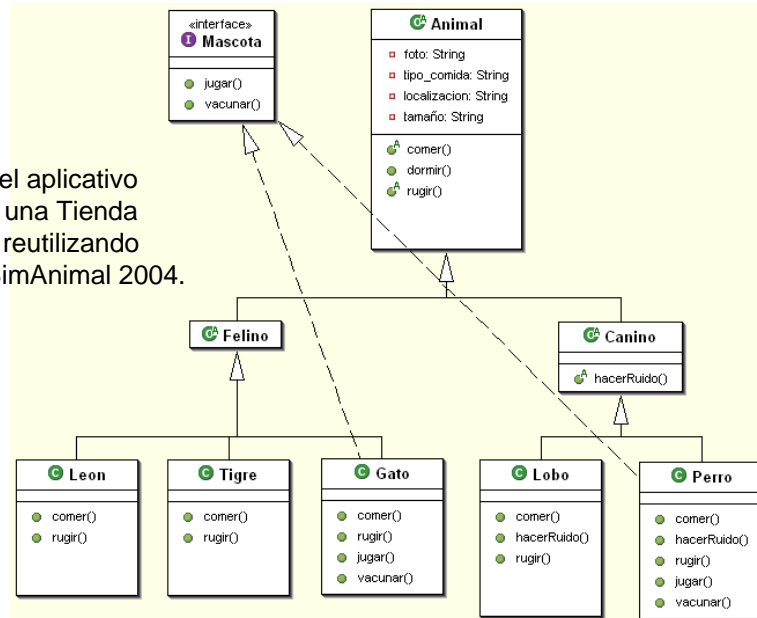


## Interfaces

- Un interface se trata como un tipo cualquiera.
- Por tanto, cuando hablamos de polimorfismo, significa que una instancia de una clase puede ser referenciada por un tipo interface siempre y cuando esa clase o una de sus superclases implemente dicho interface.
- Un interface puede heredar de otros interfaces.

# Mas Polimorfismo




Diseño final del aplicativo de gestión de una Tienda de Mascotas, reutilizando el aplicativo SimAnimal 2004.




## Interface vs. Clase Abstracta

- Un interface no puede implementar ningún método.
- Una clase puede implementar n interfaces pero solo una clase.
- Un interface no forma parte de la jerarquía de clases. Clases dispares pueden implementar el mismo interface.
- El objetivo de un método abstracto es forzar una interfaz (API) pero no una implementación.

## Clases, Subclases, Abstractas e Interfaces

-  Haremos una clase que no herede de nadie cuando la clase no pase la prueba de Es-Un.
-  Haremos una subclase cuando necesitemos hacer una especialización de la superclase mediante sobreescritura o añadiendo nuevos métodos.
-  Haremos una clase abstracta cuando queramos definir un grupo genérico de clases y además tengamos algunos métodos implementados que reutilizar. También cuando no queramos que nadie instancie dicha clase.

## Clases, Subclases, Abstractas e Interfaces

-  Haremos un interface cuando queramos definir un grupo genérico de clases y no tengamos métodos implementados que reutilizar. O cuando nos veamos forzados por la falta de herencia múltiple en Java.

# Bibliografía



## Head First Java

Kathy Sierra y Bert Bates.  
O'Reilly



## Learning Java (2<sup>nd</sup> edition)

Patrick Niemeyer y Jonathan Knudsen.  
O'Reilly.



## Thinking in Java (3<sup>rd</sup> edition)

Bruce Eckel.  
Prentice Hall.



## The Java tutorial

<http://java.sun.com/docs/books/tutorial/>