

0101010  
0100101  
1101010

# UD7.2- POLIMORFISME

Programació – 1er DAW/DAM

# 0. CONTINGUTS

---

- Tipus de polimorfisme
  - De redefinició
  - De tipus
    - Upcasting
    - Downcasting
    - Operador *instanceOf*
    - Tipus estàtic i tipus dinàmic
    - Enllaç dinàmic
- Classes abstractes
- La classe **Object**

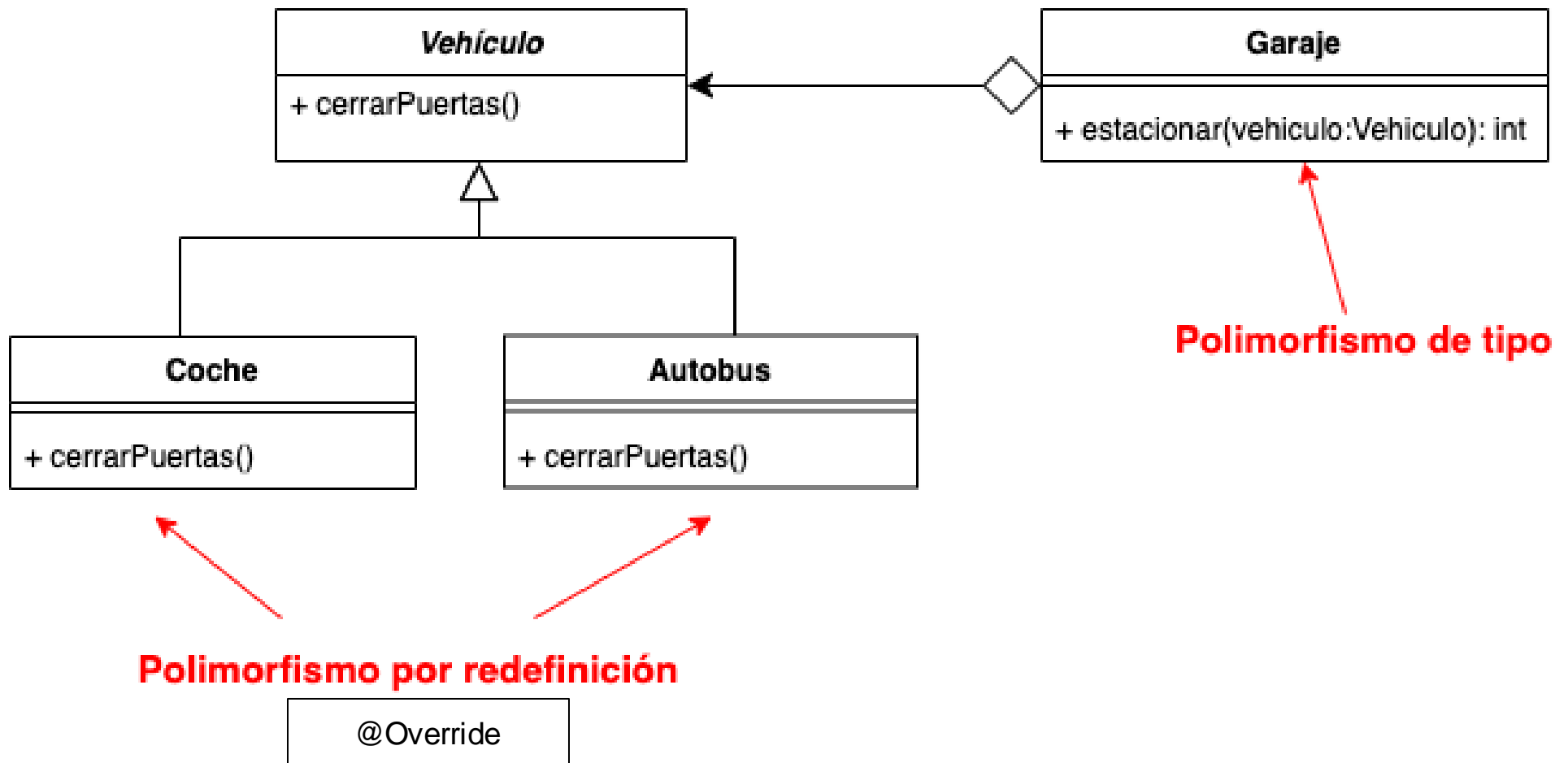
# 1. POLIMORFISME A JAVA

---

A **Java**, podem trobar **2 tipus** de **polimorfisme**:

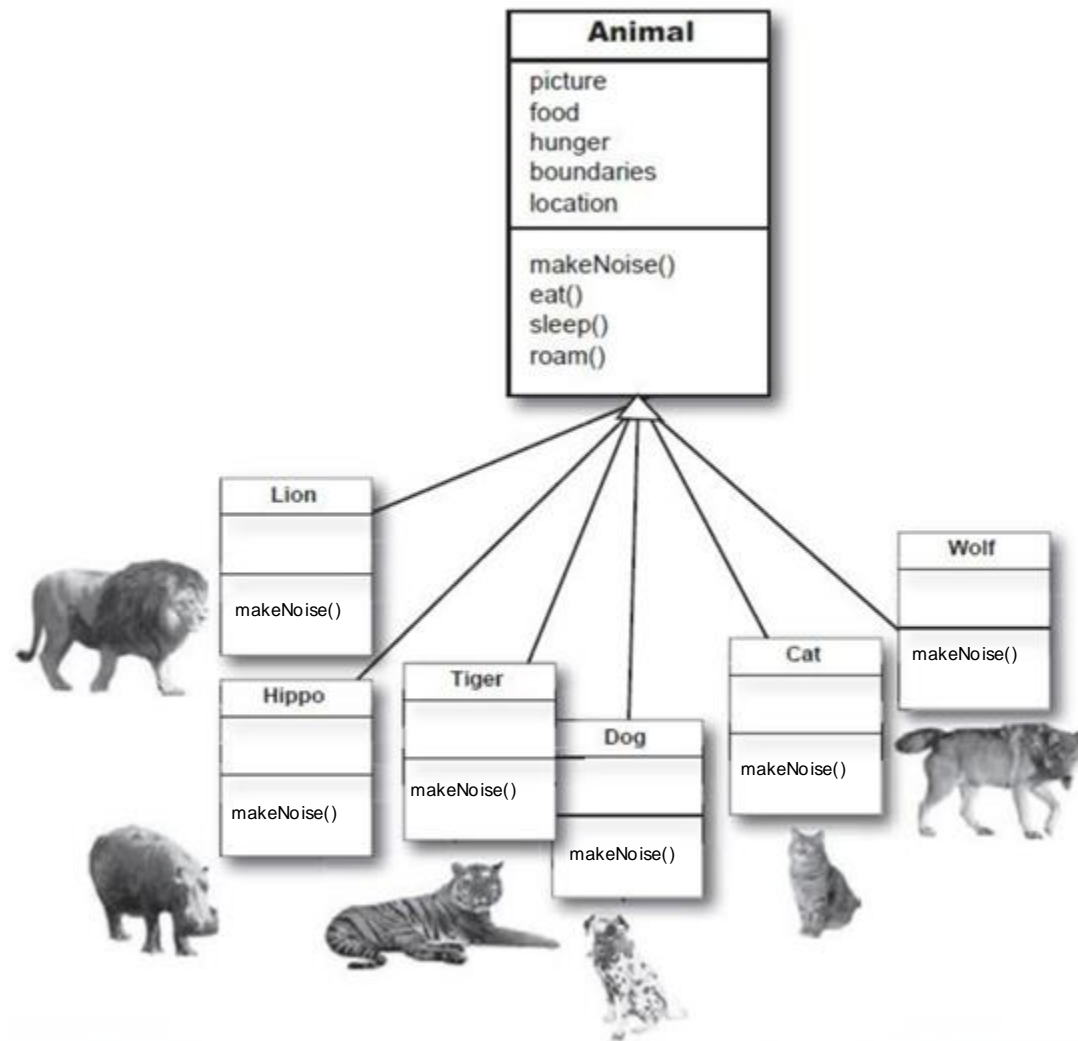
- **Polimorfisme per redefinició (sobreescriptura)**:  
ens permet, baix *una mateixa interfície* d'un mètode,  
obtenir **diferents comportaments**. (*Vist fins el moment*)
- **Polimorfisme de tipus**: mitjançant el qual una classe  
o **mètode** pot **manejar** objectes de **diferent tipus**  
mitjançant el que coneixem com **lligadura dinàmica**.

# 1. POLIMORFISME A JAVA



# 1.1 POLIMORFISME DE TIPUS

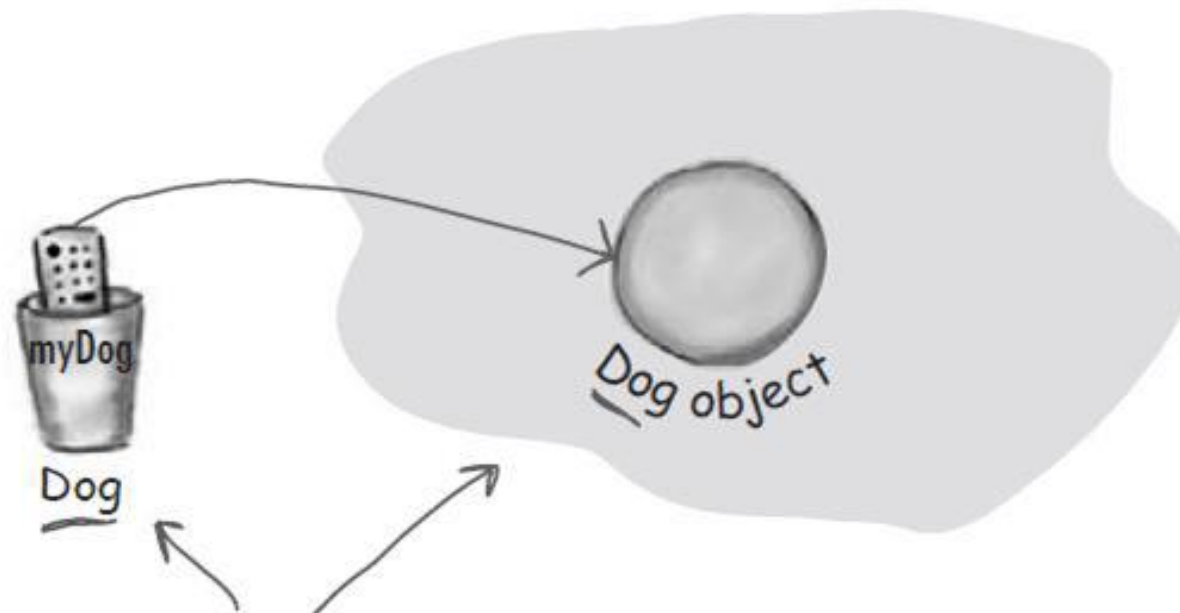
Suposant la següent **jerarquia de classes**...



# 1.1 POLIMORFISME DE TIPUS

## Sense polimorfisme ...

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



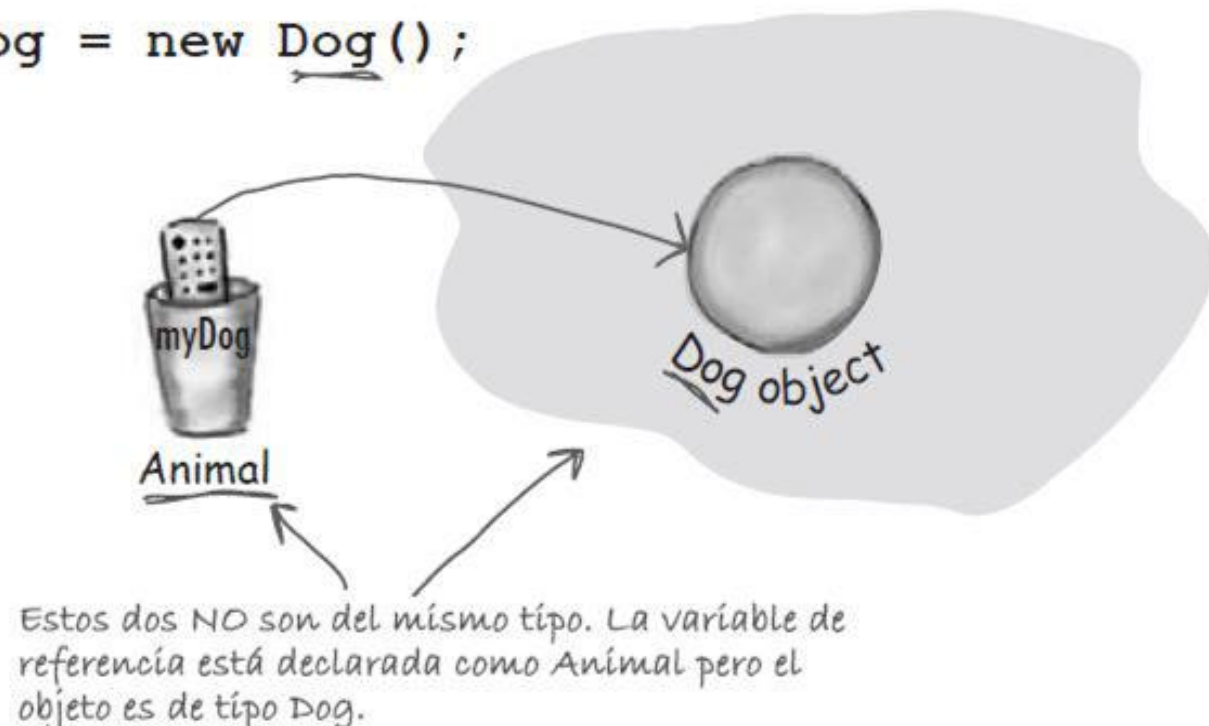
La variable de referència `myDog` y el objeto de tipo `Dog` son del mismo tipo

La variable de referència **myDog** i l'**objecte** sempre són del mateix tipus **Dog**

## 1.1 POLIMORFISME DE TIPUS

**Amb polimorfisme:** la referència pot ser qualsevol objecte que derive de la superclasse.

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



La variable està declarada com **Animal** encara que conté un objecte de tipus **Dog**

## 1.1.1 *UPCASTING*

- A una **variable** de tipus **ClasseA** podem, a més d'assignar-li **objectes** de la **ClasseA**, assignar-li **qualsevol** objecte d'una **subclasse** seua.

```
public class Animal {..}  
  
public class Lion extends Animal {..}  
  
Animal simba = new Lion();
```

Assignem l'objecte a una  
**variable** del **tipus** de la  
**superclasse** (UPCASTING)



## 1.1.1 UPCASTING

- **Atenció!** com la variable és de tipus `Animal`, **només** tindrem disponibles els mètodes públics heretats d' `Animal` i no els propis (afegits) de `Leon`.

```
public class Animal{  
    public void menjar(){  
        hungry = 0;  
    }  
}
```

```
public class Lion extends Animal{  
    public void caçar(){  
        velocitat = 30  
        System.out.println("ARGHHH!!");  
    }  
}
```

```
public class TestAnimal {  
    public static void main(String[] args) {  
        Animal simba = new Leon();  
        simba.menjar(); ✓  
        simba.caçar(); ✗  
    }  
}
```

## 1.1.2 DOWNCASTING

- Si una **variable** del **tipus** de la **superclasse** fa **referència** a un **objecte** d'una **subclasse** podem dur a terme la seva **conversió** al **tipus original**.

```
public class TestAnimal{  
    public static void main(String[] args) {  
        Animal simba = new Lion();  
        Lion simbaLeon = (Lion) simba;  
        simbaLeon.menjar();  
        simbaLeon.caçar();  
    }  
}
```

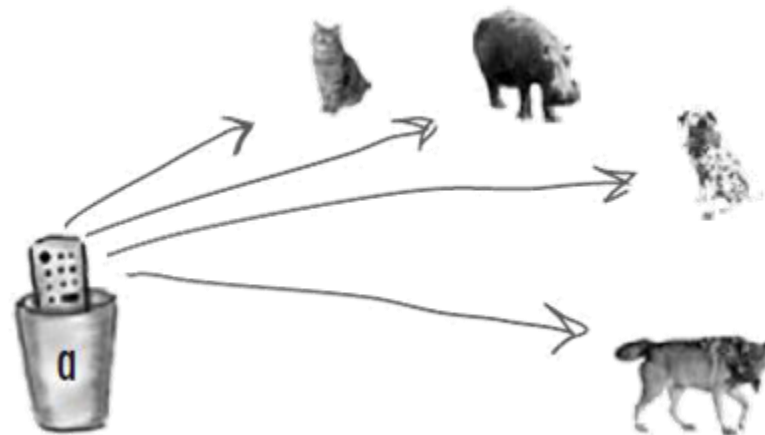
**Conversió al  
tipus original  
(DOWNCASTING)**

- ATENCIÓ:** Aquesta operació **només es pot dur a terme** si l'objecte referenciat per "simba" és realment de **tipus** Leon.
  - En cas contrari, es llançarà l'excepció `ClassCastException`.

## 1.2 EXEMPLE I

```
public class Veterinari {  
  
    public void posarInjeccio (Animal a) {  
        // Emetrà un so de dolor característic de l'animal  
        a.makeNoise();  
    }  
  
}
```

**El paràmetre *a* pot  
albergar qualsevol  
objecte del subtipus  
*Animal***



## 1.2 EXEMPLE II

```
public class TestVeterinari {
```

```
    Dog dog = new Dog();
```

```
    Lion lleo = new Lion();
```

```
    Veterinari veterinari = new Veterinari();
```

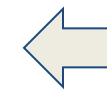
```
    veterinari.posarInjeccio(dog);
```

```
    veterinari.posarInjeccio(lleo);
```

```
}
```



**Emetrà el so  
del gos**



**Emetrà el so  
del lleó**

## 1.3. TIPUS ESTÀTIC I TIPUS DINÀMIC

- **Tipus estàtic:** tipus amb què es declara la variable que referència a l'objecte
  - Determina **QUÈ** es pot fer.
- **Tipus dinàmic:** tipus de l'objecte al què apunta la variable.
  - Determina **COM** es fa.

### Exemples:

```
Animal animal1 = new Animal(); // tipus estàtic = tipus dinàmic
```

```
Animal animal2 = new Lion(); // tipus estàtic != tipus dinàmic
```



**Enllaç dinàmic**  
(Dynamic Binding)



Tipus estàtic: Animal  
Tipus dinàmic: Lion

## 1.4 ARRAYS

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

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

```
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()

```
}
```

## 1.5 OPERADOR `instanceOf`

- Si una variable pot apuntar a diferents **subtipus** d'objectes, com sabem el subtipus **específic** de l'**objecte** al **que apunta** una variable?
- El operador `instanceOf` ens permet saber si un tipus d'objecte pertany a una subclasse específica.

```
Animal simba = new Lion();  
if (simba instanceof Lion) {  
    Lion simbaLeon = (Lion) simba;  
    simbaLeon.atacar();  
}
```

**Curiositat.** Si vullguem obtenir el tipus d'un objecte, podem fer ús del mètode `getClass()` combinat amb `getSimpleName()`  
Exemple:  
`simba.getClass().getSimpleName()`

**També podríem fer:**  
`((Lion) simba).atacar()`

# ACTIVITAT PRÈVIA

- **Activitat 7.** Fent ús de les classes desenvolupades a l'activitat 1 (de la presentació anterior), du a terme les següents modificacions:
  - La classe `PersonaAnglesa` al no formar part de la UE, haurà de tenir un atribut `passaport` i el mètode `mostrarPassaport()`.
  - Crea una classe `Duana`, haurà de disposar d'un atribut de tipus array de `Persones` que continga totes les persones que hi ha en un moment donat a la duana. L'aforament màxim és de 10 persones.
    - Crea un mètode `entrar(Persona persona)`: Quan entre una persona aquesta saludarà (Cridant al mètode corresponent) i, si queda espai, se'l deixarà entrar (s'afegirà a l'array de persones). Si la persona és de tipus anglès se li demanarà que mostre el seu passaport.
    - Si s'arriba al número màxim de persones en la aduana, se l'informarà que no pot entrar.
    - L'aduana sempre parla en anglès

Per acabar, crea una classe `TestAduana` on crees diferents tipus de persones que entren a la duana.

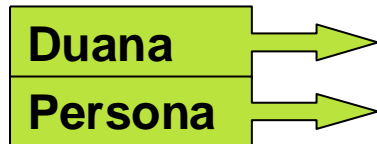
Per a una major comprensió es presenta un exemple d'execució.



# ACTIVITAT PRÈVIA

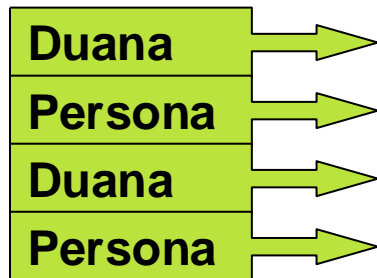
## Exemple d'execució:

===== Actividad Aduana =====



Welcome!

Hola, soy Alex Coloma

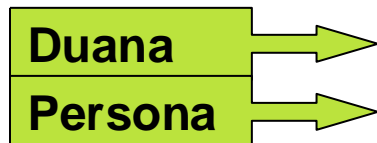


Welcome!

Hello, I'm Peter Giorgio

Could you show me your passport, please?

My passport is: PSC12131231



Welcome!

Bonjour, je suis Napoleon Bonaparte

# ACTIVITAT PRÈVIA (GRAN GRUP)

---

- **Activitat 8.** Respon a les [següents qüestions](#)

## 2. CLASSES ABSTRACTES (I)

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?

## 2. CLASSES ABSTRACTES (II)

A quin animal s'assembla un `new Animal()` ?



## 2. CLASSES ABSTRACTES (III)

- Té sentit crear un **Tigre** o un **Llop**.
- Però, té sentit crear un **Animal**? Quina forma té? **Com és?**
- **Necessitem la classe Animal per a heretar**, però realment **no hem de crear un Animal**, però sí un Llop, un Tigre o un Gos.
- Definint una classe com a abstracta (**abstract**), **evitem que pugui ser instanciada** i que es pugui crear objectes d'aquesta classe.

```
public abstract class Animal {  
  
    public void makeNoise(){  
  
    }  
  
}
```

## 2. CLASSES ABSTRACTES (IV)

---

- En dissenyar, haurem de **decidir** si una **classe** serà **abstracta** o **concreta**:
  - **Classe concreta**: Es crearan objectes de la classe.
  - **Classe abstracta**: No es podran crear objectes.
- En definir una classe com **abstracta**, el **compilador** garantirà que **no es podran crear objectes** d'aquesta classe.

```
public abstract class ...
```

## 2. CLASSES ABSTRACTES (V)

```
public class Zoo {  
  
    public static void main(String args[]){  
  
        Animal animal = new Animal();  
        Animal simba = new Lion();  
  
    }  
}
```

**INCORRECTE:** No puc **instanciar** objectes d'una classe abstracta. ¿Com són?

**CORRECTE:** Puc crear **objectes** de tipus **Lion** i assignar-lo a una variable de tipus **Animal**

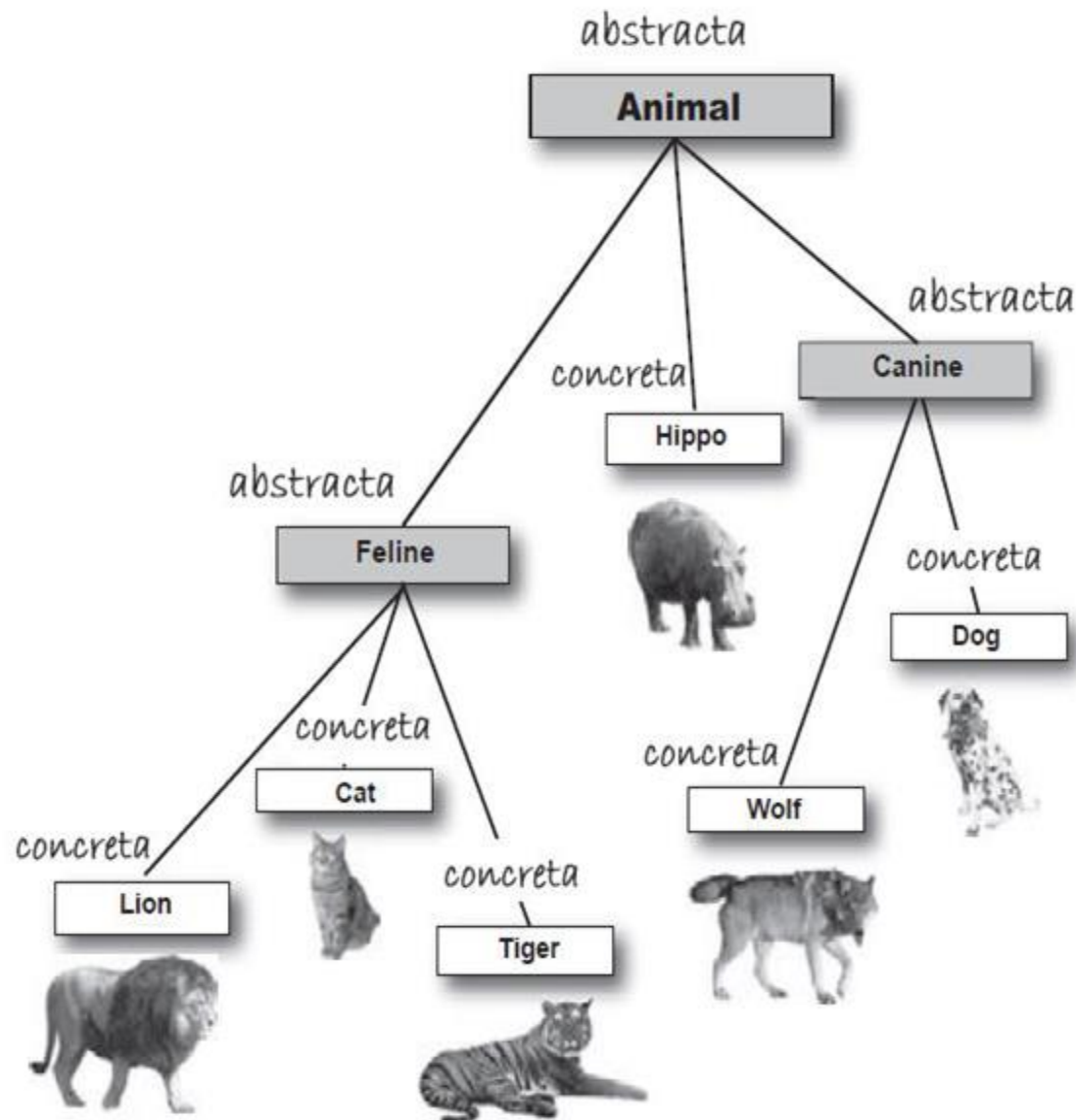
## 2. CLASSES ABSTRACTES (VI)

---

- Les **classes abstractes**, a excepció de les que tenen mètodes no abstractes, no tenen ús, ni valors, únicament **serveixen** per **heretar-ne**.
- Quan utilitzem classes abstractes, soles podem crear objectes de les subclasses.
- A l'**API de Java** trobarem **moltes classes abstractes**. Per exemple, la classe *Component* és abstracta, però la classe *Button* és concreta.



## 2.1 EXAMPLE: JERARQUIA D'ANIMALS



## 2.2 MÈTODES ABSTRACTES (ABSTRACT)

- Una **classe abstracta** ha de ser estesa (**extends**) per a poder utilitzar-la. Podem distingir **2 tipus** de mètodes:
  - **Mètodes abstracte:** tota subclasse que estenga la classe base ha d'implementar aquests mètodes (**Obligatorietat**).
    - Els mètodes abstractes no **tenen cos**(codi)
    - Són **mètodes genèrics** implementats a les subclasses.
    - Els mètodes abstractes han d'estar a classes abstractes.

```
abstract public class Animal {  
  
    //Tota sub-classe ESTÀ OBLIGADA implementar el mètode  
    public abstract void makeNoise();  
  
}
```

# ACTIVITAT FORMATIVA (PARELLES)

---

- **Activitat 9.** Jerarquia d'animals (disposes de l'enllaç a Aules)

## 3. LA CLASSE **Object**

---

- Quin és el comportament de qualsevol objecte?
  - Les classe **Object** ens proporciona una sèrie de comportaments (mètodes) comuns a qualsevol objecte.

**Alguns d'aquests mètodes són:**

- equals()
- getClass()
- toString()
- compareTo()
- ...

## 3.1 MÈTODE *equals()*

### **equals(Object o)**

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

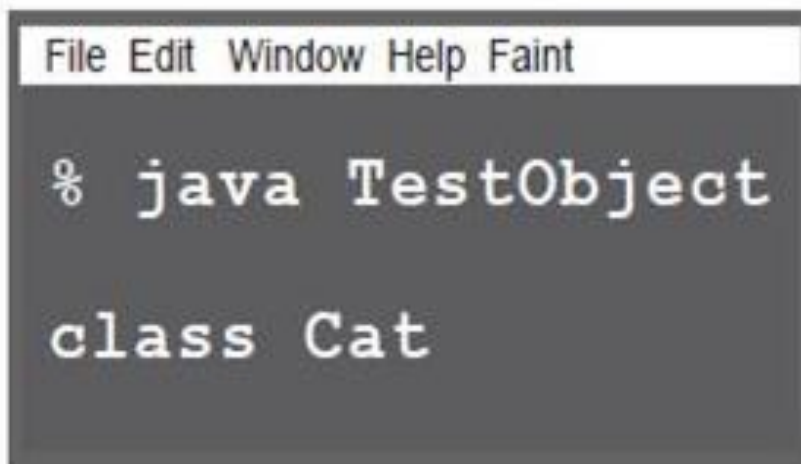
**Ens permet decidir  
quan volem que 2  
objectes es  
consideren iguals**

```
public class Dog {  
    @Override  
    public boolean equals(Object other) {  
        if (!(other instanceof Dog)){  
            return false;  
        }  
        Dog otroPerro = ((Dog) other);  
        return this.name.equals(otroPerro.name);  
    }  
}
```

## 3.2 MÈTODE *getClass()*

### **getClass()**

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



The screenshot shows a Java IDE window with a menu bar (File, Edit, Window, Help, Faint) and a text area containing the following text:

```
% java TestObject  
  
class Cat
```

Nos devuelve el  
nombre de la  
clase del objeto  
instanciado

## 3.3 MÈTODE *toString()*

### **toString()**

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

File Edit Window Help LapseIntoComa

```
% java TestObject
```

```
Cat@7d277f
```

**Ens permet  
especificar com  
serà convertit  
un objecte a un  
de tipus String**

```
public class Vehicle {  
    @Override  
    public String toString() {  
        return this.matricula;  
    }  
}
```

# ACTIVITAT PRÈVIA

---

**Activitat 10.** Sobreescriu el mètode `equals` a la classe `Animal` de l'activitat 9 de manera que es considere que 2 animals són iguals si tenen el **mateix tipus d'alimentació** i a la mateixa vegada són del **mateix tamany**. A la classe `TestAnimal` compara diferents parelles d'animals (fes servir els ja creats) fent ús del mètode `equals`.



## 4. LA CLASSE `Objects`

---

- La classe `Objects` (no confondre amb `Object`) en Java és una utilitat introduïda en Java 7 que proporciona **mètodes estàtics** per a operacions comuns en objectes.
- Ofereix mètodes per a **comparar objectes, transformar-los a cadena** i manejar els valors **nuls de manera segura**, entre altres.
- Seria paregut al que ja podem fer amb `Object`, però aportant el maneig segur de nuls.
- Mètodes més importants: `equals`, `isNull`, `nonNull`, `deepEquals`, `toString`, `clone`, **etc.**

API: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Objects.html>

## 4. LA CLASSE Objects. Examples

```
Animal leon = new Leon("leon", "felino");  
Animal lobo = null;
```

```
Objects.equals(leon, lobo); // retorna false, no son iguals  
leon.equals(lobo);         // sentència equivalent  
Objects.equals(leon, lobo); // sentència equivalent
```

¿Què ocurriria si fem `lobo.equals(leon)`?

**Exception in thread "main" java.lang.NullPointerException**

Per tant, utilitzant `Objects` tindríem un codi més segur, menys propens a errors.


## 4. LA CLASSE Objects. Examples.

```
Animal leon = new Leon("leon", "felino");
Animal lobo = null;

System.out.println(Objects.toString(leon));
System.out.println(Objects.toString(lobo));    // null

// "equivalent" a

System.out.println(leon.toString());
System.out.println(lobo.toString());           // Error
```



Degut a aquest tipus de casos en que podem obtenir Error, és millor utilitzar `System.out.println(lobo)`. D'aquesta forma ja **no es genera error**.

# POLIMORFISME

---

Això és tot... de moment :-)