GRAU D'ENGINYERIA INFORMÀTICA

PROGRAMACIÓ II

Bloc 2:

Programació Orientada a Objectes (4)

Laura Igual
Departament de Matemàtica Aplicada i Anàlisi
Facultat de Matemàtiques
Universitat de Barcelona

UNIVERSITAT DE BARCELONA

Index

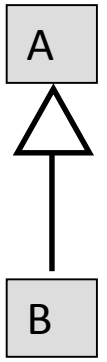
- Lligadures
- Interfícies
- Exercici de repàs

LLIGADURES

Tipus de lligadures: estàtic i dinàmic

- Donada una assignació polimorfa
- Exemple:

```
A a;
a = new B();
```



- Es a dir, una variable de la classe A és una referència a un objecte de la classe B.
- Llavors, es diu que:
  - A és el ***tipus estàtic*** de la variable **`a`** i
  - B es el ***tipus dinàmic*** de **`a`**.
- El tipus estàtic sempre es determina en temps de compilació i és fix, mentre que el tipus dinàmic només es pot conèixer en temps d'execució i pot variar.

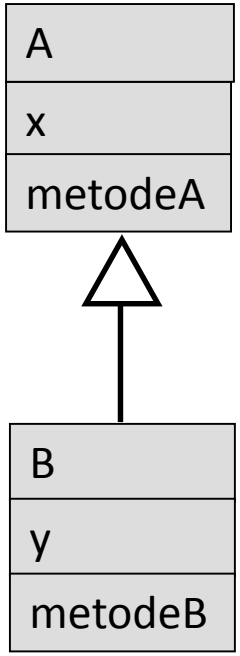
Tipus de lligadures: estàtic i dinàmic

- Java només permet invocar els mètodes i accedir a les variables conegudes per al **tipus estàtic** de a.

```
A a = new B();
a.metodeA();   // Ok
a.metodeB();   // error de compilació
               // metodeB no està definit per a A
```

accés:

```
a.x;           // Ok
a.y;           // error de compilació
```

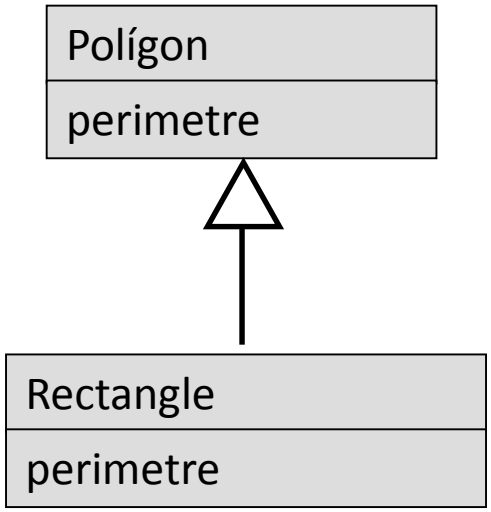


Lligadura dinàmica

En POO, què passa quan realitzem una connexió polimorfa i cridem a una operació redefinida?

```
// Pot referenciar a un objecte Polígon o Rectangle

Poligon poligon;
float peri;
Rectangle rectangle = new Rectangle();
poligon = rectangle;
peri = poligon.perimetre();
```



El compilador no té informació per a resoldre la crida.
Per defecte utilitzaria el tipus de la referència, i per tant generaria una crida a Poligon.perimetre()
Però la referència poligon pot apuntar a un objecte de la classe Rectangle amb una versió diferent del mètode

Lligadura dinàmica

- La solució consisteix en esperar a resoldre la crida en temps d'execució, quan es coneix realment els objectes connectats a la variable **poligon**, i quina és la versió del mètode **perimetre** apropiada.
- Aquest enfocament de resolució de crides s'anomena **lligadura dinàmica**

- Entenem per **resolució d'una crida** el procés pel qual es substituirà una crida a una funció per un salt a la direcció que conté el codi d'aquesta funció.

Exemple

```
public class Poligon {
    public void imprimirIdentitat(){
        System.out.println("Sóc Poligon");
    }
}
```

```
public class Rectangle extends Poligon{
    @Override
    public void imprimirIdentitat(){
        System.out.println(“Sóc Rectangle");
    }
}
```

```
public class Test {
    public static void main(String[] args){
        Poligon[] pol = new Poligon[2];

        Poligon elemA = new Poligon();
        Rectangle elemB = new Rectangle();

        pol[0] = elemA;
        pol[1]= elemB;

        pol[0].imprimirIdentitat();
        pol[1].imprimirIdentitat();
    }
}
```

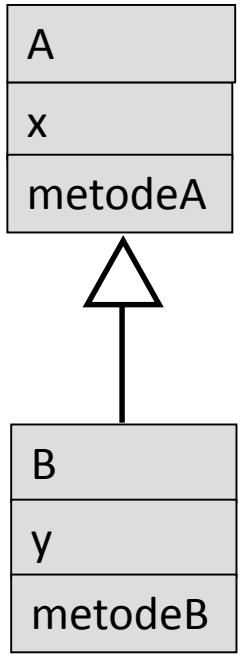
Sortida per pantalla →

```
Sóc Poligon
Sóc Rectangle
```

Exercici

- Donat el següent codi de la classe A i la classe B (que hereta de la classe A) i el diagrama il·lustrant la relació entre les classes

```
public class A{
	protected int x;
	public void metodeA(){
		....
	}
}
public class B extends A{
	private int y;
	public void metodeB(){
		....
	}
}
```



Exercici

Indica al costat de cadascuna de les línies del següent codi si hi haurà errors de compilació o no i explica breument perquè:

```
0 public static void main(String[] args) {
1           A a = new B();
2           B b = new B();
3           A c;
4           c = b;
5           int j = b.x;
6           int i = a.x;
7            int k = a.y;
8           a.metodeA();
9            a.metodeB();
10          b.metodeA();
11          b.metodeB();
12 }
```

Exercici

Indica al costat de cadascuna de les línies del següent codi si hi haurà errors de compilació o no i explica breument perquè:

```
0 public static void main(String[] args) {
1           A a = new B();   OK
2           B b = new B();        OK
3           A c;                  OK
4           c = b;                OK
5           int j = b.x;          OK
6           int i = a.x;          OK
7           int k = a.y;          Error de compilació, l'atribut y no està defint per a A.
8           a.metodeA();          OK
9           a.metodeB();          Error de compilació, metodeB no està definit per a A
10          b.metodeA();          OK
11          b.metodeB();          OK
12 }
```

Exercici

- Especifica si hi ha alguna **conversió** de tipus **implícita** en el codi anterior i en cas afirmatiu en quines línies.

- Si afegim un nou mètode a la classe A anomenat imprimir que imprimeix el missatge “Missatge d’A”, però no el sobreescriu a la classe B, que passa quan fem una crida d’aquesta forma:

  b.imprimir();

- Indica com has de sobreescriure el mètode imprimir a la classe B de manera que quan fas la crida

  b.imprimir();

  La sortida sigui: “Missatge de B”

- Ara, indica com has de sobreescriure el mètode imprimir a la classe B de manera que quan fas la crida

  b.imprimir();

  La sortida sigui: “Missatge d’A”

  “Missatge de B”

Exercici

- Especifica si hi ha alguna **conversió** de tipus **implícita** en el codi anterior i en cas afirmatiu en quines línies.

Solució:

Sí, a la línia 1 i 4

- Si afegim un nou mètode a la classe A anomenat imprimir que imprimeix el missatge "Missatge d'A", però no el sobreescriu a la classe B, que passa quan fem una crida d'aquesta forma:

b.imprimir();

Solució:

Apareixerà el missatge: "Missatge d'A"

- Indica com has de sobreescriure el mètode imprimir a la classe B de manera que quan fas la crida

b.imprimir();

La sortida sigui: "Missatge de B"

Solució:

```
public void imprimir(){
        System.out.println("Missatge de B");
}
```

- Ara, indica com has de sobreescriure el mètode imprimir a la classe B de manera que quan fas la crida

b.imprimir();

La sortida sigui: "Missatge d'A"

"Missatge de B"

Solució:

```
public void  imprimir(){
        super.imprimir();
        System.out.println("Missatge de B");
}
```

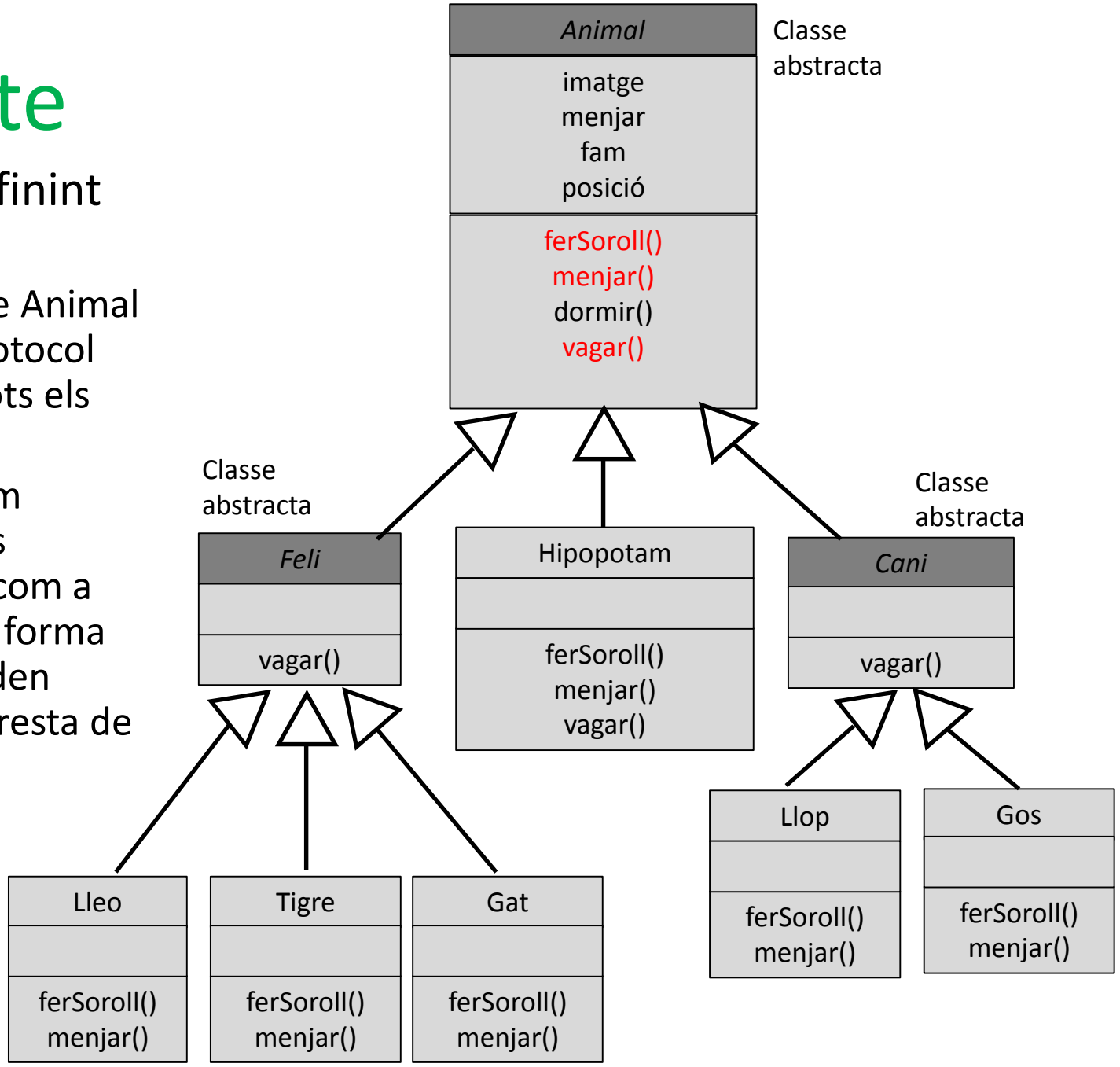
INTERFÍCIES

Introducció

- Introducció d'interfícies amb un exemple
- Construïm la jerarquia d'herències de la classe Animal.

Contracte

- Comencem definint un contracte:
  - La superclasse Animal defineix el protocol comú per a tots els animals.
  - A més, definim algunes de les superclasses com a abstractes de forma que no es poden instanciar. La resta de les classes s'anomenen concretes.



Array polimòrfic

```
public class LlistaAnimals {
    private Animal[] animals = new Animal[5];
    private int nextIndex=0;

    public void add(Animal a){
        if (nextIndex < animals.length){
            animals[nextIndex] = a;
            System.out.println("Animal afegit a la posició " + nextIndex);
            nextIndex++;
        }
    }
}
```

LlistaAnimals.java

Array polimòrfic

```
public class TestLlistaAnimal {
    public static void main(String[] args){
        LlistaAnimals llista = new LlistaAnimals();
        Gos gos = new Gos();
        Gat gat = new Gat();
        llista.add(gos);
        llista.add(gat);
    }
}
```

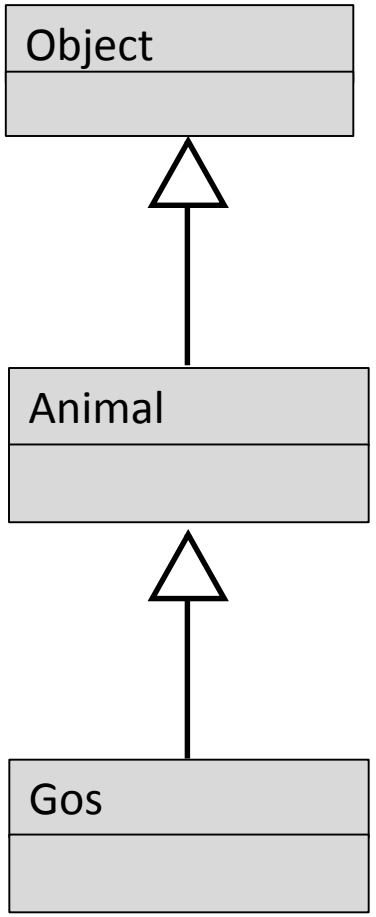
TestLlistaAnimals.java

Estem afegint tot tipus d'animals a l'array

Animal afegit a la posició 0
Animal afegit a la posició 1

Llista polimòrfica

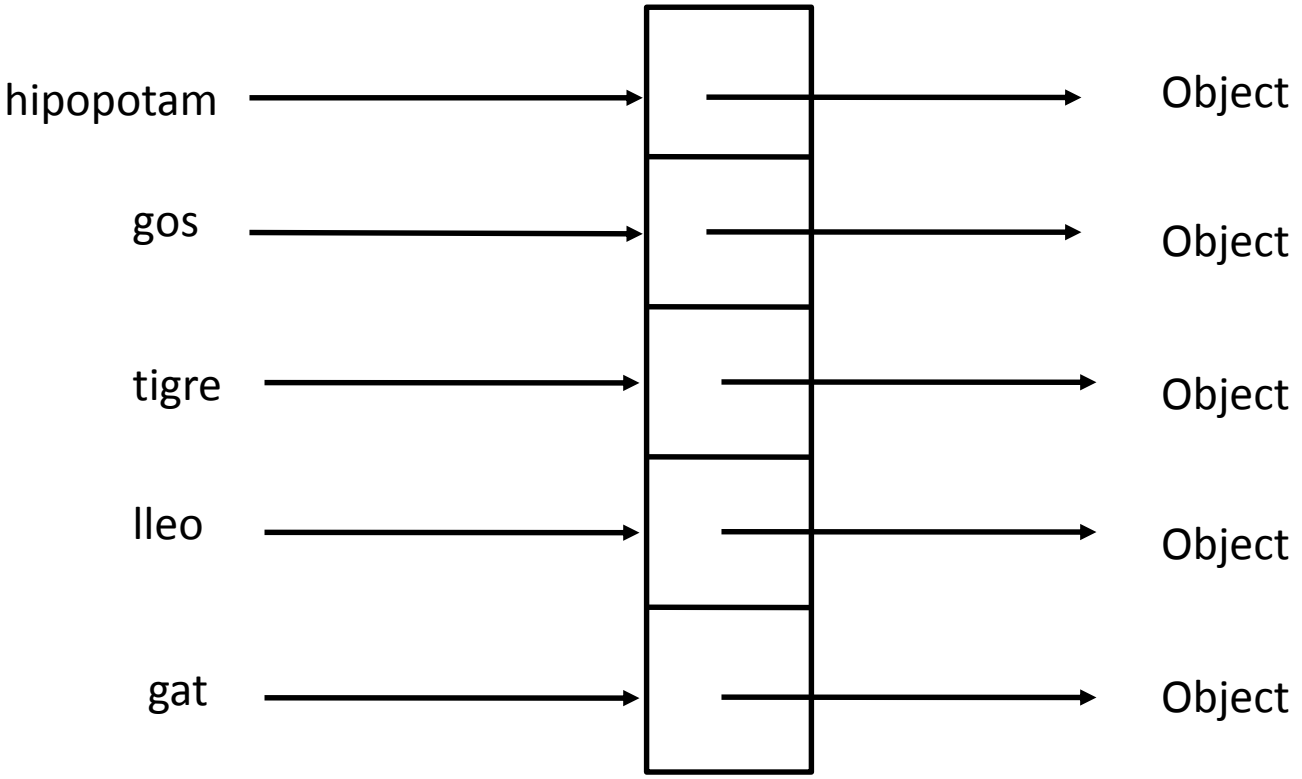
- També es podria optar per fer servir la classe Object que és encara més genèrica i referenciar a qualsevol tipus d'objectes.
- Però això porta alguns inconvenients!!!



Llista polimòrfica

ArrayList laLlistaAnimals = new ArrayList(); ← Llista per contenir tot tipus d'Objectes.

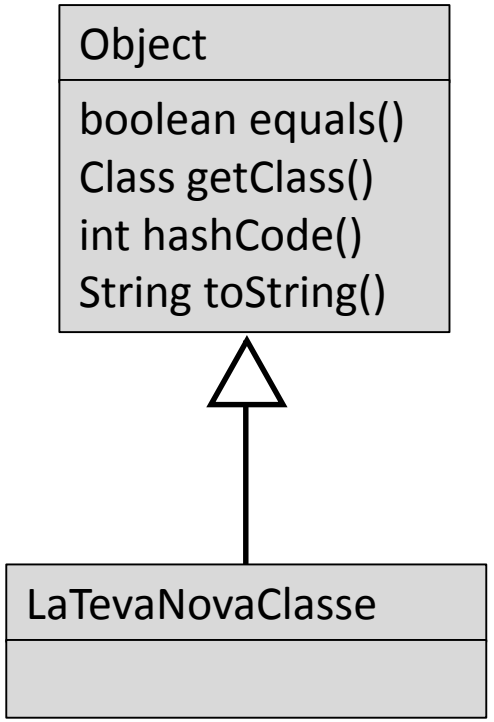
Gos gos = laLlistaAnimals.get(0); No compilarà!



Posis el que posis en cada posició quan recuperis els objectes aquests seran de tipus Object.

Classe Object

- Qualsevol classe implementada per tu hereta de la classe Object.



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");
}
```

```
% java TestObject
false
```

2. getClass()

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

```
% java TestObject
class Cat
```

3. hashCode()

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

```
% java TestObject
8202111
```

4. toString()

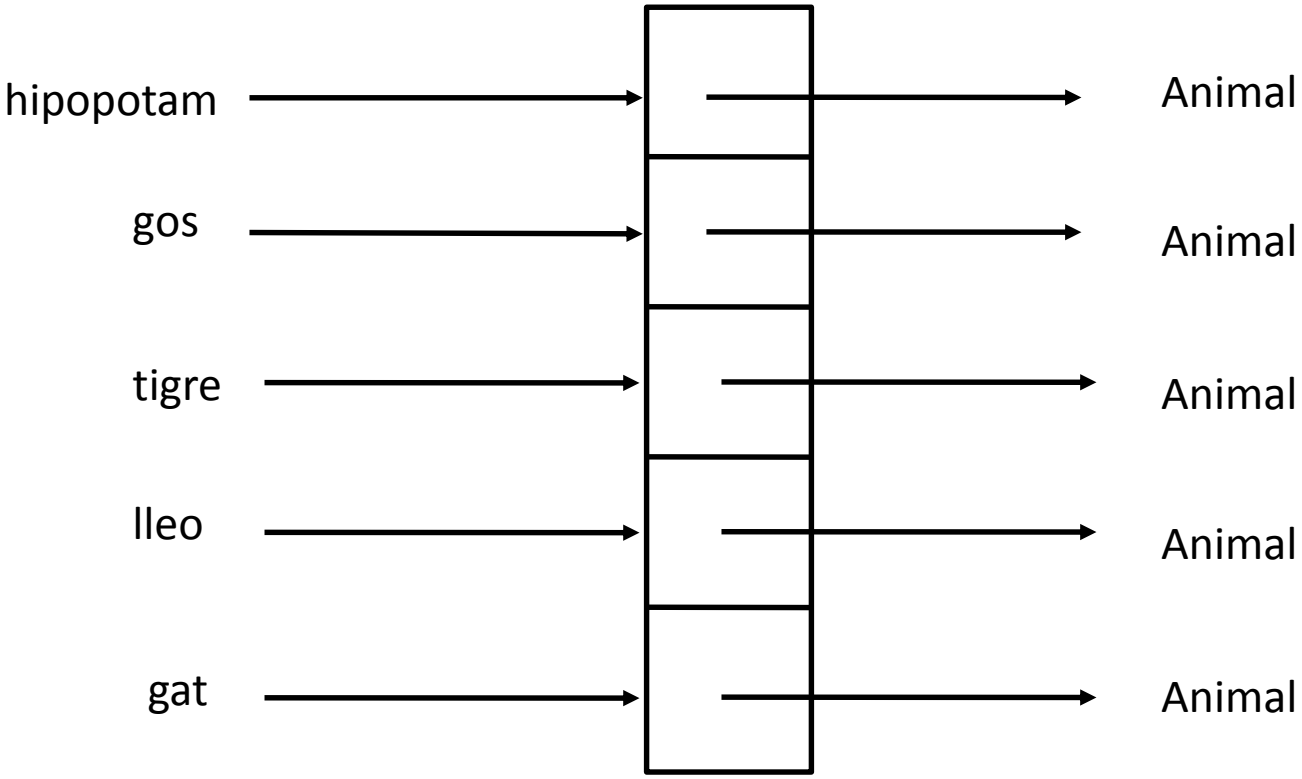
```
Cat c = new Cat();
System.out.println(c.toString());
```

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

Array polimòrfic

```
ArrayList<Animal> laLlistaAnimals = new ArrayList<Animal>();
```

Quan pot ser útil?



Exemple

```
package paquetInterficies;
import java.util.ArrayList;

public abstract class Animal {
    public abstract void ferSoroll();
}
```

Animal.java

Exemple

```
package paquetInterficies;
import java.util.ArrayList;

public class Gat extends Animal{
 public void ferSoroll(){
       System.out.println("miau");
   }
}
```

Gat.java

```
package paquetInterficies;
import java.util.ArrayList;

public class Gos extends Animal{
   public void ferSoroll(){
       System.out.println("guau");
   }
}
```

Gos.java

Exemple

```
package paquetInterficies;
import java.util.ArrayList;

public class TestAnimals {
   public static void main(String[] args){
      ArrayList<Animal> arrayAnimals = new ArrayList<Animal>();

      Gos gos = new Gos();
      Gat gat = new Gat();
      arrayAnimals.add(gos);
      arrayAnimals.add(gat);
      arrayAnimals.get(0).ferSoroll();
      arrayAnimals.get(1).ferSoroll();
   }
}
```

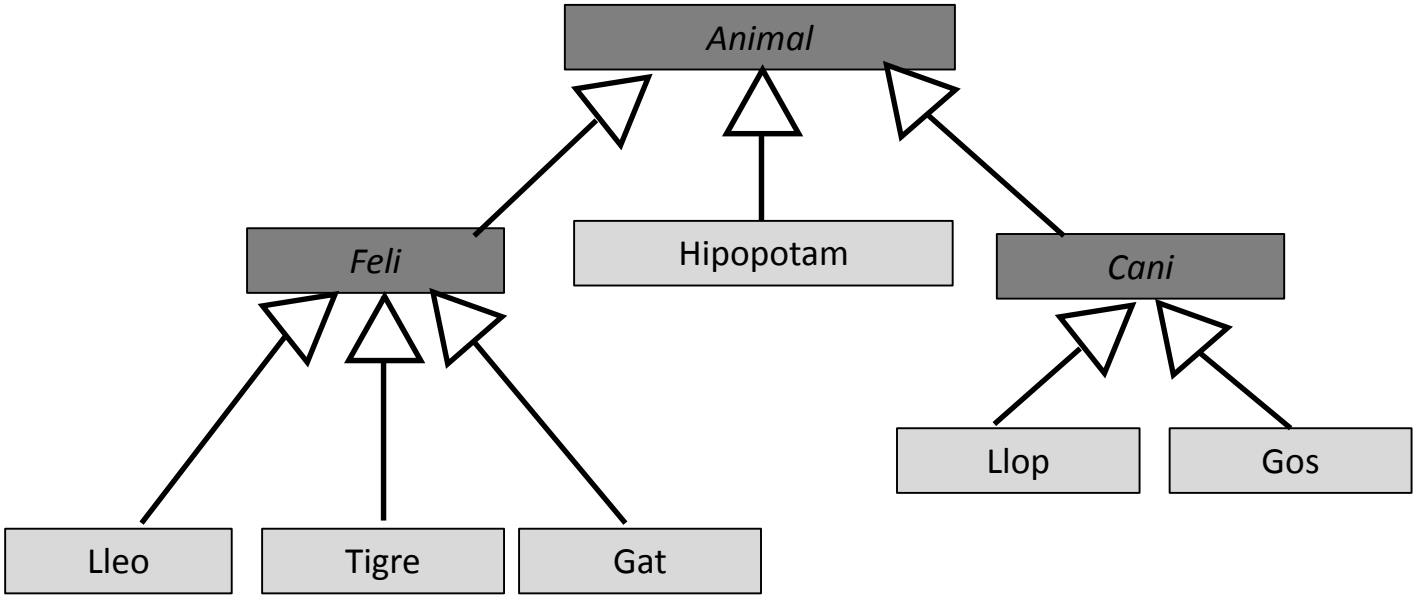
ferSoroll és un mètode polimòrfic

Sortida per pantalla:
guau
miau

TestAnimal.java

Volem afegir els comportaments de les mascotes. Possibles dissenys?

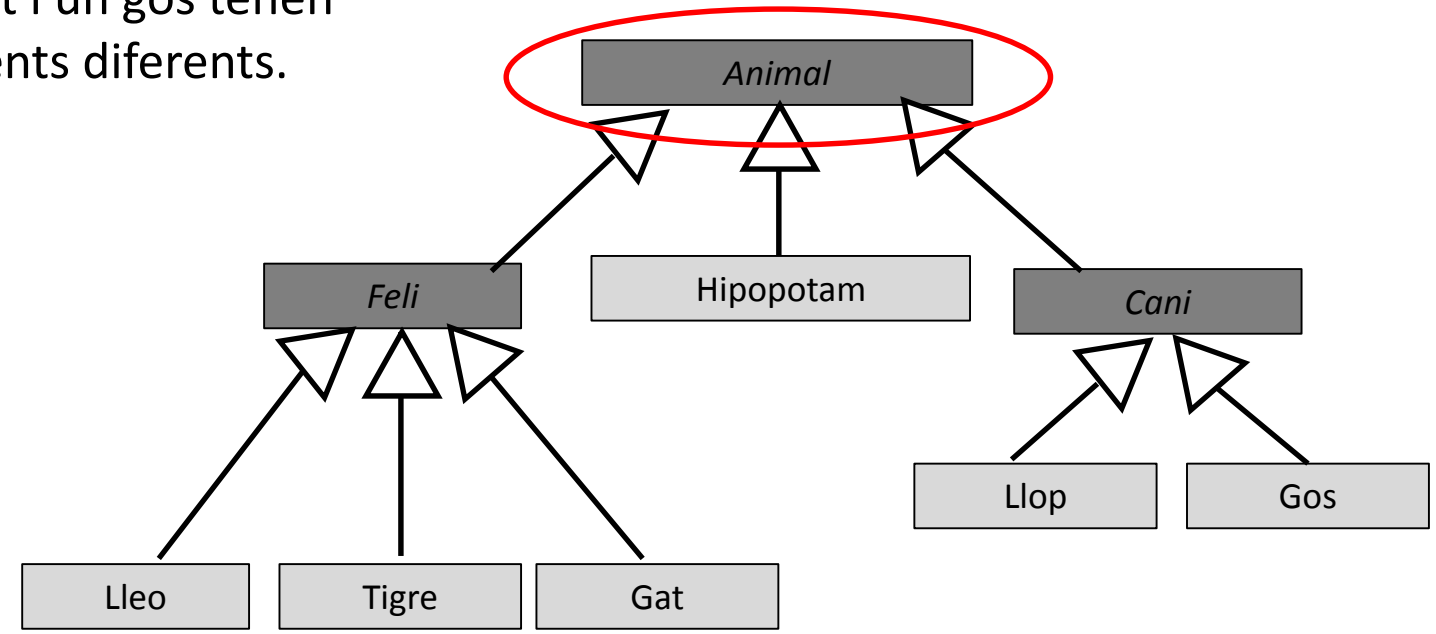
- Veiem diferents opcions de disseny per reutilitzar algunes de les classes existents en un programa d'una tenda de **mascotes**.



Opció 1

- Posem els mètodes de mascota en la classe Animal.

  Pros: No modifiquem les classes existents i les noves classes que afegim heretaran aquests mètodes.
  Contres: Un Hipopòtam no és una mascota!
  A més, un gat i un gos tenen comportaments diferents.

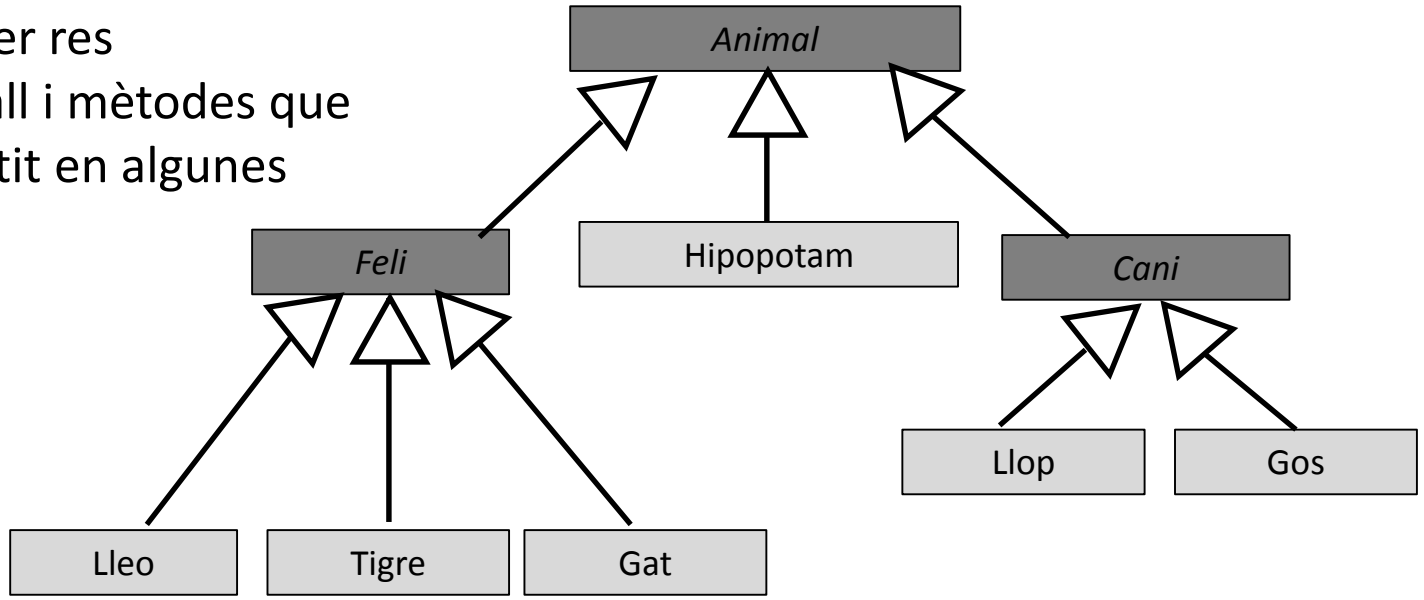


Opció 2

- Posem els mètodes de mascota en la classe Animal, però fem els **mètodes abstractes** forçant les subclasses de Animal a sobreescrire'ls.

Pros: Els mateixos que l'opció 1, però a més podem definir no-mascotes. Com? Fent que les implementacions no facin res.

Contres: S'han d'implementar tots els els mètodes abstractes de la classe Animal encara que sigui per no fer res
→ molt treball i mètodes que no tenen sentit en algunes classes.

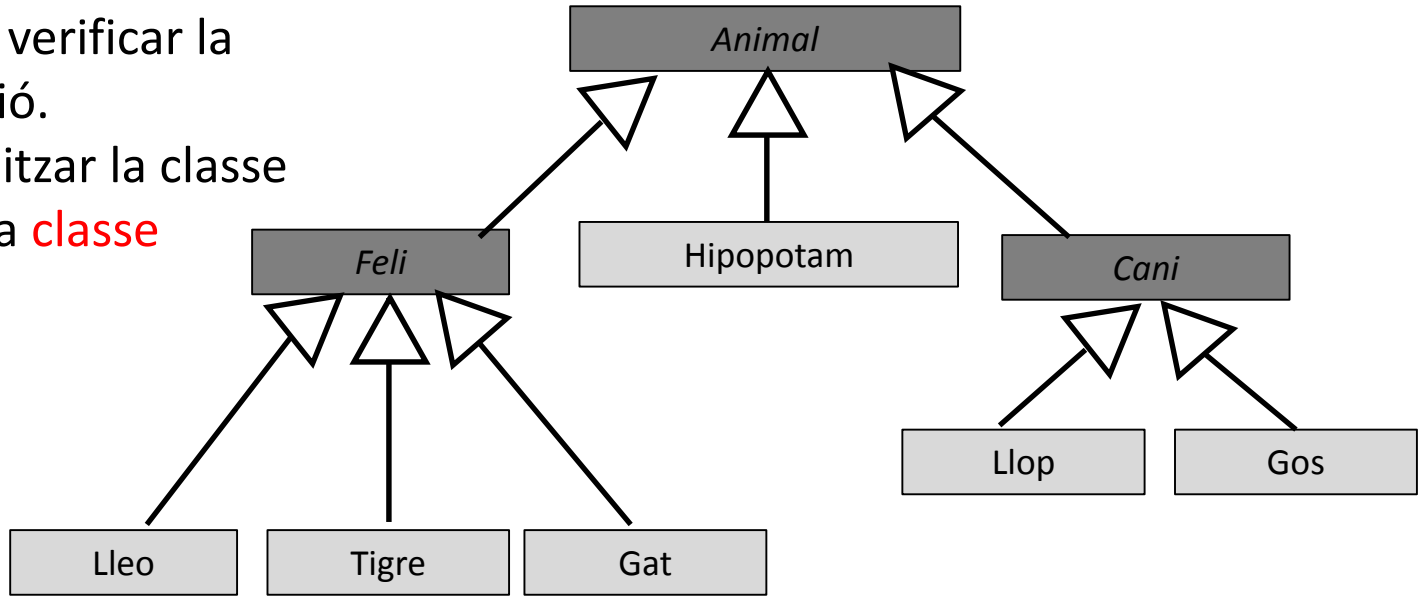


En aquest cas, només hauríem de posar dins de la classe Animal, els mètodes que s'apliquen a totes les seves subclasses.

Opció 3

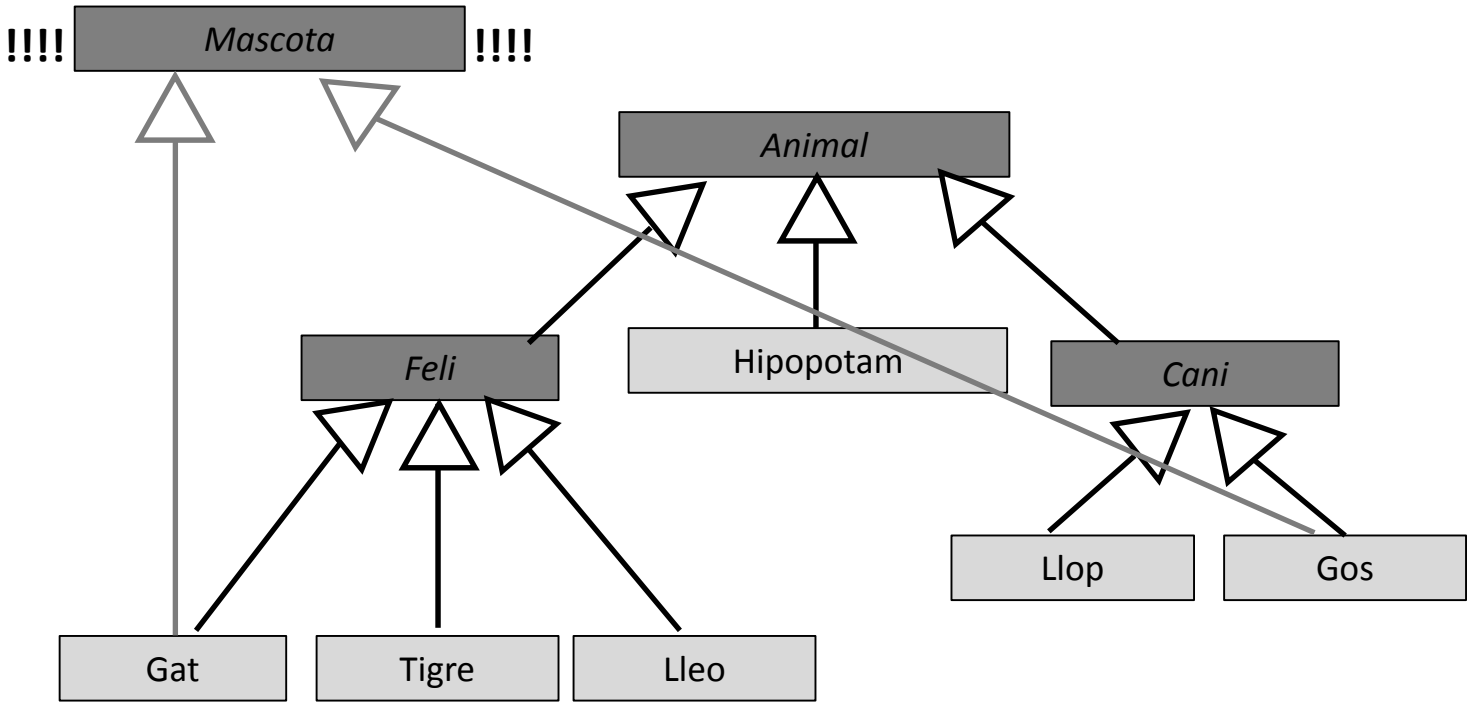
- Posem els mètodes de mascota només en les classes que ho són.

Pros: Desapareixen els hipopòtams com a mascotes i els mètodes estan on toca.
Contres: Tots els programadors hauran de conèixer el protocol.
No hi ha contracte que obliga el compilador a verificar la implementació.
No es pot utilitzar la classe Animal com la classe polimòrfica.



Necessitem dues superclasses

→ Herència múltiple



→En lloc de classes abstràctes, utilitzarem **interfícies**.

Interfícies

- Una interfície és un conjunt de **declaracions de mètodes** (sense definició)
- Una interfície també pot definir **constants** que són implícitament *public*, *static* i *final*, i sempre s'han d'inicialitzar en la declaració
- Totes les classes que implementen una interfície estan obligades a proporcionar una definició als mètodes de la interfície
- Una interfície defineix el protocol d'implementació d'una classe

Interfícies

- Una classe pot implementar més d'una interfície

→representa una alternativa a l'herència múltiple en Java.

- La paraula clau és:

implements + el nom de la interfície

interface *nom_interficie* {
  tipus_retorn nom_metode (*llista_arg*) ;
  . . .
}

class *nom_classe* **implements** *nom_interficie* {
  tipus_retorn nom_metode (*llista_arg*) {
    <codi>
  }
}

Implementació

```
public interface Mascota {
    public void serAmigable();
    public void jugar();
}
```

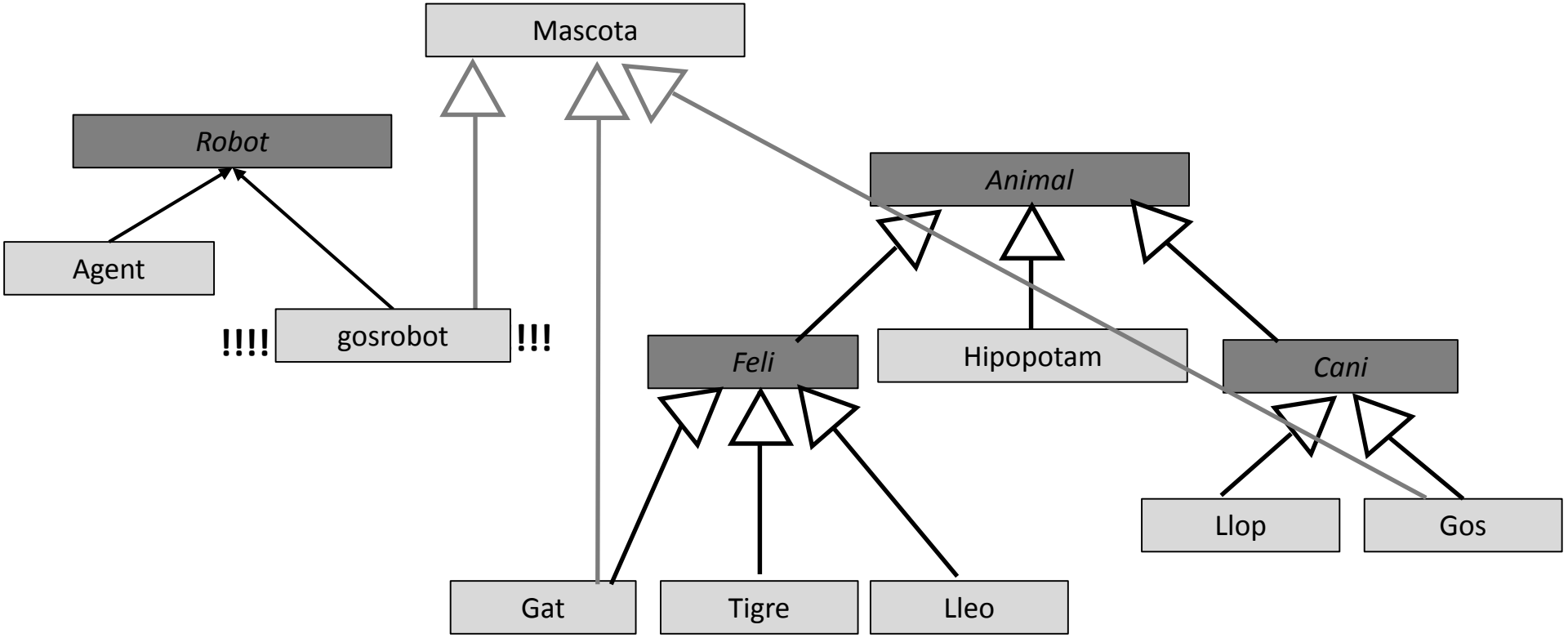
Mascota.java

Exemple

```
public class Gos extends Animal implements Mascota{
    public void ferSoroll(){
        System.out.println("guau");
    }
    public void serAmigable() {
        System.out.println("fa gràcies");
    }
    public void jugar() {
       System.out.println("juga");
    }
}
```

Gos.java

Classes de diferents arbres d'herència poden implementar la mateixa interfície



Interfície

Quan utilitzar una interfície en lloc d'una classe abstracta?

- Per la seva senzillesa es recomana utilitzar interfícies sempre que sigui possible.
- Si la classe ha d'incorporar atributs, o resulta interessant la implementació d'alguna de les seves operacions, llavors declarar-la com a classe abstracta.
- Dins la biblioteca de classes de **Java** es fa un ús intensiu de les interfícies per a caracteritzar les classes.

- Alguns exemples:
  - Per a que un objecte pugui ser guardat en un fitxer, la seva classe ha d'implementar la interfície *Serializable,*
  - Per a que un objecte sigui duplicable, la seva classe ha d'implementar la interfície *Cloneable,*
  - Per a que un objecte sigui ordenable, la seva classe ha d'implementar la interfície *Comparable.*

Extensió d'interfícies

- Les interfícies poden extendre altres interfícies
- La sintaxis es:

interface *nom_NovaInterficie* **extends** *nom_interficie* , . . . {
    tipus_retorn nom_metode (*llista_arguments*) ;
    . . .
}

Exemple: Interfícies

```
public interface VideoClip {
  // comença la reproducció del video
  void play();
  // reprodueix el clip en un bucle
  void bucle();
  // para la reproducció
  void stop();
}

//I una classe que implementa la interfície:
class LaClasse implements VideoClip {
 void play() { <codi> }
 void bucle(){ <codi> }
 void stop() { <codi> }
}
```

Exemple: Interfícies

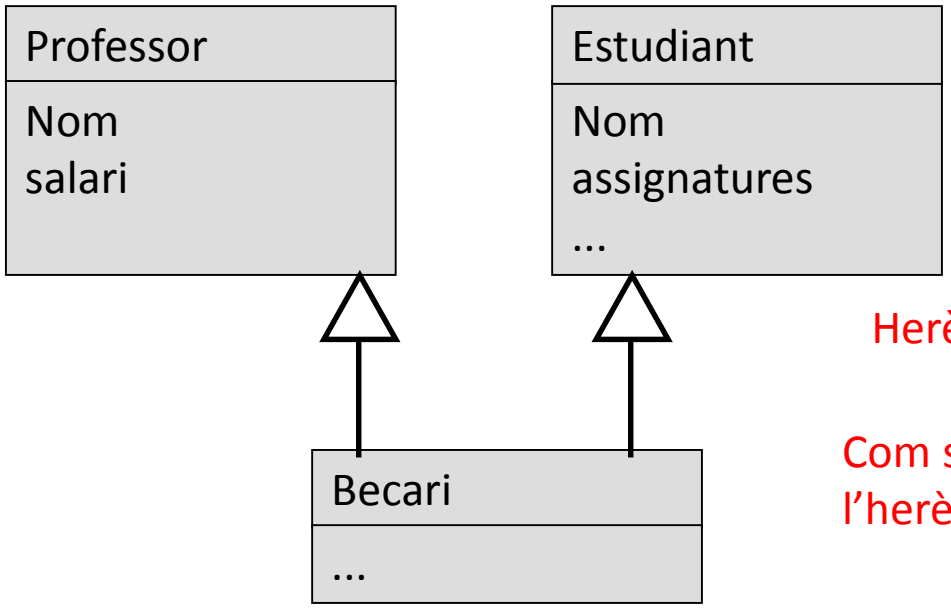
```
public interface VideoClip {
  // comença la reproducció del video
  void play();
  // reprodueix el clip en un bucle
  void bucle();
  // para la reproducció
  void stop();
}
```

//I una altra classe que també implementa la interfície:

```
Class LaAltraClasse implements VideoClip {
      void play()  { <codi nou> }
      void bucle() { <codi nou > }
      void stop()  { <codi nou > }
}
```

Interfície per herència múltiple

- Un exemple un poc més complex:
- Si volem implementar el següent disseny:

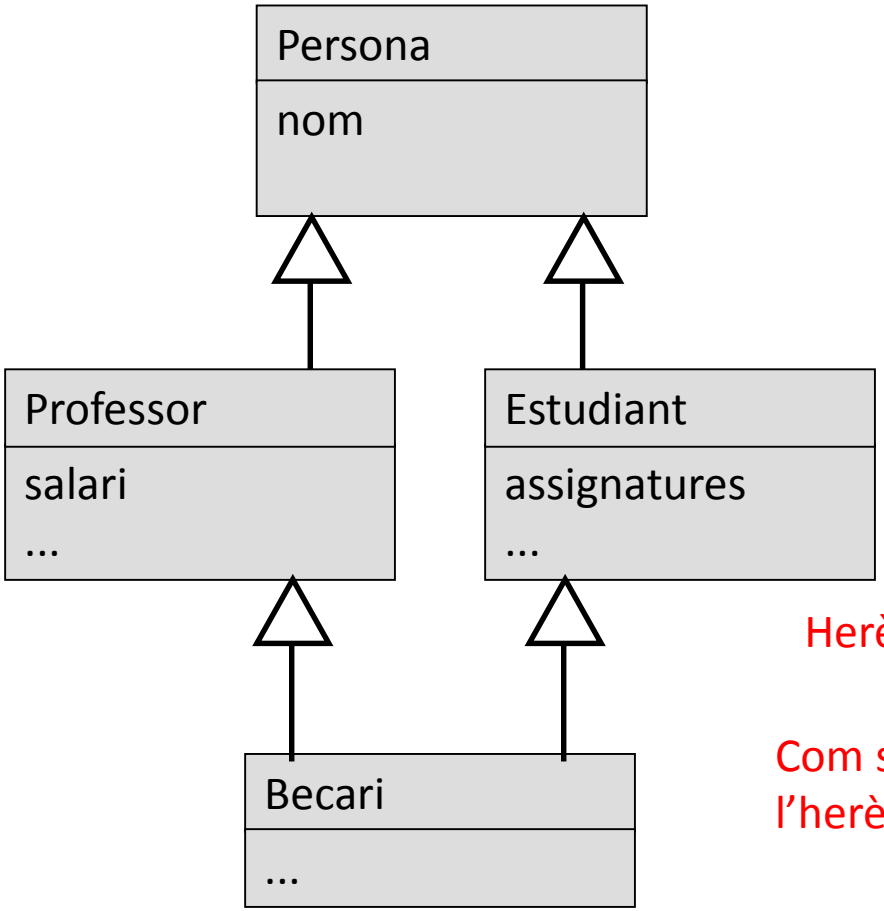


Herència múltiple

Com solucionem el problema de l'herència múltiple?

Interfície per herència múltiple

- Un exemple un poc més complex:
- Si volem implementar el següent disseny:



Herència múltiple

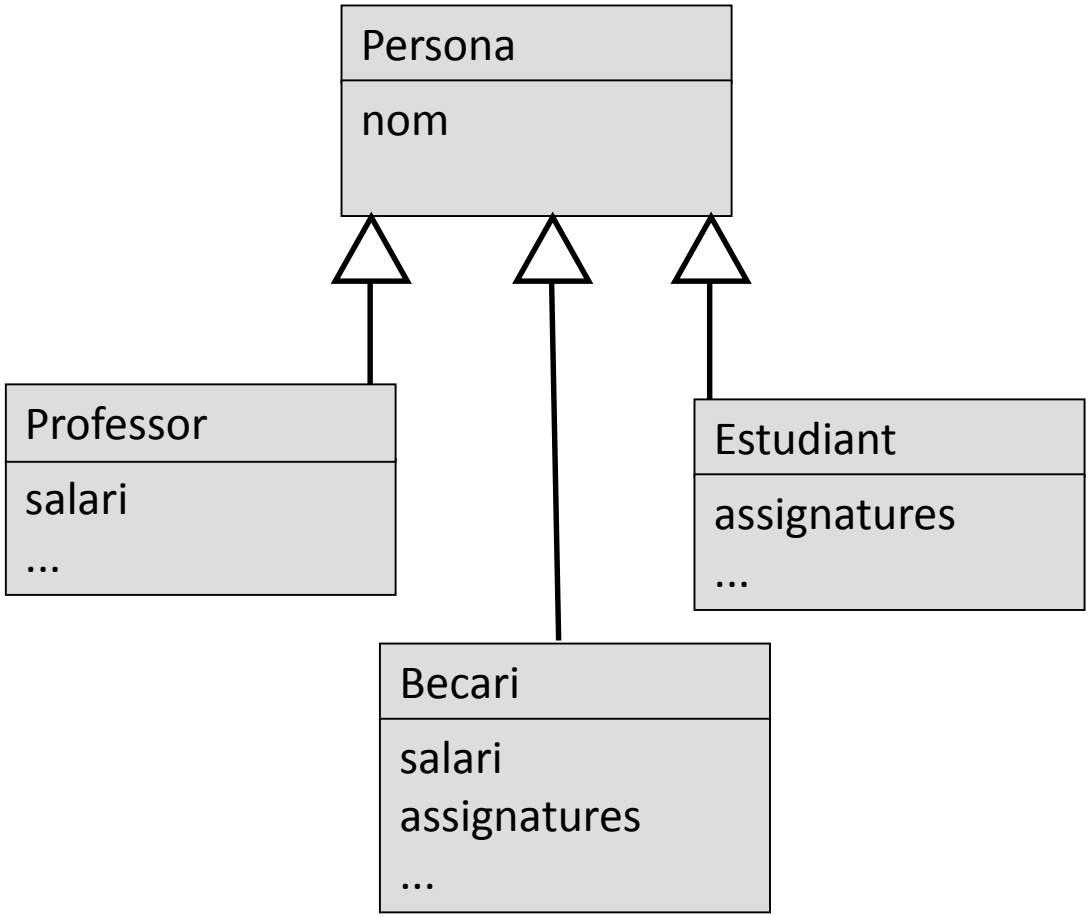
Com solucionem el problema de l'herència múltiple?

Observacions

- O simplifiquem el disseny o utilitzem interfícies per solucionar aquest problema.
- Solució Standard:
  - Una classe per heretar
  - Una interfície per implementar
- Fent servir interfícies, hi ha diverses opcions d'implementació.

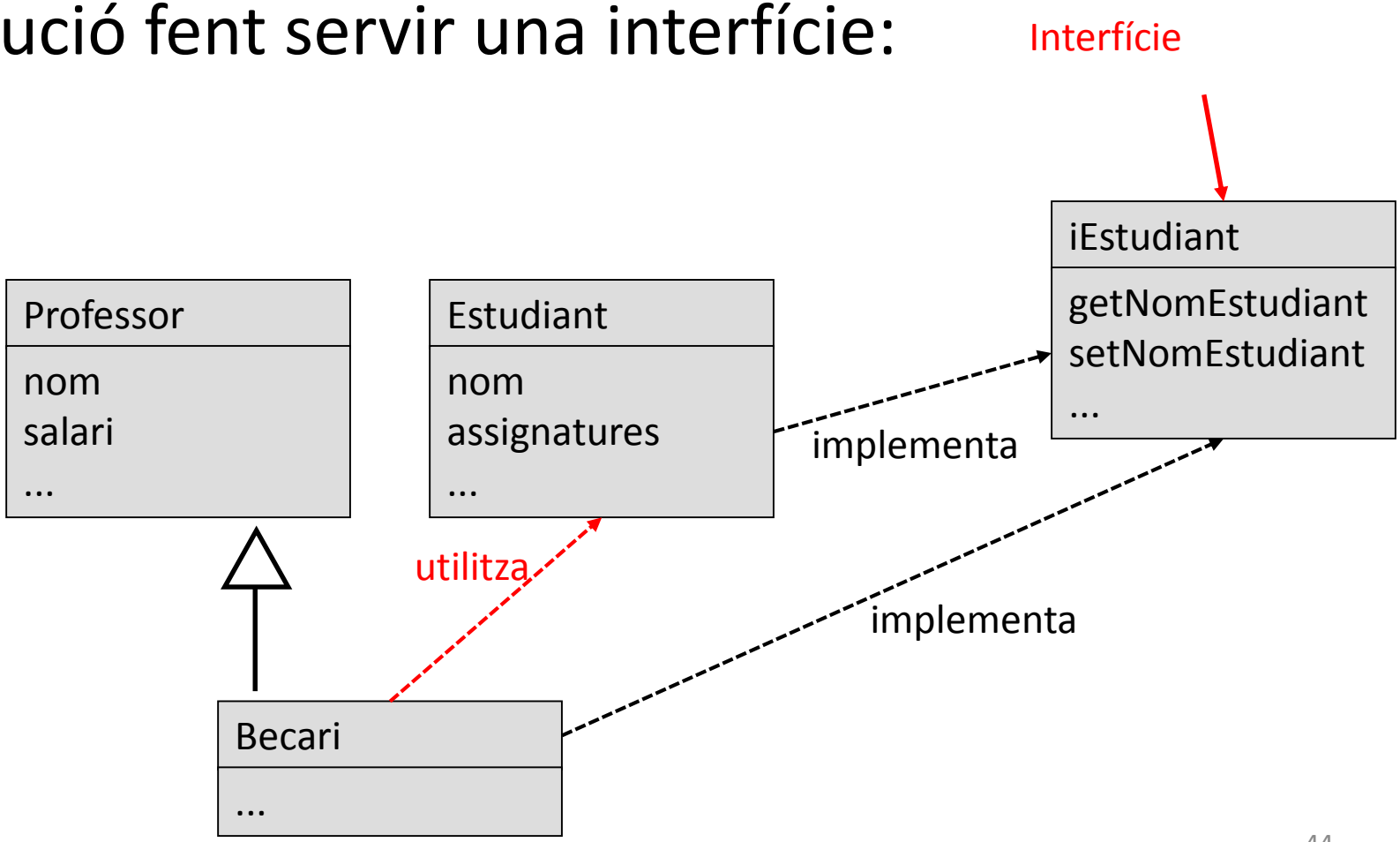
Interfície per herència múltiple

- Solució fent servir un nou disseny:



Interfície per herència múltiple

- Solució fent servir una interfície:



Referències

- Bertrand Meyer, “**Construcción de software orientado a objetos”, Prentice Hall, 1998.**

- “Software Architecture and UML” de Grady Booch (Rational Software). Presentació P. Letelier.

- Bert Bates, Kathy Sierra. **Head First Java**. O’Reilly Media, 2005.