GRAU D'ENGINYERIA INFORMÀTICA

PROGRAMACIÓ II
CURS 12-13

Bloc 2:
Programació Orientada a Objectes (4)

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

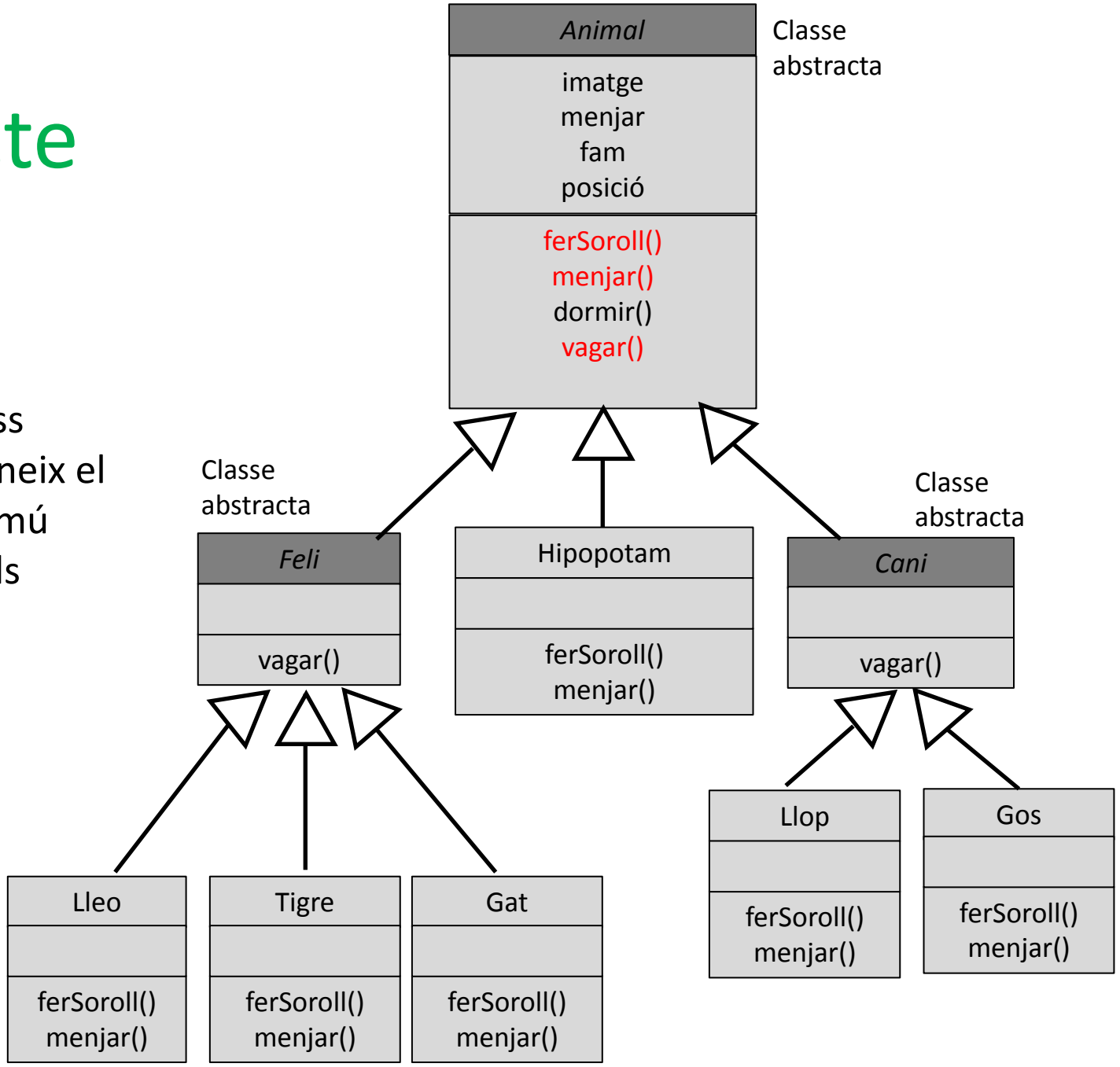
INTERFÍCIES

Introducció

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

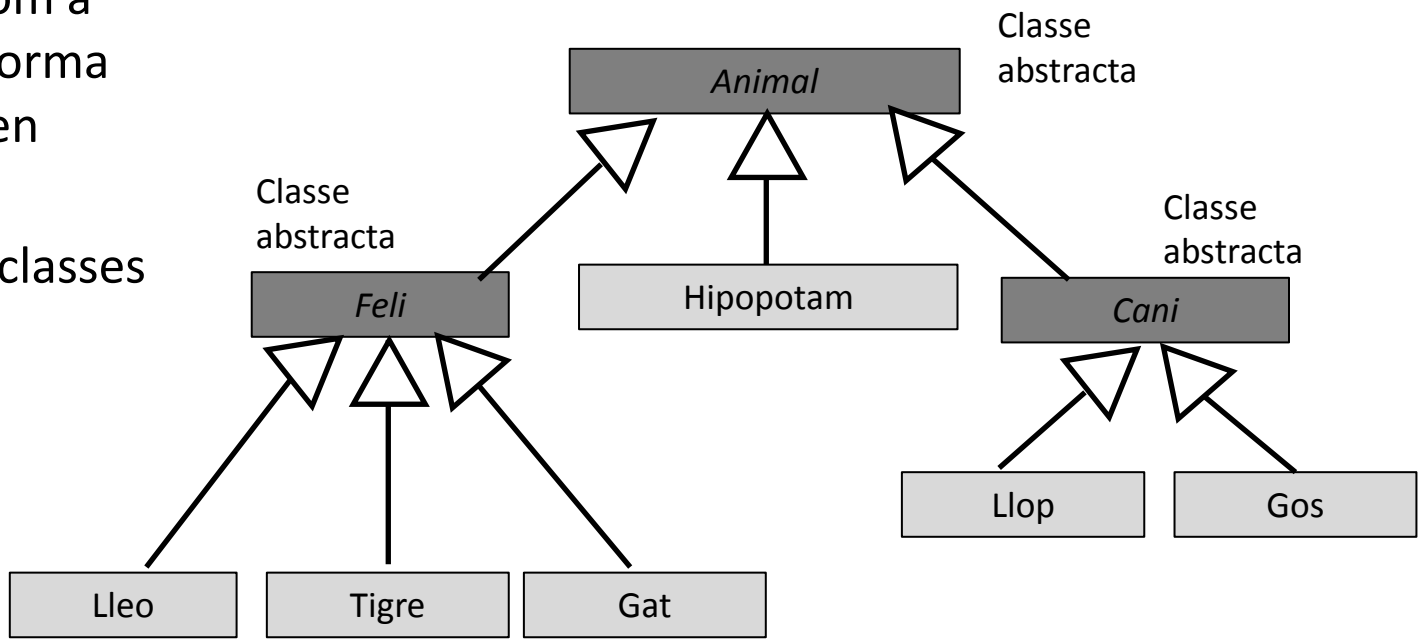
Contracte

- Comencem definint un contracte:
  - La superclass Animal defineix el protocol comú per a tots els animals.



Contracte

- 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);
    }
}
```

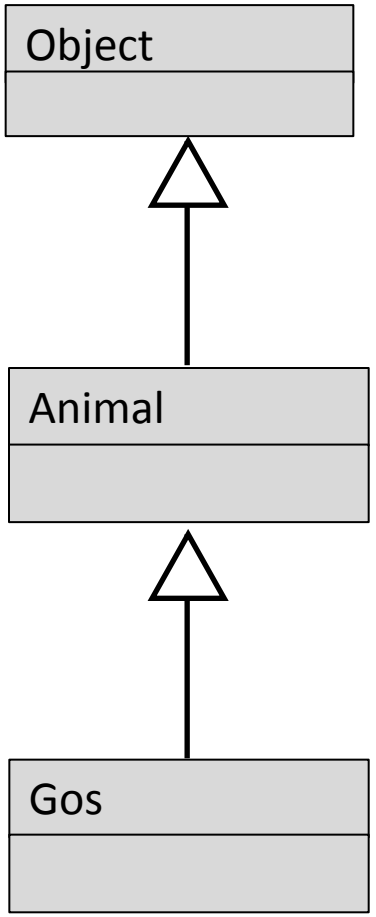
TestLlistaAnimals.java

Estem afegint tot tipus d'animals a l'array

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

Llista polimòrfica

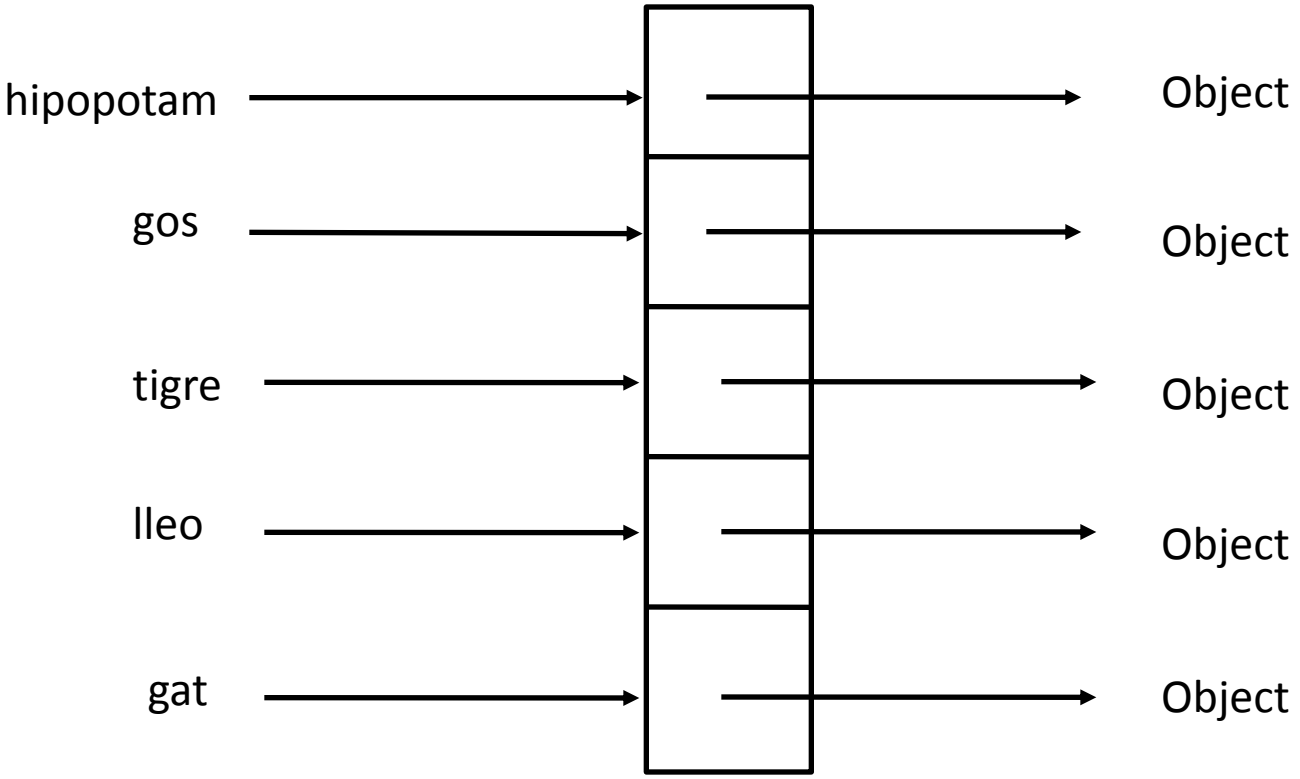
- També es podria optar per fer servir la classe Object que és encara més genèrica i referenciar a qualsevol tipus d'objectes.
- Però això porta alguns inconvenients!!!



Llista polimòrfica

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

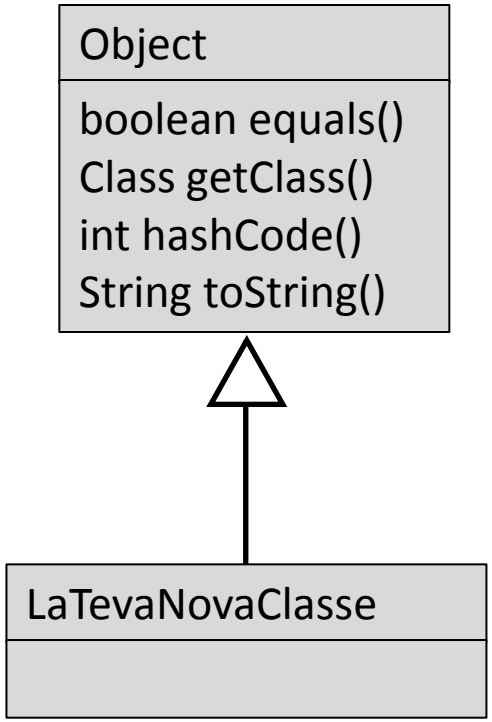
Gos gos = laLlistaAnimals.get(0); No compilarà!



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()

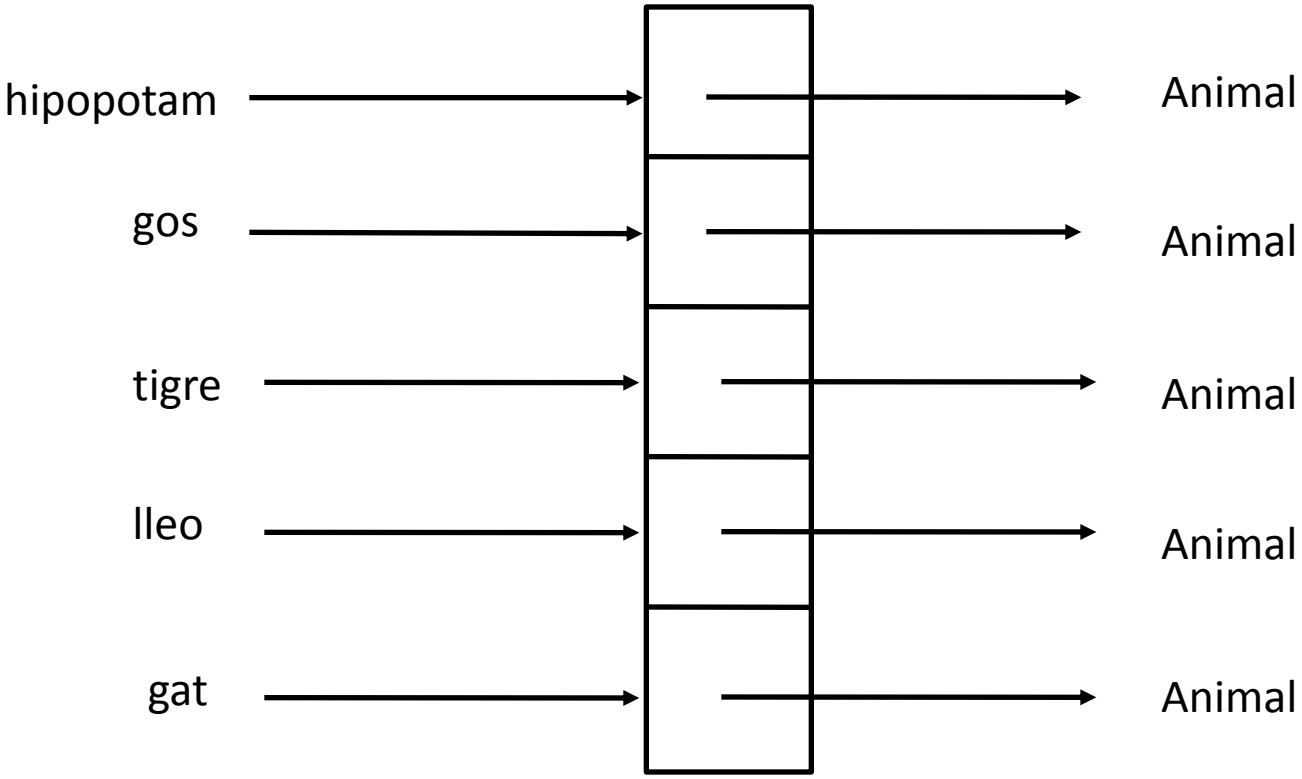
```
Cat c = new Cat();
System.out.println(c.toString());
```

```
% 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();
   }
}
```

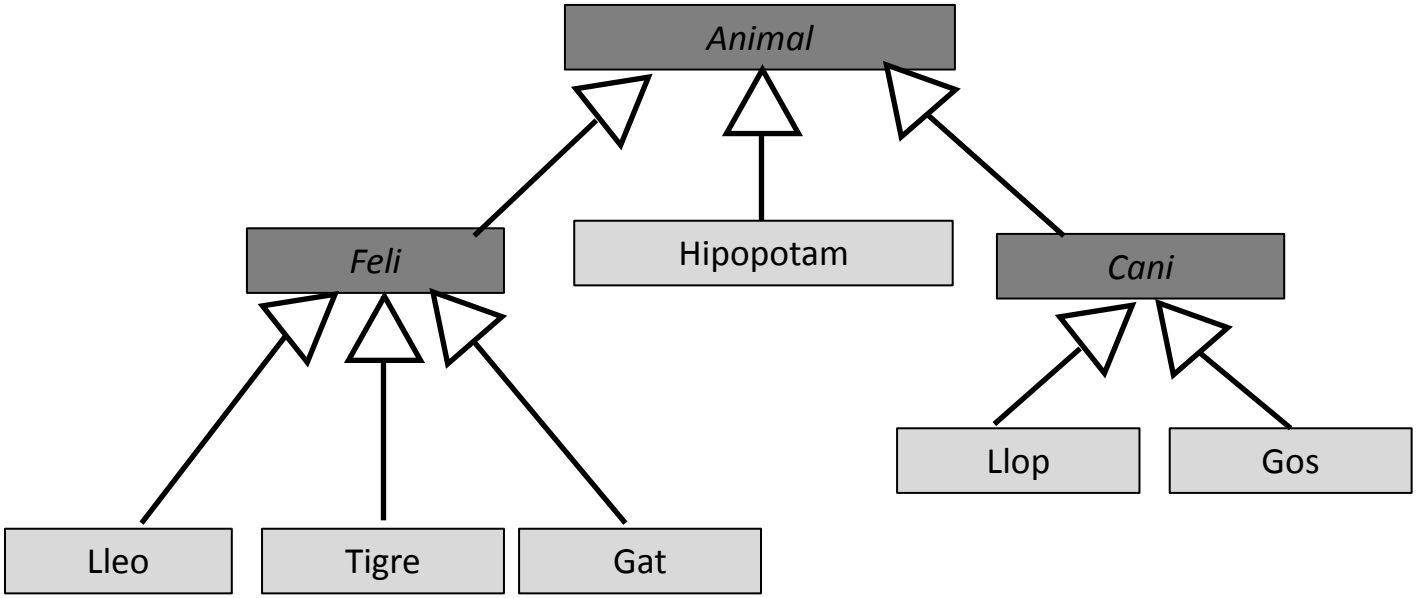
ferSoroll és un mètode polimòrfic

Sortida per pantalla:
guau
miau

TestAnimal.java

Possibles dissenys:
Volem afegir els comportaments de les mascotes

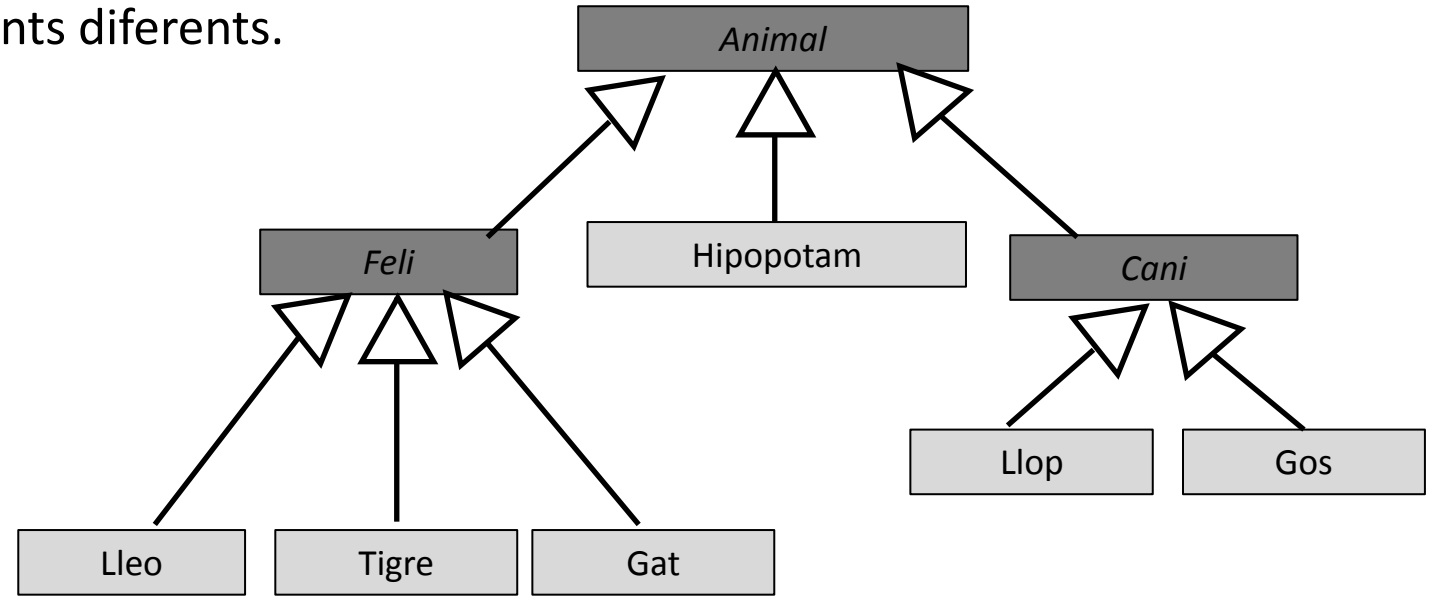
- Veiem diferents opcions de disseny per reutilitzar algunes de les classes existents en un programa d'una tenda de **mascotes**.



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.

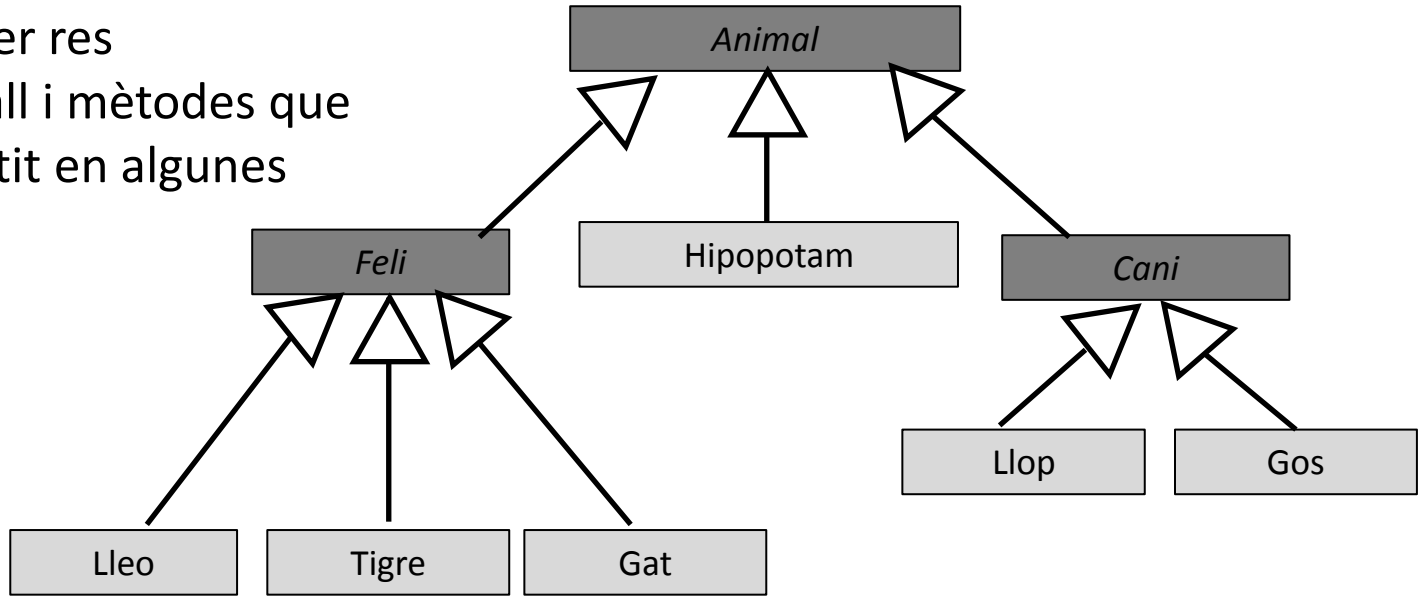


Opció 2

- 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.

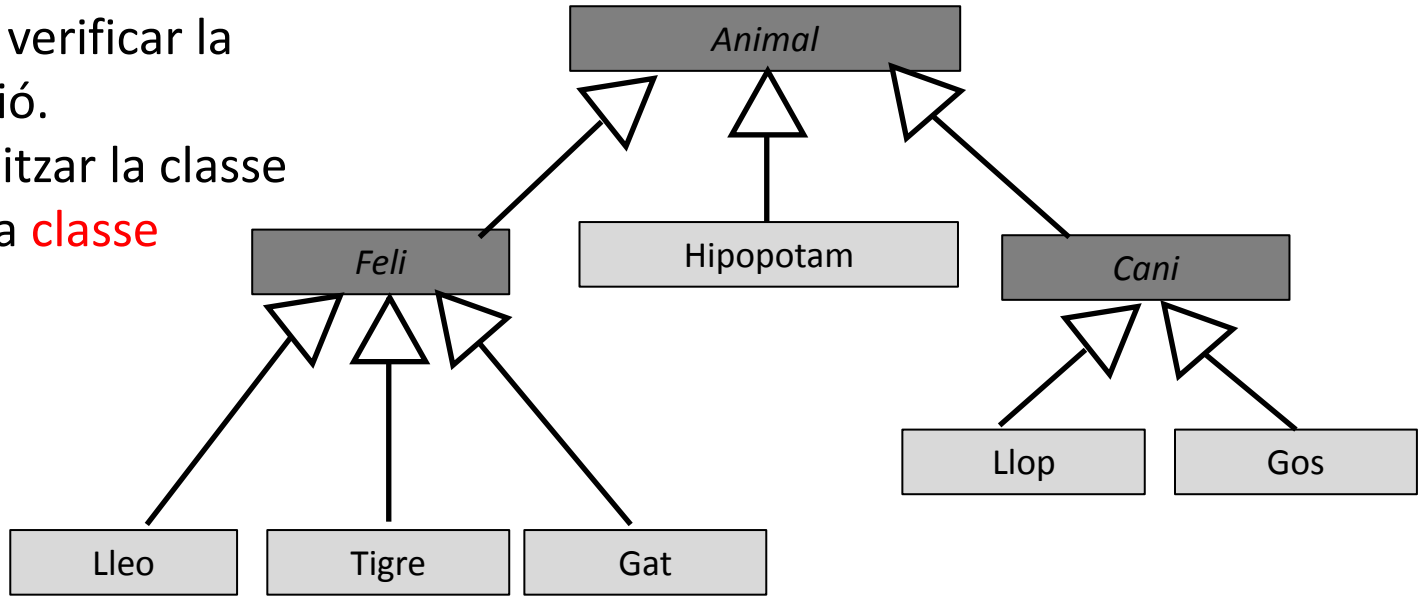


En aquest cas, només hauríem de posar dins de la classe Animal, els mètodes que s'apliquen a totes les seves subclasses.

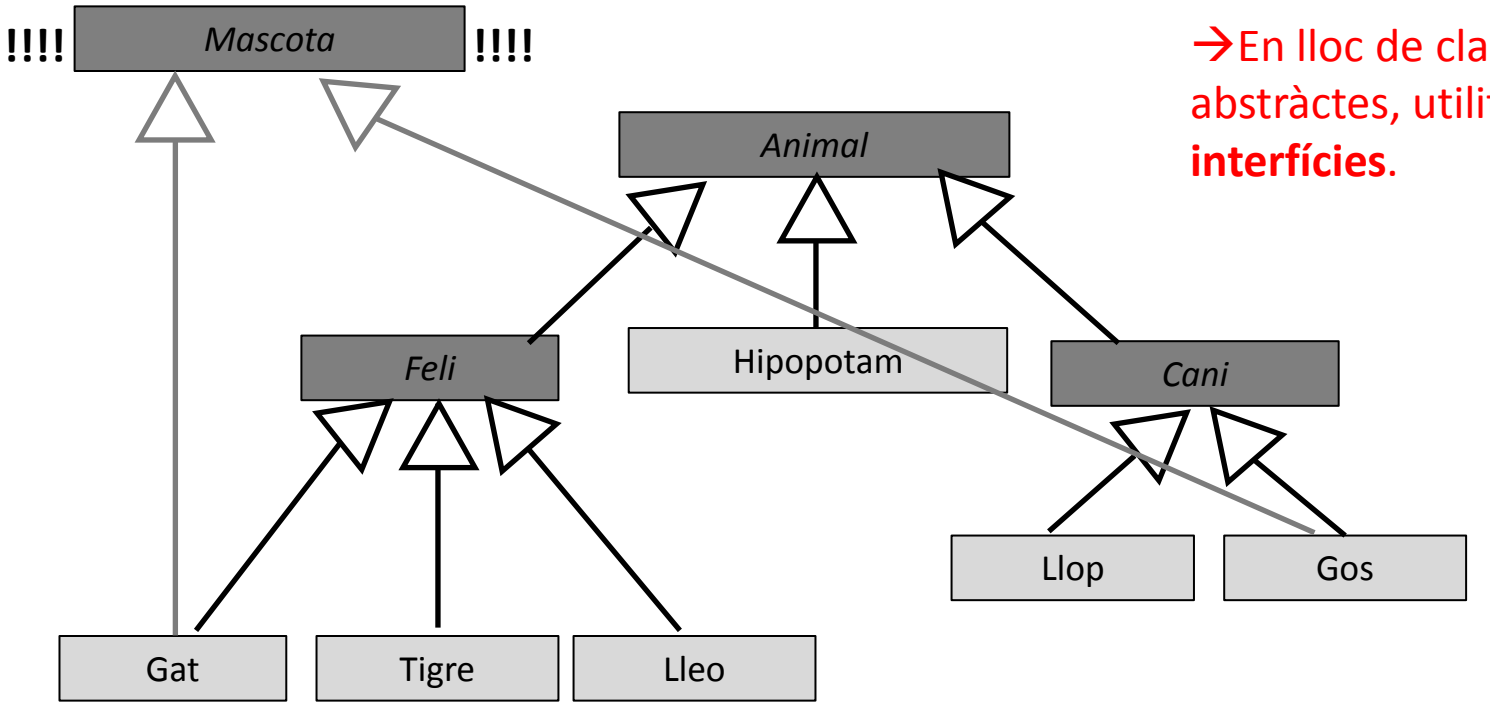
Opció 3

- 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



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

→ Herència múltiple

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 abstract void serAmigable();
    public abstract void jugar();
}
```

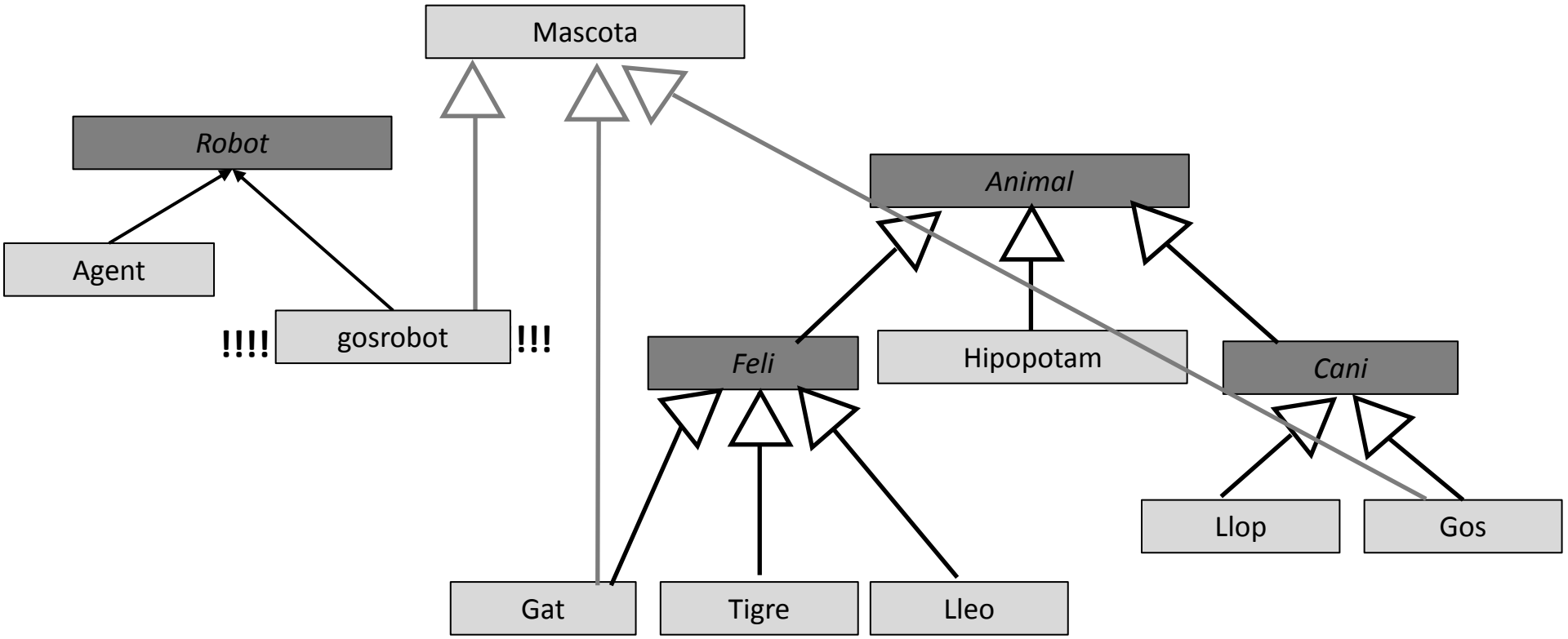
Mascota.java

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

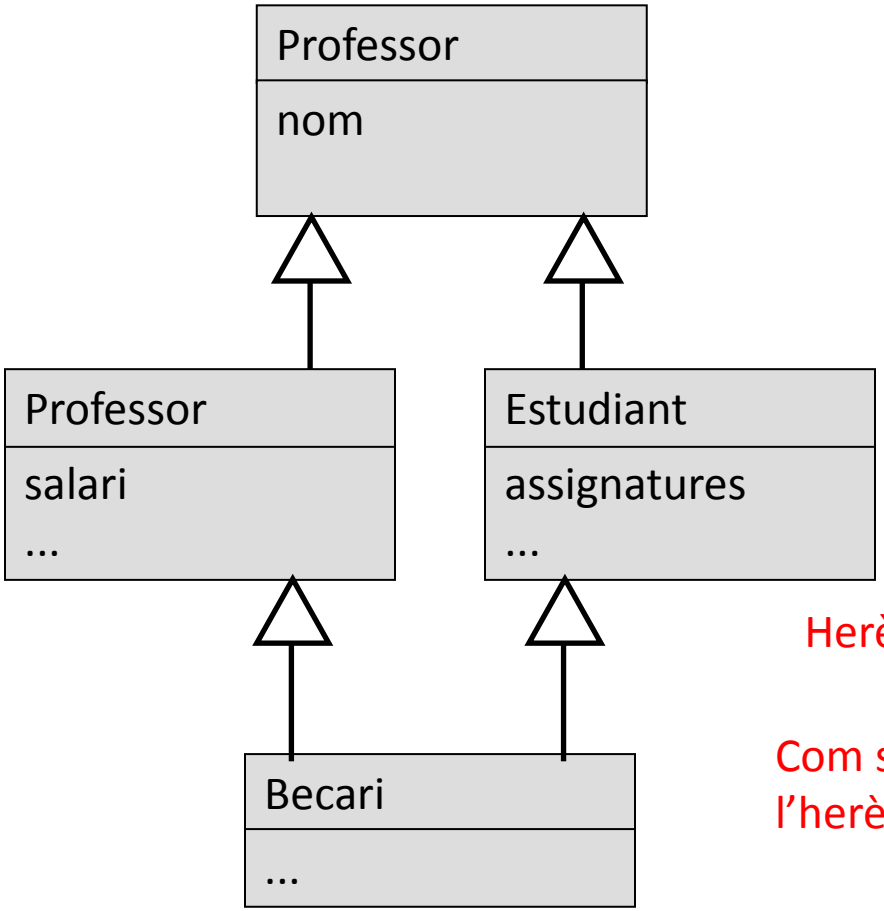
```
public interface VideoClip {
  // comença la reproducció del video
  void play();
  // reprodueix el clip en un bucle
  void bucle();
  // para la reproducció
  void stop();
}
```

//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

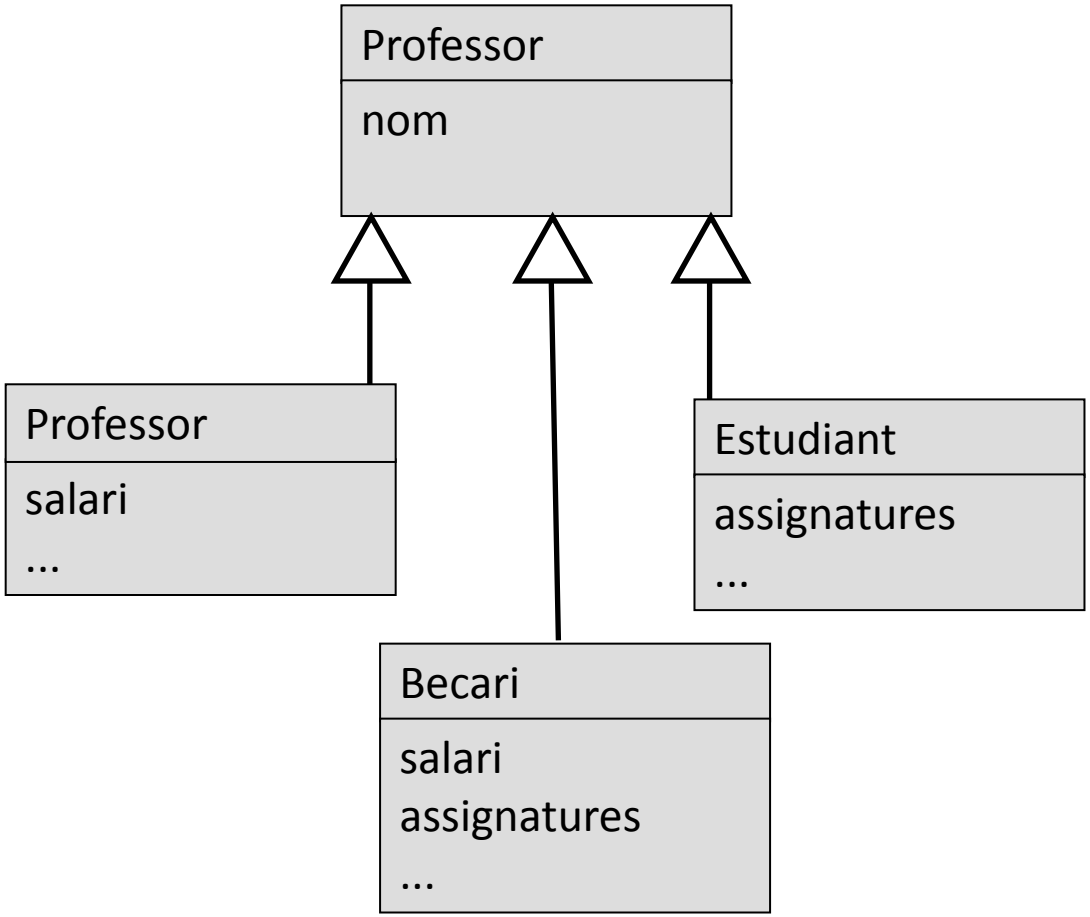
Com solucionem el problema de l'herència múltiple?

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

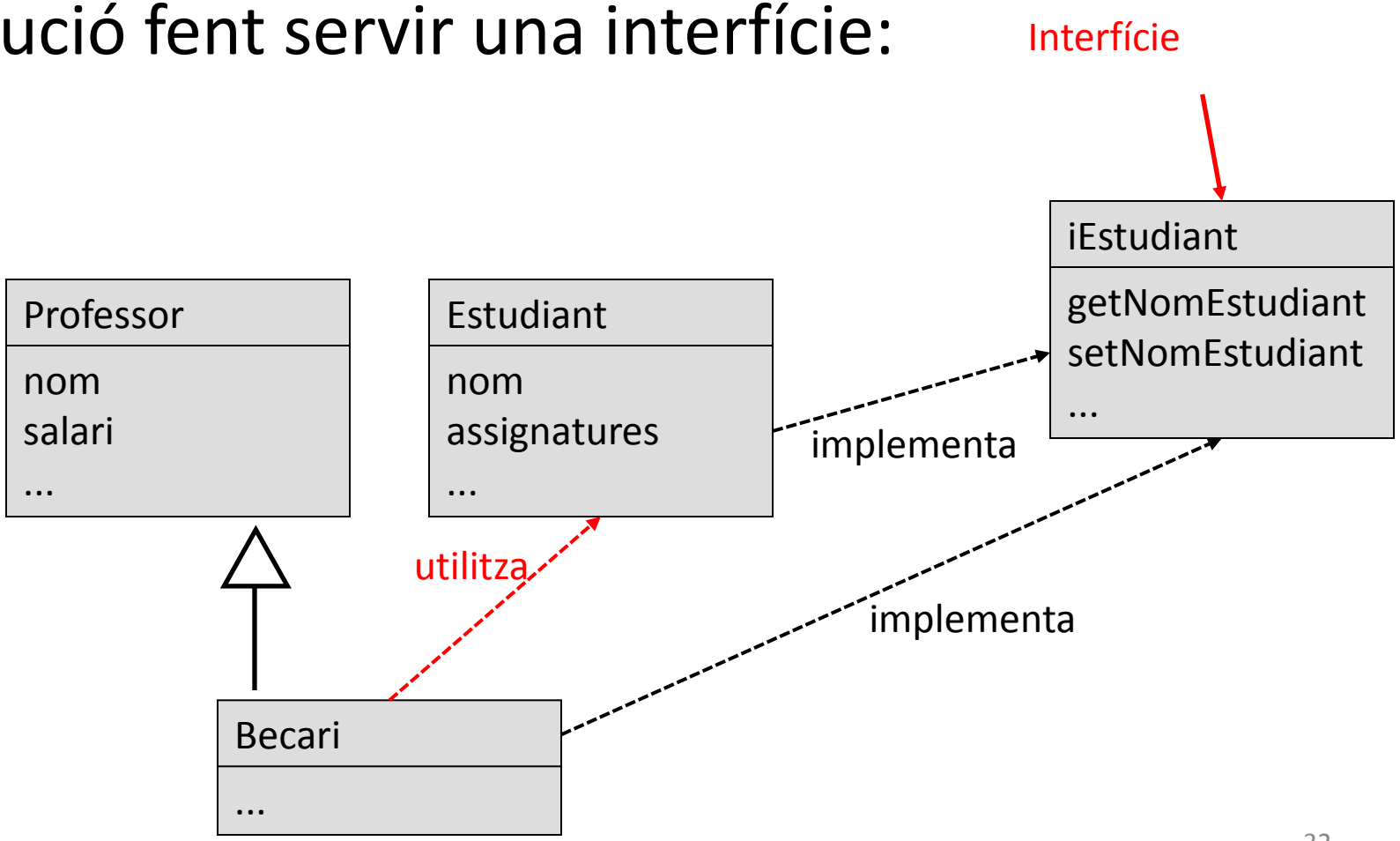
Interfície per herència múltiple

- Solució fent servir un nou disseny:



Interfície per herència múltiple

- Solució fent servir una interfície:



Solució 1: Exemple Interfícies

```
public class Professor{
	private String nom;
	private int salari;
	public Professor(String pNom, int pSalari) {
		nom = pNom;
		salari = pSalari;
	}
	public String getNom() {
		return nom;
	}
	public int getSalari() {
		return salari;
	}
}
```

Professor.java

Solució 1: Exemple Interfícies

```
public interface IEstudiant {

    public String getNomEstudiant ();
    public void setNomEstudiant (String pNom);

}
```

IEstudiant.java

Solució 1: Exemple Interfícies

```
public class Estudiant implements IEstudiant {

    private String nom;
    public Estudiant(String pNom) {
        nom = pNom;
    }
    public String getNomEstudiant () {
        return nom;
    }
    public void setNomEstudiant (String nom) {
        this.nom = nom;
    }
}
```

Estudiant.java

Solució 1: Exemple Interfícies

```
public class Becari extends Professor implements IEstudiant {


    private Estudiant estudiant;

    public Becari(String nom, int salari) {
        super(nom, salari);
        estudiant = new Estudiant(nom);
    }
    public String getNomEstudiant() {
        return estudiant.getNomEstudiant();
    }
    public void setNomEstudiant(String nom) {
        estudiant.setNomEstudiant(nom);
    }
}
```

Becari.java

Defineix un objecte de la classe Estudiant

Observacions

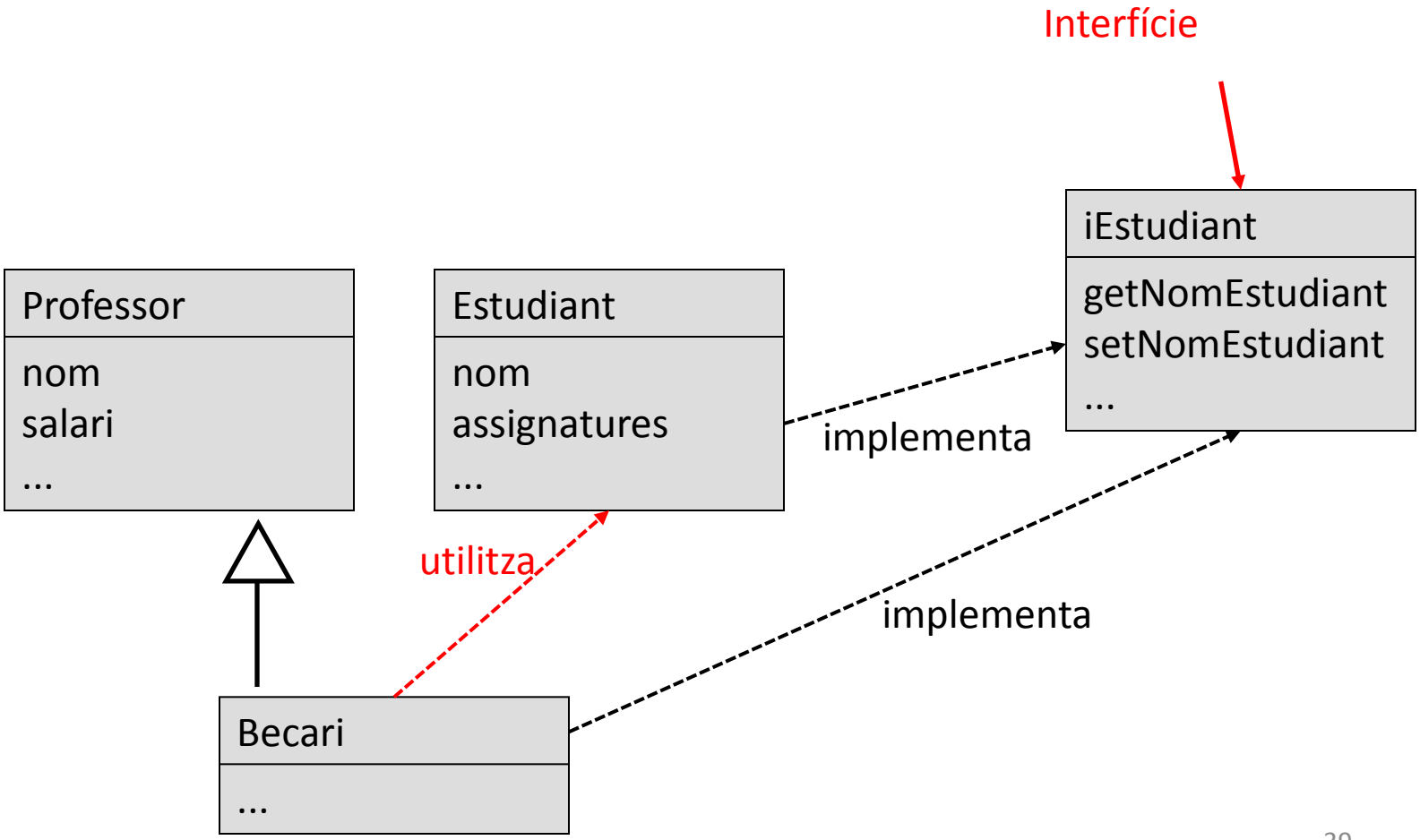
- Problema d'aquesta implementació:
  - Si canviem el nom de l'estudiant, el nom del professor no canvia

Solució 1: Exemple Interfícies

```
public class TestBecari {
 public static void main(String[] args) {

       Becari becari;
       becari = new Becari("Joan",1000);
       System.out.println("salari becari = " + becari.getSalari());

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

       becari.setNomEstudiant("Joan Francesc");

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

   }
}
```

Solució 2: Exemple Interfícies

- Solució fent servir una interfície:



Solució 2: Exemple Interfícies

```
public class Professor{
        private String nom;
        private int salari;
        public Professor(String pNom, int pSalari) {
                nom = pNom;
                salari = pSalari;
        }
        public String getNom() {
                return nom;
        }
        public int getSalari() {
                return salari;
        }
        public void setNom(String nom) {
          this.nom=nom;
        }
        public void setSalari(int salari) {
          this.salari=salari;
        }
}
```

Professor.java

Solució 2: Exemple Interfícies

```
public interface IEstudiant {

    public String getNomEstudiant ();
    public void setNomEstudiant (String pNom);
    public String getAssignatures();
    public void setAssignatures(String assignatures);

}
```

IEstudiant.java

Solució 2: Exemple Interfícies

```
public class Estudiant implements IEstudiant {

        private String nom;
        private String assignatures;
        public Estudiant(String pNom) {
                nom = pNom;
        }
        public String getNomEstudiant () {
                return nom;
        }
        public void setNomEstudiant (String nom) {
                this.nom = nom;
        }
        public String getAssignatures () {
                return assignatures;
        }
        public void setAssignatures (String assignatures) {
                this.assignatures = assignatures;
        }
}
```

Estudiant.java

Solució 2: Exemple Interfícies

```
public class Becari extends Professor implements IEstudiant {
      private Estudiant estudiant;
       public Becari(String nom, int salari) {
          super(nom, salari);
          estudiant = new Estudiant(nom);
       }
       public String getNomEstudiant() {
              return super.getNom();
       }
       public void setNomEstudiant(String nom) {
              super.setNom(nom);
       }
      public String getAssignatures() {
        return estudiant.getAssignatures();
      }
      public void setAssignatures(String assignatures) {
         estudiant.setAssignatures(assignatures);
      }
}
```

Becari.java

Solució 2: Exemple Interfícies

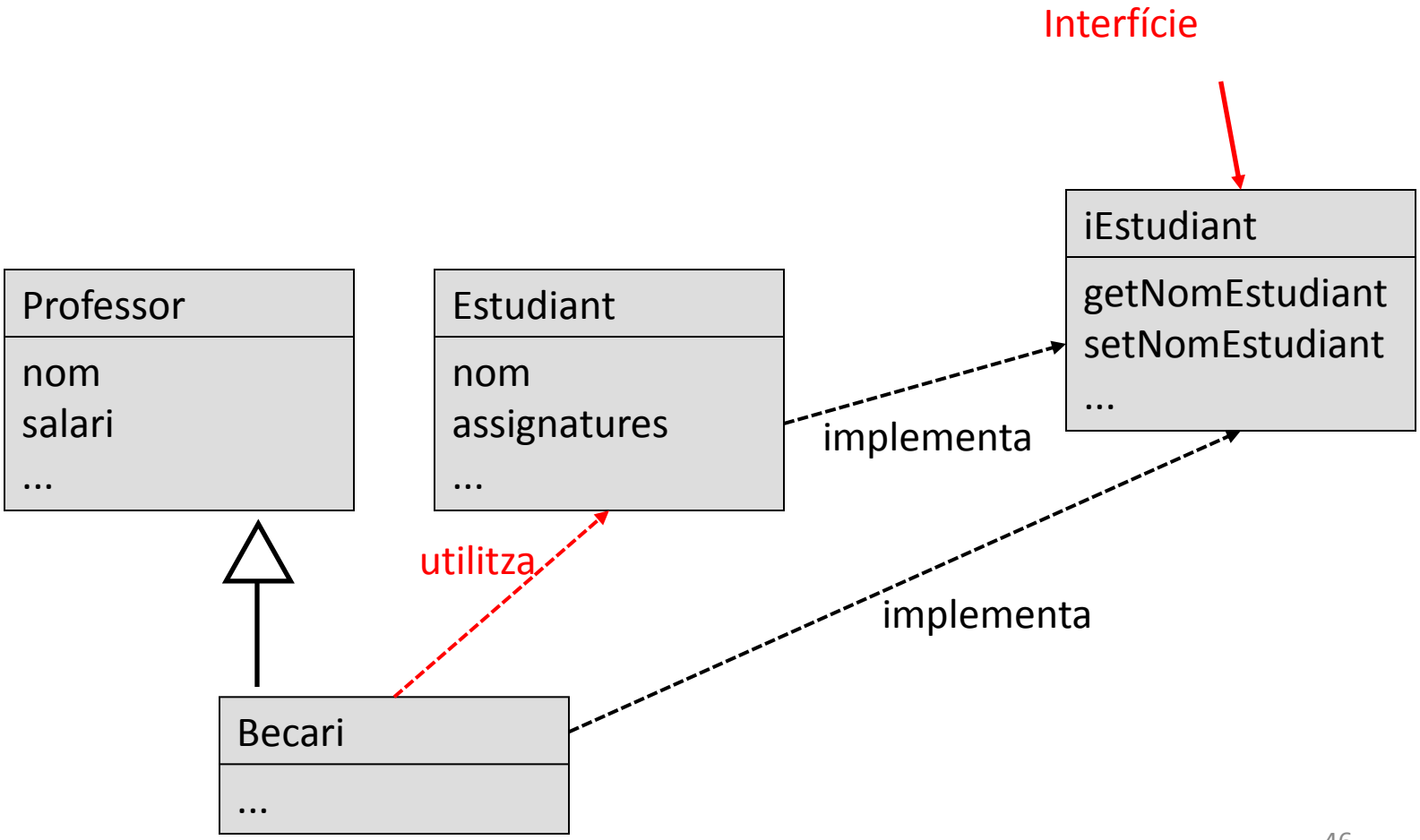
```
public class TestBecari {
 public static void main(String[] args) {

       Becari becari;
       becari = new Becari("Joan",1000);
       System.out.println("salari becari = " + becari.getSalari());

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

       becari.setNomEstudiant("Joan Francesc");

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

   }
}
```

Observacions

- Problema d'aquesta implementació:
  - L'objecte becari té dos mètodes per accedir al nom un és getNom() i l'altre getNomEstudiant()
- Hi ha una altra opció de disseny per evitar el problema de l'herència múltiple?

Solució 3: Exemple Interfícies

- Solució fent servir una interfície:



Solució 3: Exemple Interfícies

```
public class Professor{
        private String nom;
       private int salari;
       public Professor(String pNom, int pSalari) {
                nom = pNom;
                salari = pSalari;
       }
       public String getNom() {
                return nom;
       }
       public int getSalari() {
                return salari;
       }
      public void setNom(String nom) {
         this.nom=nom;
       }
       public void setSalari(int salari) {
         this.salari=salari;
       }
}
```

Professor.java

Solució 3: Exemple Interfícies

```
public interface IEstudiant {

      public String getNom();
      public void setNom(String pNom);
       public String getAssignatures();
      public void setAssignatures(String assignatures);
}
}
```

IEstudiant.java

Solució 3: Exemple Interfícies

```
public class Estudiant implements IEstudiant {

        private String nom;
        private String assignatures;

        public Estudiant(String pNom) {
                nom = pNom;
        }
        public String getNom() {
                return nom;
        }
        public void setNom(String nom) {
                this.nom = nom;
        }
        public String getAssignatures () {
                return assignatures;
        }
        public void setAssignatures (String assignatures) {
                this.assignatures = assignatures;
        }
}
```

Estudiant.java

Solució 3: Exemple Interfícies

```
public class Becari extends Professor implements IEstudiant {

    private Estudiant estudiant;

     public Becari(String nom, int salari) {
        super(nom, salari);
        estudiant = new Estudiant(nom);
     }

     public String getAssignatures() {
       return estudiant.getAssignatures();
    }

    public void setAssignatures(String assignatures) {
        estudiant.setAssignatures(assignatures);
    }
```

Becari.java

Solució 3: Exemple Interfícies

```
public class TestBecari {
 public static void main(String[] args) {

       Becari becari;
       becari = new Becari("Joan",1000);
       System.out.println("salari becari = " + becari.getSalari());

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

       becari.setNomEstudiant("Joan Francesc");

       System.out.println("nom del professor = " + becari.getNom());
       System.out.println("nom de l'estudiant = " + becari.getNomEstudiant());

   }
}
```

Observacions

- En aquest cas, no cal la sobreescritura dels mètodes getNom i setNom a la classe Becari.

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.