

GRAU D'ENGINYERIA INFORMÀTICA

# PROGRAMACIÓ II

**Bloc 2:**

**Programació Orientada a Objectes (5)**

**Laura Igual**

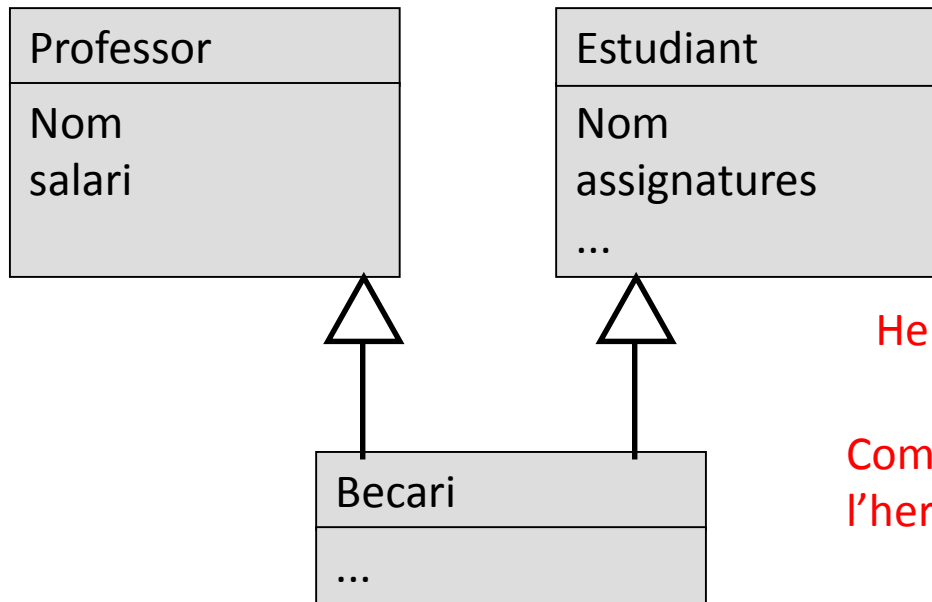
Departament de Matemàtica Aplicada i Anàlisi

Facultat de Matemàtiques

Universitat de Barcelona

# Interfície per herència múltiple

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

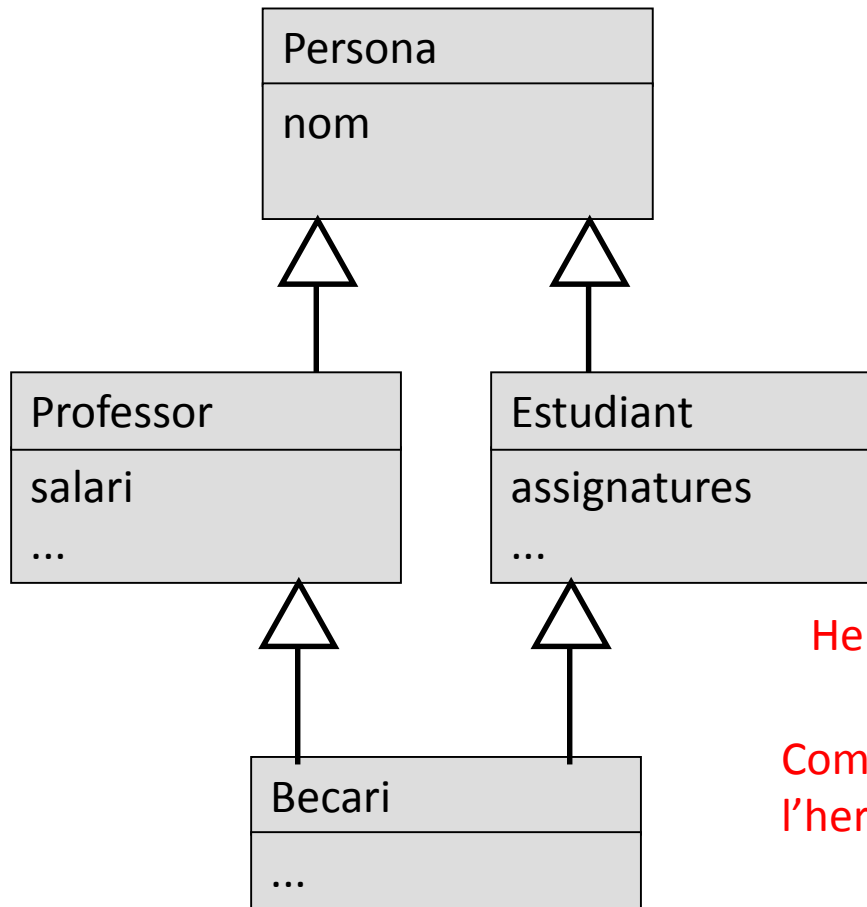


Herència múltiple

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

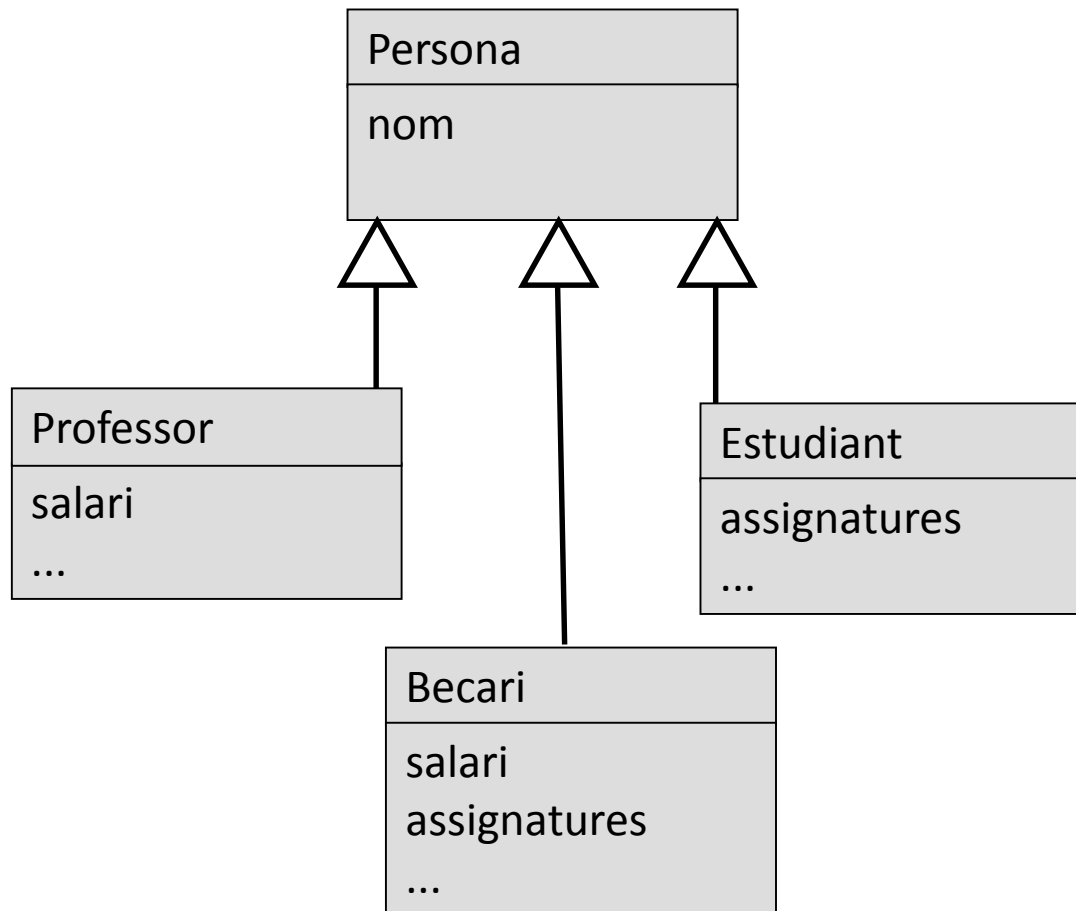
Com solucionem el problema de l'herència múltiple?

# 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.

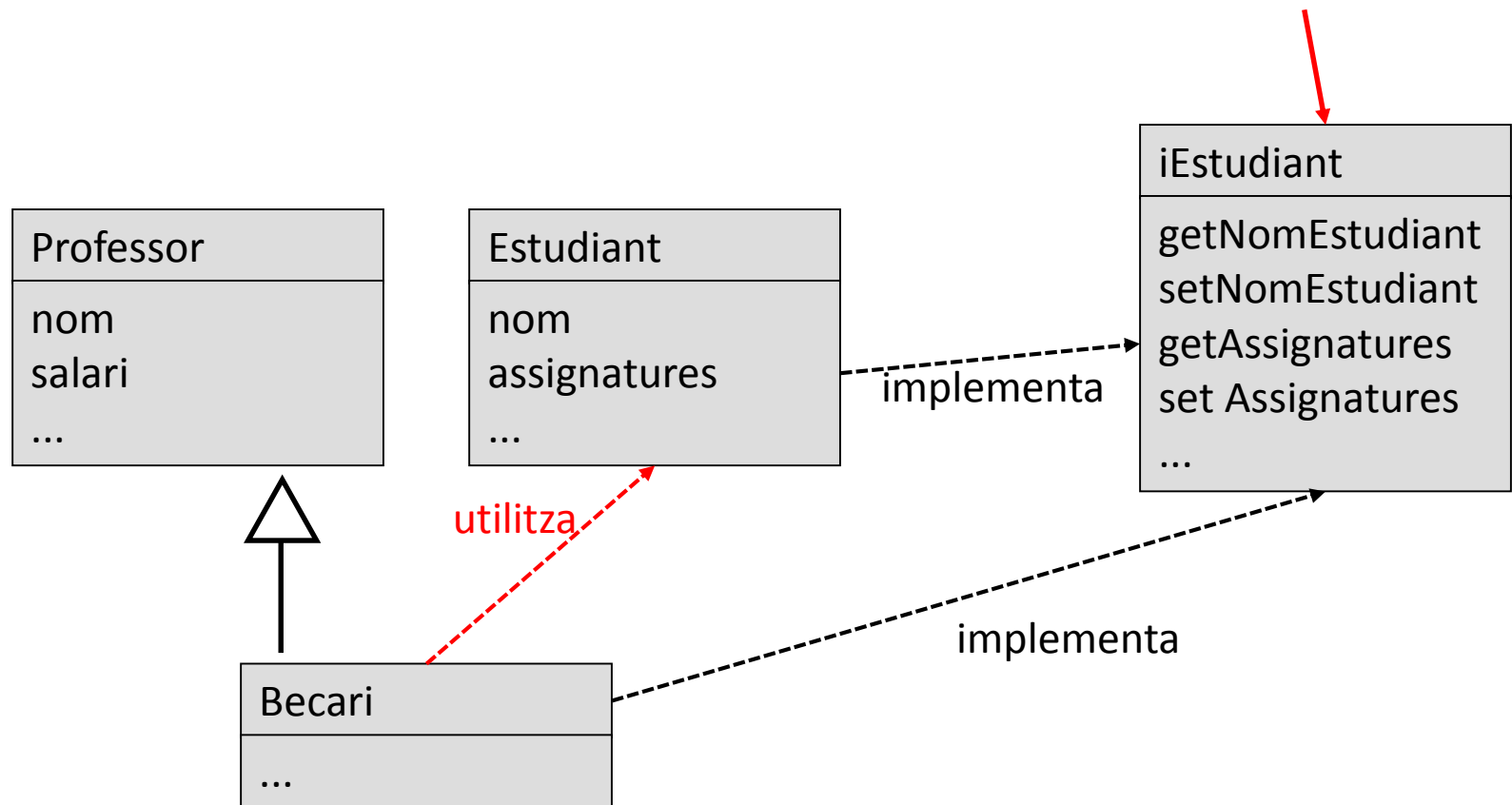
# Interfície per herència múltiple

- Solució fent servir un nou disseny:



# Interfície per herència múltiple

- Solució fent servir una interfície:



# 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**

Classe sense setters

# Solució 1: Exemple Interfícies

```
public interface IEstudiant {  
  
    public String getNomEstudiant ();  
    public void setNomEstudiant (String pNom);  
    public String getAssignatures();  
    public void setAssignatures(String assignatures);  
  
}
```

**IEstudiant.java**



# Solució 1: 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 1: Exemple Interfícies

## Becari.java

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

Defineix un objecte  
de la classe  
Estudiant

# Observacions

- Problema d'aquesta implementació:
  - Si canviem el nom de l'estudiant (mitjançant el mètode setter), el nom del professor no canvia

# Solució 1: Exemple Interfícies

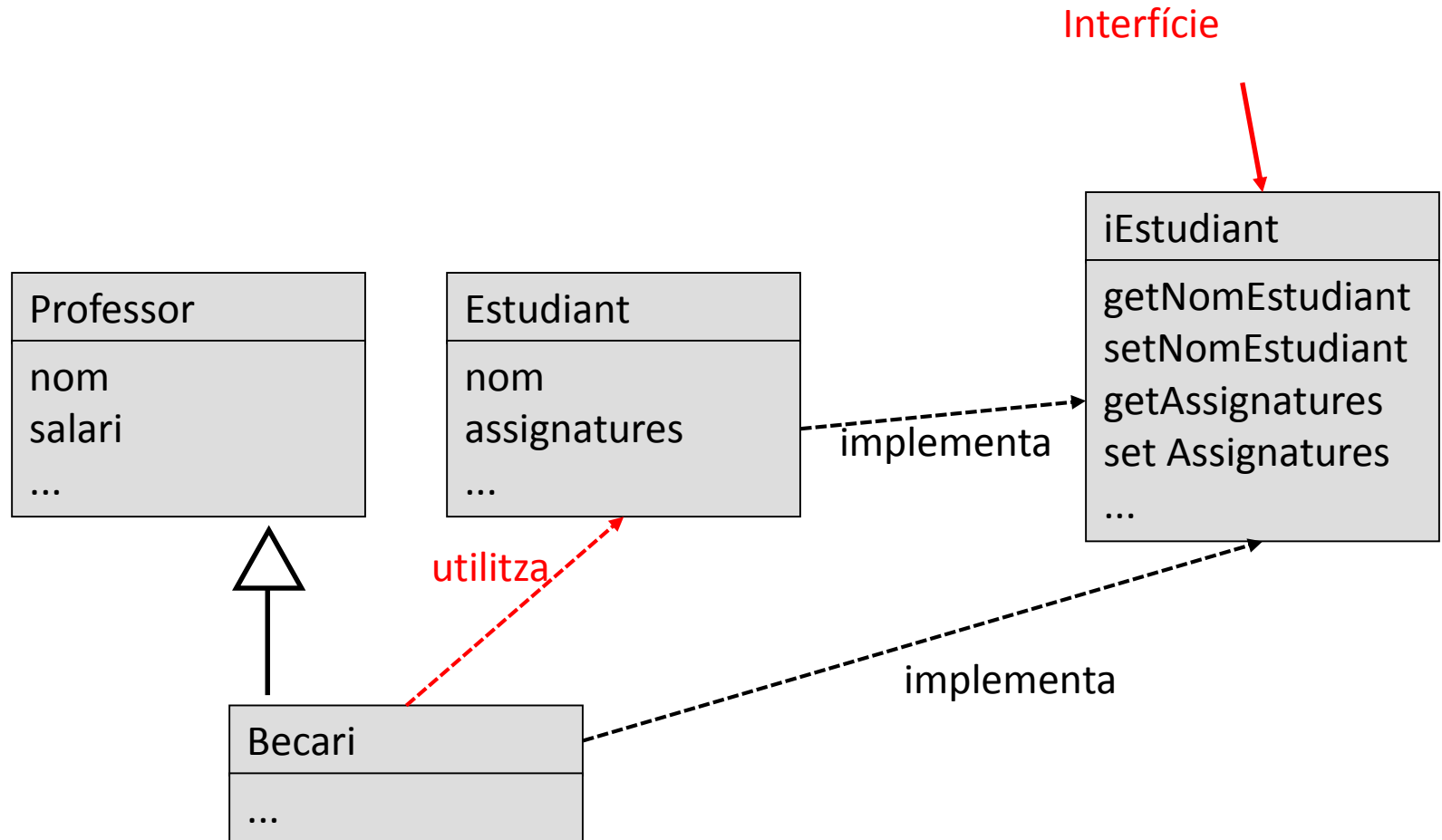
```
public class TestBecari {  
    public static void main(String[] args) {  
  
        Becari becari;  
        becari = new Becari("Joan",1000);  
        System.out.println("salari becari = " + becari.getSalari());  
  
        System.out.println("nom del professor = " + becari.getNom());  
        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());  
  
    }  
}
```

Joan  
Joan

Joan  
Joan Francesc

# Solució 2: Exemple Interfícies

- Solució fent servir una interfície:



# Solució 2: Exemple Interfícies

**Professor.java**

```
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;
    }
}
```

# 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**

El nom de l'estudiant no  
s'utilitza

# 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());  
  
    }  
}
```

Joan  
Joan

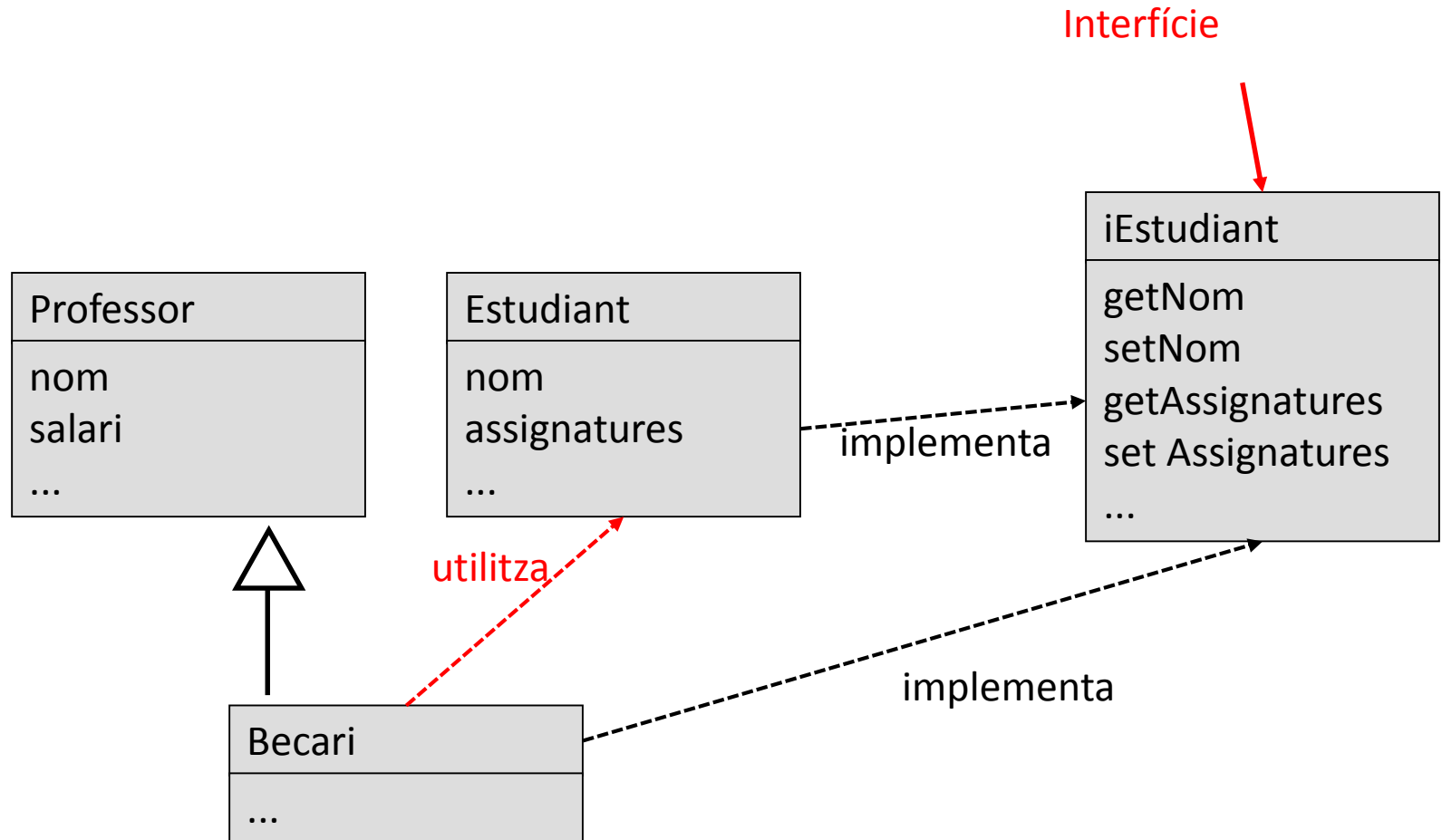
Joan Francesc  
Joan Francesc

# 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**

No hi ha sobreescritura  
dels mètodes getNom i  
setNom.



# 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());  
  
    }  
}
```

Joan  
Joan

Joan Francesc  
Joan Francesc

# Observacions

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

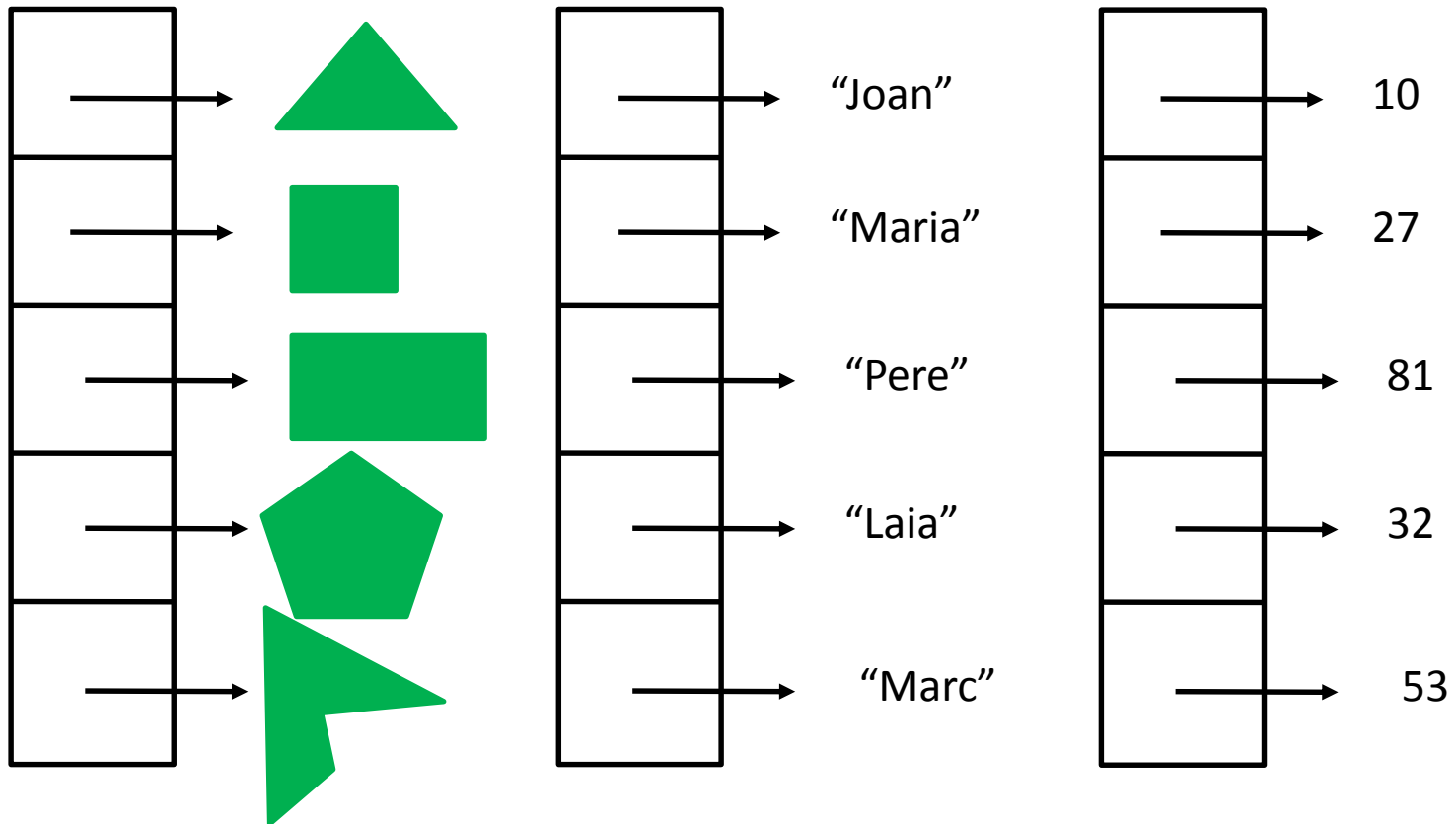
# **COL·LECCIONS: INTRODUCCIÓ A UN EXEMPLE PRÀCTIC**

# Genericitat en Java

- Si defineixo una estructura de dades de tipus `Object`
- Ja que tot tipus és compatible amb l'arrel, obtenim les propietats:
- **Inserció:**
  - Puc inserir qualsevol tipus d'objectes
  - El control l'ha d'implementar el programador
- **Extracció:**
  - Recupero elements de tipus `Object`
  - Fa falta fer una conversió explícita

# Estructures de dades

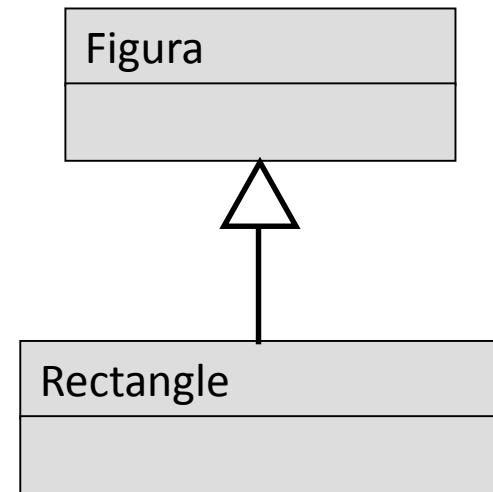
- Estructures de dades polimorfes: que contenen objectes de tipus diferents (**tots descendents d'un tipus comú**).



# Informació de classes en temps d'execució

- Després de realitzar una connexió polimorfa és freqüent la **necessitat de tornar a recuperar l'objecte original**, per a accedir a les seves operacions pròpies
- Exemple:**

```
Figura [] figures = new Figura[10];  
...  
Figures[0]= new Rectangle(); Connexions  
Figures[1]= new Cercle();      polimorfes  
....  
Figura fig;  
for (int i=0; i<10; i++) {  
    fig = figures[i];  
}
```



Pot interessar recuperar  
un Rectangle o Cercle  
en lloc d'una Figura.

# Informació de classes en temps d'execució

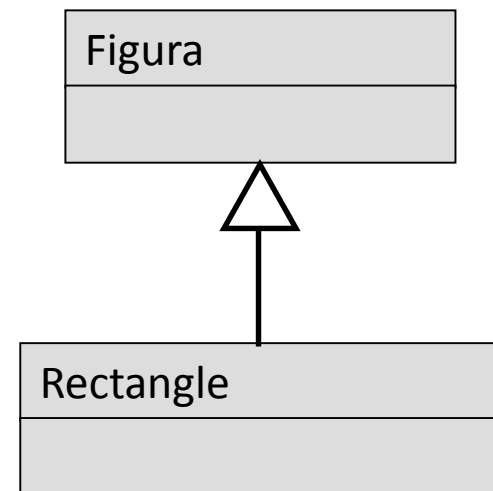
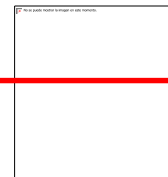
- Es tracta de l'operació inversa al polimorfisme (upcasting), denominada **downcasting**
  - Si el polimorfisme implica una generalització, el downcasting implica una especialització.
- Al contrari que el upcasting, el downcasting no pot realitzar-se directament mitjançant una connexió amb una referència de la classe de l'objecte.
- Recordatori:

## Upcasting

```
Figura figura = new Rectangle();
```

## Downcasting

```
Rectangle rectangle = new Figura();  
Rectangle rectangle = (Rectangle)figures[i];
```



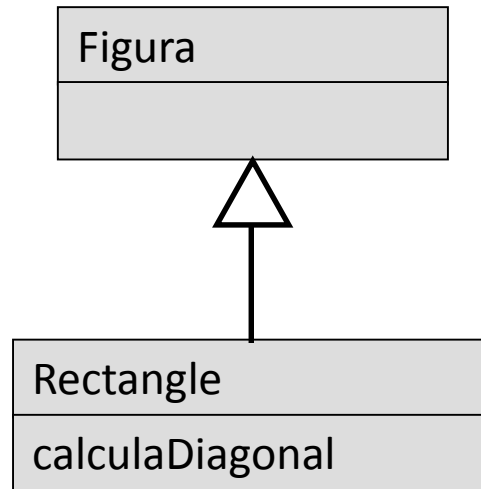
# Informació de classes en temps d'execució

- Un casting permet forçar la connexió a la referència
- Un intent de casting impossible generarà una excepció ***ClassCastException*** en temps d'execució
- Possibles accions:
  - Podem capturar aquesta excepció per a determinar si l'objecte apuntat per la referència és del tipus esperat o no, realitzant accions diferents en cada cas *try .... catch*
  - O, podem utilitzar **instanceof** per determinar si l'objecte és de la classe esperada abans de realitzar el casting.



# Exemple: diagonal màxima

- El mètode diagonal, és un mètode pròpi de la subclasse.



# Exemple: diagonal màxima

```
Figura [] figures = new Figura[10];  
...  
float actual, maxDiagonal=0;  
for (int i=0; i<10; i++){  
    actual = figures[i].calculaDiagonal();  
    if (actual>maxDiagonal)  
        maxDiagonal=actual;  
}
```

Mètode propi de la  
classe Rectangle

Donarà error de compilació!

¿Què passa si no és un rectangle?  
Tindríem que preguntar pel tipus

# Identificació del tipus en temps d'execució

- `if (figures[i] instanceof Rectangle) ...`
- `java.lang` conté la classe **Class**:
  - Conèixer el nom de la classe d'un objecte:  
**String getName()**
  - Saber si un objecte és instància de la classe o d'una subclasse:  
**boolean isInstance(Object o)**
- `if figures[i].getClass().getName().equals("Rectangle")...`

# instanceof vs. equivalencia de Class

- `instanceof` o `isInstance`  
“Ets d'aquesta classe o d'una classe derivada d'aquesta?”
- Comparant els objectes `Class`  
“Ets exactament d'aquesta classe?”
- Exemple: `Rectangle` és una subclasse de la classe `Figura`

```
Rectangle r = new Rectangle();
```

```
(r instanceof Figura) → true
```

```
(r.getClass().equals(Figura.class)) → false
```

# **FRAMEWORK COL·LECCIONS**

# Framework Col·leccions

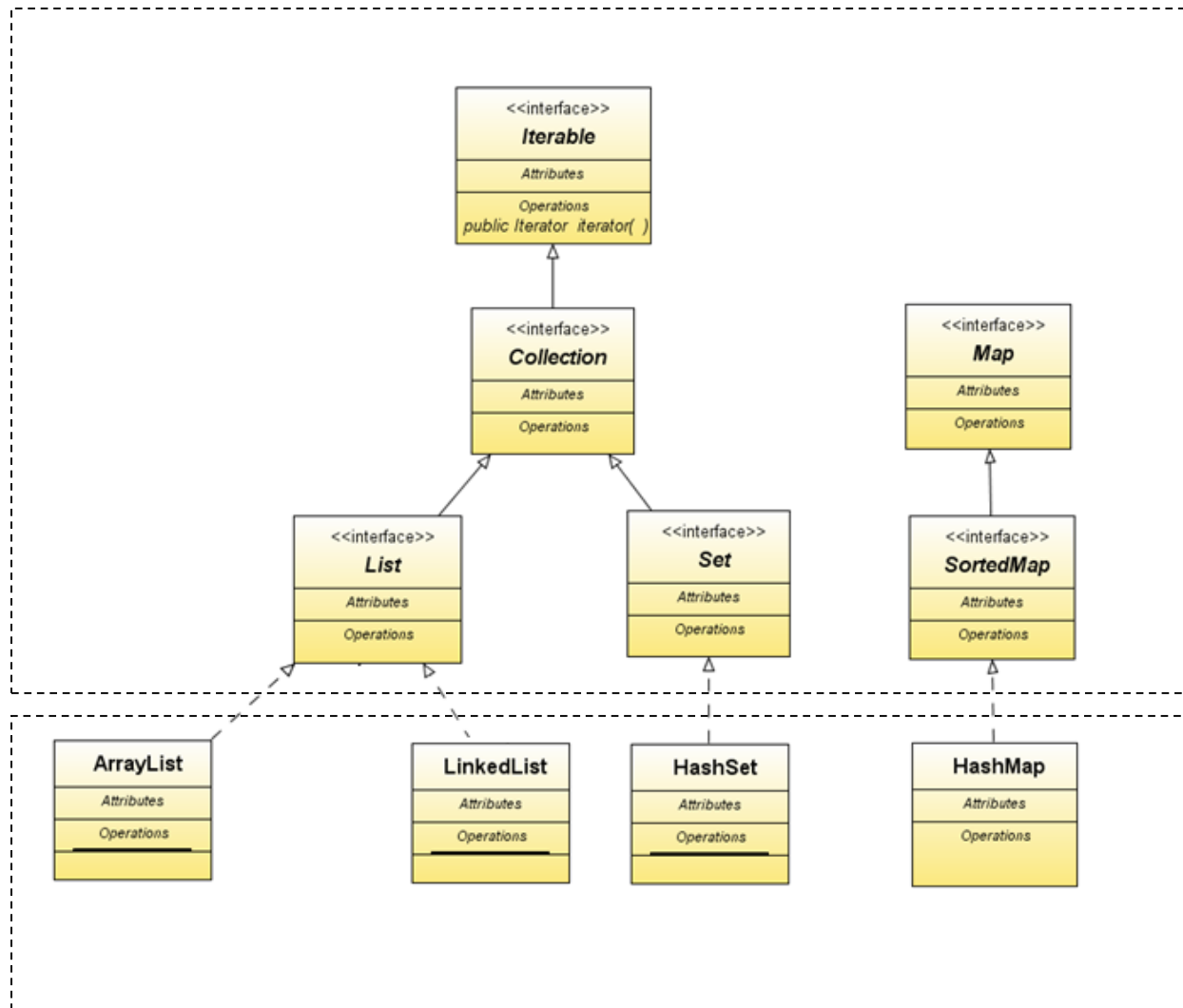
- Una **col·lecció** és un objecte que agrupa múltiples elements en una única unitat.
- Normalment representen elements d'informació dins d'un grup natural, com
  - una bústia de correu (una col·lecció de correus),
  - un directori (una col·lecció de fitxers),
  - una guia telefònica (una associació entre noms i números de telèfon).
- La llibreria standard de Java ens ofereix classes i interfícies que ens permeten manegar col·leccions d'objectes
- **Piles, Cues, Llistes, Conjunts** són casos particulars de col·leccions d'objectes

# Col·leccions

- Encara que ArrayList és la que més utilitzem a les pràctiques, hi ha altres col·leccions útils:
  - LinkedList,
  - HashSet,
  - HashMap,

# Diagrama de classes simplificat

## Interfaces



## Implementations



# ArrayList vs. LinkedList

- LinkedList: una altra implementació d'una llista.
- Una qüestió d'implementació:
  - Quan necessiteu accedir de forma seqüencial i teniu un nombre poc variable d'elements → ArrayList.
  - Quan necessiteu esborrar o inserir al davant o al mig moltes vegades el contingut de la llista → LinkedList.

Creació d'una **LinkedList**

```
LinkedList map = new LinkedList ();
```

# Mapes: Exemples d'Ús

- Conté clau i valor

- Creació d'una **Map**

```
Map map = new HashMap();  
map.put("joan", "777777777");  
map.put("anna", "888888888");  
map.put("jordi", "999999999");
```

```
// això imprimeix '888888888'
```

```
System.out.println("El telèfon d'Anna és: "+ map.get("anna"));
```

El telèfon d'Anna és: 888888888

# Col·leccions i iteradors

La interfície `Iterator`

- Un iterador és un objecte que proveeix una forma de processar una col·lecció d'objectes, un a un, seguint una seqüència.
- Un iterador ens permet recorre els elements d'una col·lecció d'objectes
- Un iterador es crea formalment implementant la interfície `Iterator<E>`, que conté 3 mètodes:
  - **hasNext** → retorna un resultat booleà que és cert si a la col·lecció queden objectes per processar
  - **next** → retorna el següent objecte a processar
  - **remove** → elimina l'objecte més recentment retornat pel mètode `next`

# Col·leccions i iteradors

## La interfície `Iterator`

```
public interface Iterator<E>
{
    E next();
    Boolean hasNext();
    void remove(); //opcional
}
```

- Alguna cosa és iterable si es pot iterar sobre ell. Per poder iterar usem un iterador.
- Una classe és iterable si és capaç de retornar-nos un iterador:

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

# Col·leccions i iteradors

## La interfície `Iterator`

- Implementant la interfície **`Iterator`** una classe formalment estableix que els objectes d'aquesta classe són iteradors
- El programador ha de decidir com implementar les funcions d'iteració
- Un iterador, per tant, caracteritza una seqüència

# Col·leccions: Exemples d'Ús

- **Creació d'una col·lecció d'objectes**

```
Collection c = new ArrayList();  
c.add("Hello");  
c.add("World");
```

- **Recorregut d'una col·lecció amb un iterador**

```
for (Iterator i = c.iterator(); i.hasNext(); ) {  
    String s = (String)i.next();  
    System.out.println(s);  
}
```

- **Recorregut d'una col·lecció amb un *for .. each***

```
for (Object item : c) {  
    System.out.println(item.toString());  
}
```

# Tipus parametritzats

- També podem construir els nostres pròpis tipus parametrizats.

```
public class TaulaBicicleta{  
    private Bicicleta [] taula;  
    private int numElements;  
    ....  
    public void afegir(Bicicleta element){  
        ...  
    }  
}
```

```
public class TaulaPellicula{  
    private Pellicula [] taula;  
    private int numElements;  
    ....  
    public void afegir(Pellicula element){  
        ...  
    }  
}
```

```
public class Taula<T> {  
    private T [] taula;  
    private int numElements;  
    ....  
    public void afegir(T element){  
        ...  
    }  
}
```

← Genèric

→ Exemple: ArrayList

# Exemples d'Ús

- **Exemple 1:** Definició de mètodes que treballen contra la interface Collection.

Col·leccions de tipus heterogeni

**CreaColeccio.java**

- **Exemple 2:** Col·leccions de tipus homogeni

**CreaColeccioHomogenea.java**



# Col·leccions i iteradors: Exemple 1

```
import java.util.*;

public class CreaColeccio {

    public static void main(String[] args) {
        Collection myCollection1 = new ArrayList();
        Collection myCollection2 = new HashSet();

        fillCollection(myCollection1);
        fillCollection(myCollection2);
        showCollection(myCollection1);
        showCollection(myCollection2);
        treuMaria(myCollection1);
        treuMaria(myCollection1);
        diguesSiEstaMaria(myCollection1);
        diguesSiEstaMaria(myCollection2);
    }
```

# Col·leccions i iteradors: Exemple 1

```
public static void fillCollection(Collection c) {  
    c.add(34);  
    c.add("Pepe");  
    c.add(new Gat("Sasha"));  
}
```

```
public static void showCollection(Collection c) {  
    if (c.isEmpty()) { System.out.println("La col·lecció esta buida");  
    } else {  
        System.out.println("La col·lecció conté " + c.size() + " elements:");  
        System.out.println(c);  
    }  
}
```

# Col·leccions i iteradors: Exemple 1

```
public static void treuMaria(Collection c) {  
    c.remove("Maria");  
}
```

```
public static void diguesSiEstaMaria (Collection c) {  
    if (c.contains("Maria")) {  
        System.out.println("Maria està dins de la col·lecció");  
    } else {  
        System.out.println("Maria no està a la col·lecció");  
    }  
}
```

```
}
```

# Col·leccions i iteradors: Exemple 2

```
public class CreaColeccioHomogenea {  
    public static void main(String[] args) {  
        Collection<Gat> myCollection1 = new ArrayList<Gat>();  
        showCollection(myCollection1);  
        fillCollection(myCollection1);  
        showCollection(myCollection1);  
    }  
}
```

```
class Gat {  
    String nom;  
    Gat(String n) {  
        nom = n;  
    }  
    public String toString() {  
        return nom;  
    }  
    public void miolar(){  
        System.out.println("miau");  
    }  
}
```

# Col·leccions i iteradors: Exemple 2

(Continua implementació classe CreaColeccioHomogenea)

```
public static void fillCollection(Collection<Gat> c) {  
    c.add(new Gat("Misu"));  
    c.add(new Gat("Marramiau"));  
    c.add(new Gat("Sasha"));  
}  
  
public static void showCollection(Collection<Gat> c) {  
    if (c.isEmpty()) {  
        System.out.println("La col·lecció està buida");  
    } else {  
        System.out.println("La col·lecció conté " + c.size() + " elements:");  
        System.out.println(c);  
    }  
}
```

# Example: iterators

```
public static void fesMiolar(Collection<Gat> c){  
    Iterator<Gat> it = c.iterator();  
    while(it.hasNext()) {  
        Gat g = it.next();  
        g.miolar();  
    }  
}
```

# Exemple (1)


...

```
ArrayList integerList = new ArrayList();
```

```
integerList.add( new Integer(1));
```

```
integerList.add( new Integer(2));
```

*Els elements de la lista són de tipus Object.*



```
Iterator listIterator = integerList.iterator();
```


```
while(listIterator.hasNext()) {
```

```
    Integer item = (Integer) listIterator.next();
```

```
}
```

...

*El programador ha de fer el casting*



# Exemple (2)

**El mateix exemple afegint un error:**

...


```
ArrayList integerList = new ArrayList();
```

```
integerList.add( new Integer(1));
```

```
integerList.add( new Integer(2));
```

```
integerList.add("Joan");
```

No hi ha cap  
restricció dels  
elements



```
Iterator listIterator = integerList.iterator();
```


```
while(listIterator.hasNext()) {
```

```
    Integer item = (Integer) listIterator.next();
```

```
}
```

```
...
```

El compilador no se n'adona del  
casting il·legal.  
Serà detectat en temps d'execució.





# Exemple (3)

**// Eliminar paraules de 4 lletres de la col·lecció c. Els elements haurien de ser String**

```
static void expurgate(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (((String) i.next()).length() == 4)  
            i.remove();  
}
```

No utilitzar un bucle de recorregut d'indexos per fer això.

No funcionarà! Exercici: provar-ho.

El mateix exemple modificat per a utilitzar tipus generics:

**// Eliminar paraules de 4 lletres de la col·lecció c**

```
static void expurgate(Collection<String> c) {  
    for (Iterator<String> i = c.iterator(); i.hasNext(); )  
        if (i.next().length() == 4)  
            i.remove();  
}
```

# Exemple (4)

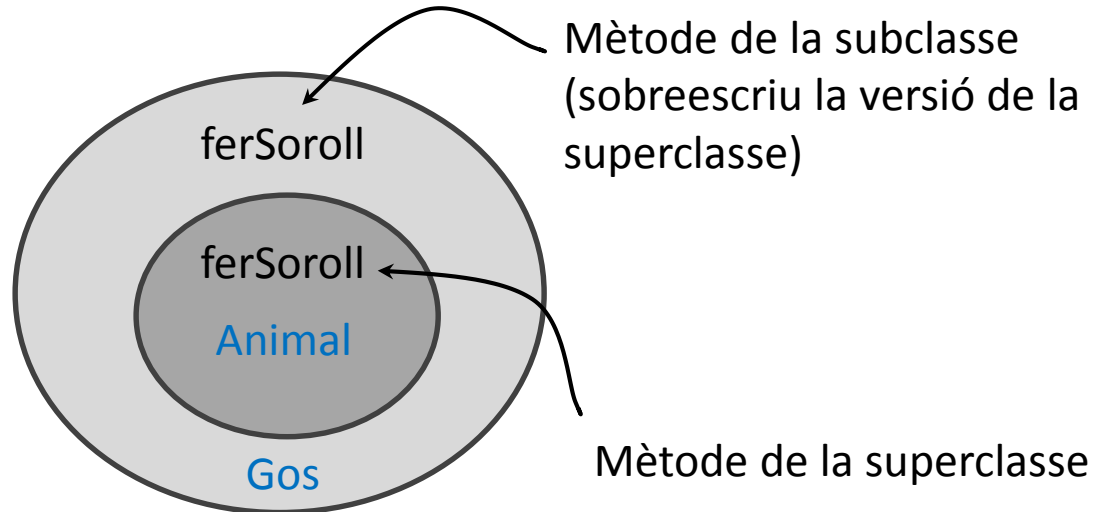
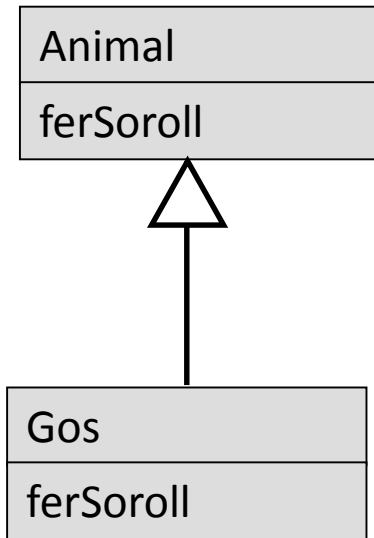
Programa que simula un zoològic:

```
public class Zoo{  
    private ArrayList<Animal> hostes = new ArrayList<Animal>();  
    public void nuevoAnimal(Animal a){  
        hostes.add(a);  
    }  
}
```

Codi genèric per  
tots els animals

- Si no utilitzem polimorfisme tindríem que saber en temps de disseny quins animals tindrà exactament.
- Per tant dissenyes una classes que sigui lo suficientment genèrica com per a que accepti qualsevol tipus d'animal sense necessitat de saber quin tipus d'animal és: `ArrayList<Animal>`

# Exemple: Herència



## **Nota:**

Una referència a un objecte de la subclasse (**Gos**) sempre cridarà a la versió de la subclasse del mètode sobreescrit. Això és el polimorfisme. Però el codi de la subclasse pot cridar **super.ferSoroll()** per invocar la versió de la superclasse.

La paraula reservada **super** és una referència a la porció de la superclasse d'un objecte. Quan el codi de la subclasse utilitza **super**, com en **super.ferSoroll()**, la versió del mètode de la superclasse s'executarà.

# En l'exemple d'Interfícies

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

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

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

# En l'exemple d'Interfícies

```
public class TestLlistaAnimals {  
    public static void main(String[] args){  
        ArrayList<Animal> llistaAnimals =  
            new ArrayList<Animal>();  
        Gos gosEx = new Gos();  
        Gat gatEx = new Gat();  
        llistaAnimals.add(gosEx);  
        llistaAnimals.add(gatEx);  
        //...
```

```
        // continuació del mètode main:  
        Gos gos;  
        Gat gat;  
        Iterator<Animal> itrLlista = llistaAnimals.iterator();  
        Animal animal;  
        while(itrLlista.hasNext()) {  
            animal = itrLlista.next();  
            System.out.println("He extret el animal del tipus:" + animal.getClass());  
            // animal.persegueix(); // Error de compilació  
            // animal.esgarrapar(); // Error de compilació  
            // No puc fer aquestes crides abans he de fer un cast de la següent  
            manera:  
            if (animal instanceof Gos){  
                gos = (Gos) animal;  
                gos.persegueix();  
            }else if (animal instanceof Gat){  
                gat = (Gat) animal;  
                gat.esgarrapar();  
            }  
        }  
    }  
}
```

## Sortida per pantalla:

He extret el animal del tipus:class unAltreInterfícies.Gos  
persegueix

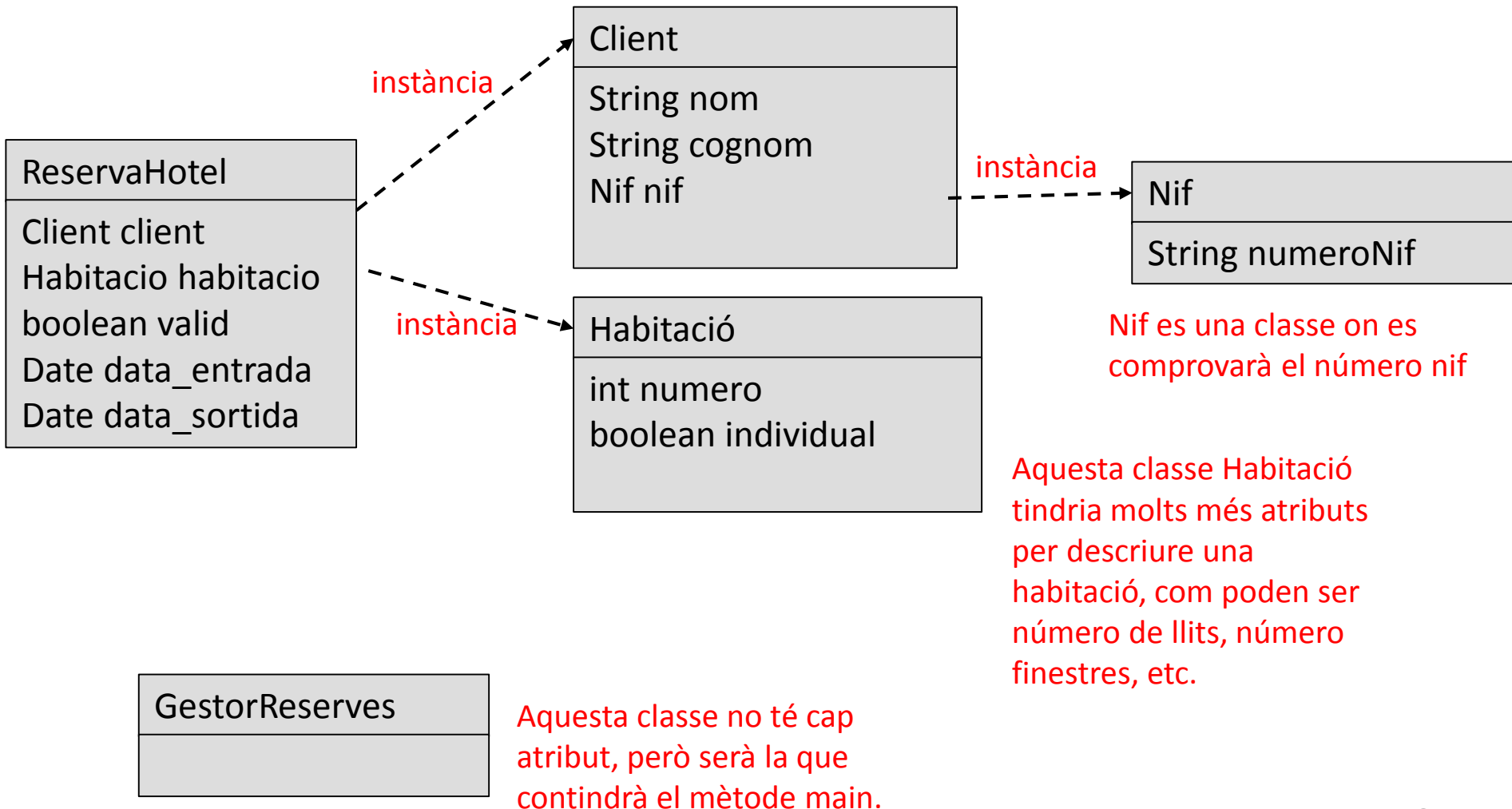
He extret el animal del tipus:class unAltreInterfícies.Gat  
esgarrapa

# **EXERCICI REPÀS**

# Exercici

- Es vol implementar una aplicació de gestió de reserves d'hotel seguint el diagrama de classes definit a continuació. On apareixen el nom de les classes i la llista dels seus atributs.
- Implementa les classes:
  - **ReservaHotel**
  - **Habitacio**
  - **Client**
  - **GestorReserves**
- Al mètode main de la classe **GestorReserves** s'ha de crear una reserva i validar-la.

# Exercici: Diagrama de classes





# Exemple: Implementació

```
public class GestorReserves {  
    public static void main(String[] args){  
        Habitacio habitacio = new Habitacio(1);  
        Client client = new Client("XXXXXX");  
        // Demanar la informació sobre el client  
        // Crear una nova reserva del hotel:  
        ReservaHotel novaReserva = new ReservaHotel(habitacio, client);  
        // Validar quan ja s'ha pagat la reserva:  
        novaReserva.validar();  
    }  
}
```

```
public class ReservaHotel{  
    Habitacio habitacio;  
    Client client;  
    Date data_entrada;  
    Date data_sortida;  
    boolean valid;  
    // constructor de la classe:  
    public ReservaHotel(Habitacio habitacio, Client client){  
        this.habitacio = habitacio;  
        this.client = client;  
        valid = false;  
    }  
    // mètode per validar la reserva:  
    public void validar(){  
        valid = true;  
    }  
    // més mètodes ...  
}
```

```
public class Habitacio{  
    int numero;  
    boolean individual;  
    // constructor de la classe:  
    public Habitacio(int numero){  
        this.numero= numero;  
        this. individual = false;  
    }  
    public Habitacio(int numero, boolean individual){  
        this.numero= numero;  
        this. individual = individual;  
    }  
    //... setters & getters ...  
}
```

No cal definir el constructor per defecte si no vols crear una reserva sense assignar habitació al client.

El constructor per defecte existeix mentre no es sobrecarregui amb qualsevol conjunt de parametres (inclos sense parametres).

# Exemple: Implementació

Una altra opció d'Implementació de la classe ReservaHotel:

```
public class ReservaHotel{  
    Habitacio habitacio;  
    Client client;  
    Date data_entrada;  
    Date data_sortida;  
    boolean valid = false;  
    // constructors de la classe:  
    public ReservaHotel(){  
        valid = false;  
    }  
    public ReservaHotel(Habitacio habitacio, Client client){  
        this();  
        this.habitacio = habitacio;  
        this.client = client;  
    }  
    // mètode per validar la reserva:  
    public void validar(){  
        valid = true;  
    }  
    // més mètodes ...  
}
```

Si volem crear una  
resea sense assignar  
habitació i client ho  
podem fer així.

# Exemple

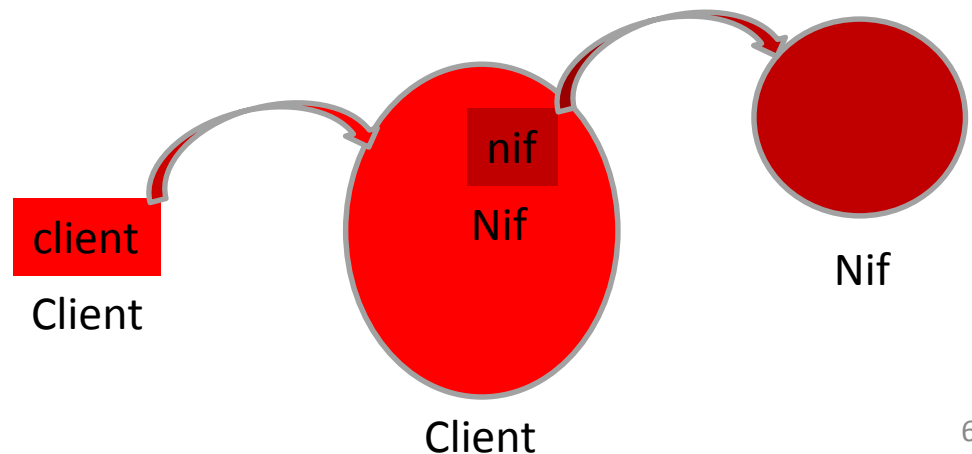
```
public class Client{  
    String nom;  
    String cognom;  
    Nif nif;  
  
    // constructor de la classe:  
    public Client(String nif){  
        nif = new Nif(nif);  
    }  
    // mètodes d'accés i d'escriptura de la classe:  
    public void setNom(String nom){  
        this.nom=nom;  
    }  
    public void setCognom(String cognom){  
        this.cognom = cognom;  
    }  
    public String getNom(){  
        return this.nom;  
    }  
    public String getCognom(){  
        return this.cognom;  
    }  
    // més mètodes  
}
```

Sempre que creem un nou client ha de tenir un Nif associat.

# Exemple: Observació

```
public class GestorReserves {  
    public static void main(String[] args){  
        Habitacio habitacio = new Habitacio(1);  
        Client client = new Client("44444444P");  
        // demanar la informació sobre el client  
        // Crear una nova reserva del hotel:  
        ReservaHotel novaReserva = new ReservaHotel(habitacio, client);  
        // Validar quan ja s'ha pagat la reserva:  
        novaReserva.validar();  
    }  
}
```

Quan instanciem un objecte de la classe Client, estem instanciant un objecte de la classe Nif.



# 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.