



# Unió Europea

Fons Social Europeu

*L'FSE inverteix en el teu futur*



GENERALITAT  
VALENCIANA



## UF09 - POO II

*- Teoria -*

PROGRAMACIÓ  
CFGS DAM

Autor:

Àngel Olmos Giner

segons el material de Carlos Cacho i Raquel Torres  
(Programació) i Sergio Badal (Entorns de Desenvolupament)

[a.olmosginer@edu.gva.es](mailto:a.olmosginer@edu.gva.es)

2022/2023

# POO - II

## ÍNDEX DE CONTINGUTS

1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. HERÈNCIA
4. POLIMORFISME
5. CLASSES ABSTRACTES
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES

# 1. LA CLASSE *ARRAYLIST*

## INTRODUCCIÓ



No és estrictament POO però és un bon moment per a introduir-ho

1. Un *ArrayList* és una **estructura de dades dinàmica** del tipus col·lecció que implementa una **llista de grandària variable**
2. És similar a un *Array* amb els avantatges que la seua grandària creix dinàmicament a mesura que s'afigen elements (**no és necessari fixar la seua grandària en crear-lo**)
3. Permet emmagatzemar **dades i objectes de qualsevol tipus**



# 1. LA CLASSE *ARRAYLIST*

## *DECLARACIÓ I OMPLIMENT*



Ara ja no cal especificar la grandaria

Un *ArrayList* es declara com una classe



```
ArrayList llista = new ArrayList()
```

Necessitarem importar la classe → *import java.util.ArrayList*

Per a **inserir elements** en un *ArrayList* es pot utilitzar-se el **mètode add()**

```
llista.add(3.14);
```

```
llista.add('A');
```

```
llista.add("Lluis");
```

Pot incloure diferents tipus de dades

Juntament amb objectes

```
llista.add(new Persona("65971552A", "Luis", "González Collado", 17));
```

```
llista.add(new Persona("16834954R", "Raquel", "Dobón Pérez", 62));
```

Aquest codi afig referències a objectes del tipus *Persona* dins l'*ArrayList*

# 1. LA CLASSE *ARRAYLIST*

## *MÈTODES D'ACCÉS I MANIPULACIÓ*



- `int size()`; retorna el nombre d'elements de la llista
- `E get(int index)`; retorna una referència a l'element en la posició `índex`
- `void clear()`; elimina tots els elements de la llista. Estableix la grandària a zero
- `boolean isEmpty()`; retorna *true* si la llista no conté elements
- `boolean add(E element)`; insereix *element* al final de la llista i retorna *true*
- `void add(int index, E element)`; insereix *element* en la posició `índex` de la llista. Desplaça una posició tots els altres elements de la llista (no substitueix ni esborra altres elements)
- `void set(int index, E element)`; substitueix l'element en la posició `índex` per *element*
- `boolean contains(Object obj)`; cerca l'objecte *obj* en la llista i retorna *true* si existeix

# 1. LA CLASSE *ARRAYLIST*

## *MÈTODES D'ACCÉS I MANIPULACIÓ*



- `int indexOf(Object obj)`; cerca l'objecte *obj* en la llista, començant pel principi, i retorna l'índex on se troba. Retorna -1 si no existeix
- `int lastIndexOf(Object obj)`; com `indexOf()` però cerca des del final de la llista
- `E remove(int index)`; elimina l'element en la posició índex i el retorna
- `boolean remove(Object obj)`; elimina la primera ocurrència d'*obj* en la llista. Retorna *true* si ho ha trobat i eliminat, *false* en un altre cas
- `void remove(int index)`; Elimina l'objecte de la llista que es troba en la posició índex. És més ràpid que el mètode anterior, ja que no necessita recórrer tota la llista

Documentació oficial ArrayList

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*

- for + get()*
- Objecte *Iterator* i els seus mètodes

```
// Recorregut amb FOR+GET()
System.out.println("\n// Recorregut amb FOR+GET()");

for (int i = 0; i < llista.size(); i++) {
    System.out.println(llista.get(i)); // Invoca el mètode Persona.toString()
}
```

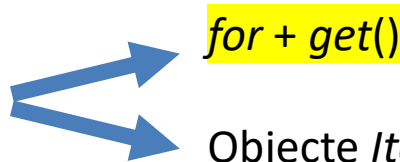
println() { INVoca AUTOMÀTICAMENT AL MÈTODE toString() }

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*



```
public static void main(String[] args) {  
    // TODO code application logic here  
    ArrayList llista = new ArrayList();  
  
    llista.add(-25);  
    llista.add(3.14);  
    llista.add('A');  
    llista.add("Lluís");  
  
    llista.add(new Persona("12345678P", "Peter", "Pan", 16));  
    llista.add(new Persona("99999999D", "Darth", "Vader", 54));  
}
```

DIY

Recupereu la classe  
*Persona* del tema anterior  
i afegiu-li el mètode:  
*public String toString(){*  
....  
*}*



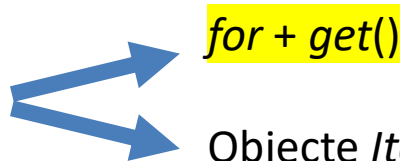


# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*



*for + get()*

Objecte *Iterator* i els seus mètodes

```
for (int i = 0; i < llista.size(); i++) {  
    System.out.println(llista.get(i)); //  
}
```

DIY

*println* invocarà el mètode  
*toString()* dels Objectes

```
// Recorregut amb FOR+GET()  
Nom: Peter, Cognoms: Pan, DNI: 12345678P, edat: 16  
Nom: Darth, Cognoms: Vader, DNI: 99999999D, edat: 54
```

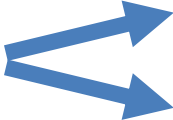
Comenta el mètode *toString()* de la  
classe *Persona* i observa la diferència

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*



*for + get()*

Objecte *Iterator* i els seus mètodes

- Un objecte *Iterator* permet recórrer llistes com si fora un índex
- Es necessita importar la classe `import java.util.Iterator`
- Es crea a partir del mètode `iterator()` que tenen classes com *ArrayList*:  
*Iterator iter = llista.iterator();*
- Els objectes *Iterator* tenen dos mètodes principals:
  - `hasNext()`: Verifica si hi ha més elements a recórrer
  - `next()`: retorna l'objecte actual i avança al següent

Podrem recórrer els *ArrayList* amb els mètodes de *Iterator* combinat amb bucles *while* o *for*

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*

*for + get()*

Objecte *Iterator* i els seus mètodes

```
// Recorregut amb objecte Iterator + WHILE
System.out.println("\n// Recorregut amb objecte Iterator + WHILE");
Iterator iter = llista.iterator(); //Iterator creat a partir del ArrayList
while(iter.hasNext()){
    System.out.println(iter.next());
}
```

Continua sempre que queden elements:

Extrau el següent element

`hasNext()` --> "hi ha un altre element?"

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Hi ha 2 formes de recórrer un *ArrayList*

*for + get()*

Objecte *Iterator* i els seus mètodes

```
// Recorregut amb objecte Iterator + FOR
System.out.println("\n// Recorregut amb objecte Iterator + FOR");
for (Iterator iterFor = llista.iterator(); iterFor.hasNext(); ) {
    System.out.println(iterFor.next());
}
```

Creació de l'objecte *Iterator*  
a l'inici del bucle

No és necessari actualitzar cap comptador

Continua sempre que queden elements

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



**Exemple 1:** Crea un *ArrayList* i implementa les següents accions:

1. Guarda contingut als 4 primers elements (el que vulgues)
2. Introdueix un nou element a la posició 2 ... **Què ha passat amb l'element que hi havia?**
3. Mostra un missatge amb la grandària del *ArrayList*
4. Recorre *l'ArrayList* i mostra el seu contingut amb un FOR + mètode *get()*
5. Recorre *l'ArrayList* i mostra el seu contingut amb un WHILE + objecte *Iterator*

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



### Exemple 1:

```
public static void main(String[] args) {  
    ArrayList numeros = new ArrayList();  
  
    numeros.add("u");  
    numeros.add("dos");  
    numeros.add("tres");  
    numeros.add("quatre");  
  
    numeros.add(2,"dos2"); // Afegix a la posició '2' i desplaça el que hi havia  
  
    System.out.println("Grandaria: " + numeros.size());  
  
    System.out.println("\nContingut (amb FOR + GET()): ");  
    for (int i = 0; i < numeros.size(); i++) {  
        System.out.println(numeros.get(i));  
    }  
  
    System.out.println("\nContingut (amb WHILE + Iterator): ");  
    Iterator iter = numeros.iterator();  
    while(iter.hasNext()){  
        System.out.println(iter.next());  
    }  
}
```

[ u, dos, dos2, tres, quatre ]

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Ús d'objectes guardats en *ArrayList* → Cal fer *casting*

```
llista.add(new Persona("98765432A", "Baby", "Yoda", 184));  
System.out.println(llista.get(0));
```

Excepte per  
a *toString()*

Classe *Persona*

```
public String toString(){  
    return ("Nom: " + this.nom + ", Cognoms: " + this.cognoms +  
           ", DNI: " + this.DNI + ", edat: " + this.edat);  
}
```

```
Nom: Baby, Cognoms: Yoda, DNI: 98765432A, edat: 184  
BUILD SUCCESSFUL (total time: 0 seconds)
```

OK → S'ha invocat directament  
al mètode *Persona.toString()*

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Ús d'objectes guardats en *ArrayList* → Cal fer *casting*

```
if(llista.get(0).esJubilat()){  
    System.out.println(llista.get(0).getNom() + " està jubilat");  
}
```

Però no es poden  
invocar la resta de  
mètodes  
directament

`llista.get(0).`

<code>equals(Object obj)</code>	<code>boolean</code>
<code>getClass()</code>	<code>Class&lt;?&gt;</code>
<code>hashCode()</code>	<code>int</code>
<code>notify()</code>	<code>void</code>
<code>notifyAll()</code>	<code>void</code>
<code>toString()</code>	<code>String</code>
<code>wait()</code>	<code>void</code>
<code>wait(long timeoutMillis)</code>	<code>void</code>
<code>wait(long timeoutMillis, int nanos)</code>	<code>void</code>

La resta de mètodes no estan accessibles



**SOLUCIÓ:** extraure fent *casting*



# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Ús d'objectes guardats en *ArrayList* → Cal fer *casting*

**SOLUCIÓ:** extraure fent *casting*

```
llista.add(new Persona("98765432A", "Baby", "Yoda", 184));
System.out.println(llista.get(0));

Persona personal = (Persona) llista.get(0);

if(personal.esJubilat()){
    System.out.println(personal.getNom() + " està jubilat");
}
```

Igual que quan fem el casting de (int) o (double)

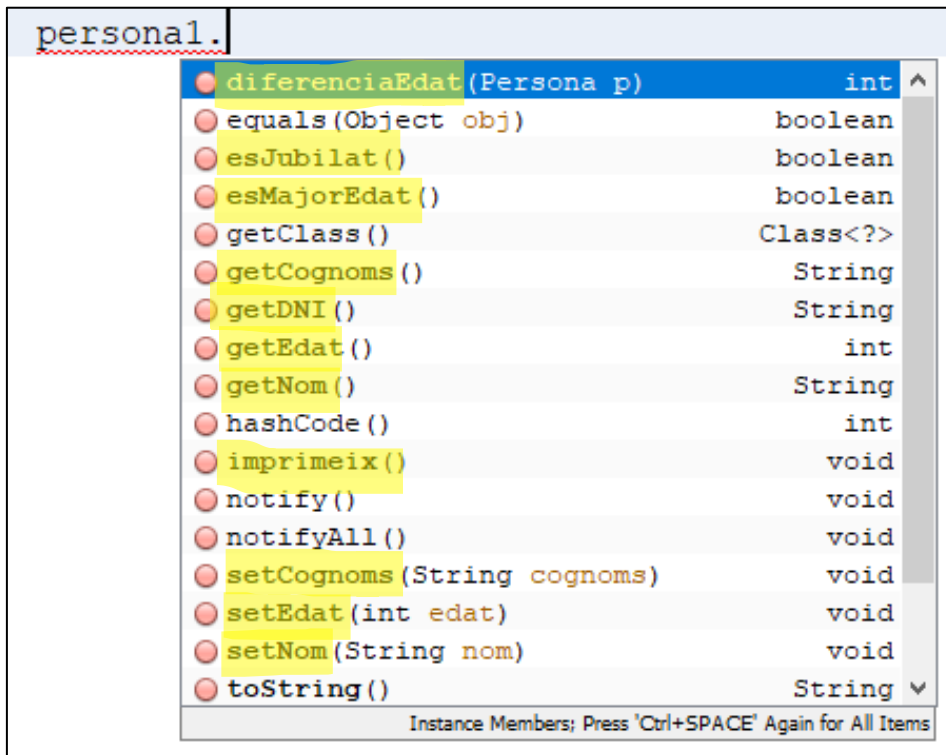
```
Nom: Baby, Cognoms: Yoda, DNI: 98765432A, edat: 184
Baby està jubilat
BUILD SUCCESSFUL (total time: 0 seconds)
```

# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



Ús d'objectes guardats en *ArrayList* → Cal fer *casting*



# 1. LA CLASSE *ARRAYLIST*

## *RECORREGUT*



**Exemple 2:** Crea la classe *Producte* amb:

1. Dos atributs: nom (*String*) i quantitat (*int*)
2. Un constructor amb paràmetres
3. Un constructor sense paràmetres on caldrà que dones valors per defecte
4. Mètodes *getters* i *setters* associats als atributs

En el programa principal:

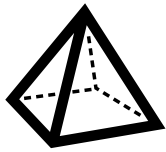
1. Afegeix 5 objectes *Producte* a una llista *ArrayList*
2. Torna a incloure un dels productes a una posició intermèdia
3. Mostra el contingut de la llista amb un FOR + *Iterator* i els *getters* dels objectes
4. Elimina un dels productes de la llista
5. Mostra el contingut de la llista amb un WHILE + *Iterator* i els *getters* dels objectes
6. Esborra la llista i mostra la comprovació de l'esborrat per pantalla

# POO - II

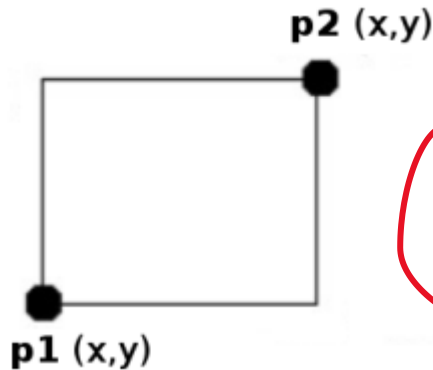
## ÍNDEX DE CONTINGUTS

1. LA CLASSE *ARRAYLIST*
2. **COMPOSICIÓ**
3. HERÈNCIA
4. POLIMORFISME
5. CLASSES ABSTRACTES
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES

## 2. COMPOSICIÓ



- És l'agrupament **d'un o diversos objectes i valors dins d'una classe**
- La composició crea una relació **'té' o 'està compost per'**

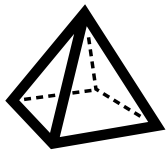


```
public class Punt {  
    int x, y;  
    ...  
}
```

Un rectangle està compost per dos punts (cadascun amb les seues coordenades x,y)

```
public class Rectangle {  
    Punt p1;  
    Punt p2;  
    ...  
}
```

## 2. COMPOSICIÓ



Un compte bancari té titular i autoritzat (ambdues persones amb dni, nom, etc.).

A més del saldo, el compte tindrà registrat un llistat de moviments (tipus, data, quantitat, etc.)

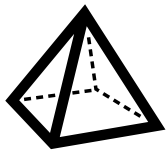
```
public class Persona {  
    String dni, nom, adreça, telèfon;  
    ...  
}
```

```
public class Moviment {  
    int tipus;  
    Date data;  
    double quantitat;  
    String concepte, origen, destinació;  
    ...  
}
```

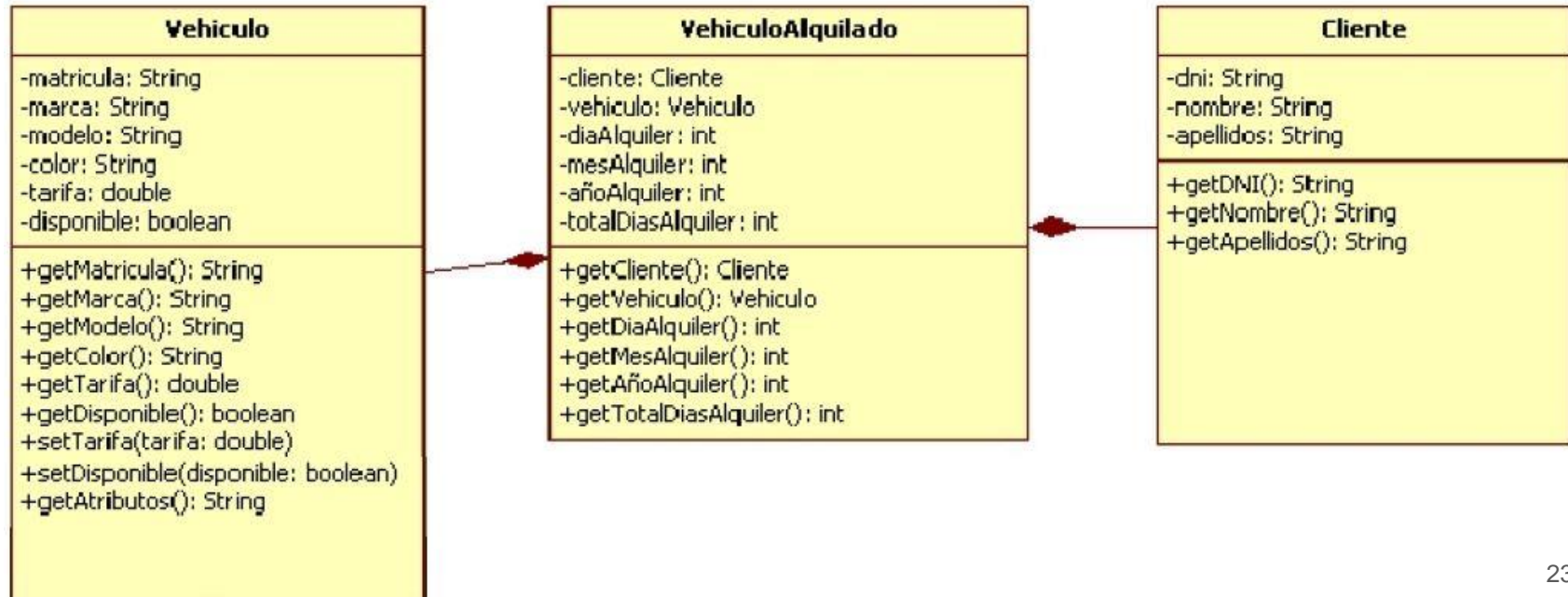
```
public class CompteBancari {  
    Persona titular;  
    Persona autoritzat;  
    double saldo;  
    Moviment moviments[];  
    ...  
}
```

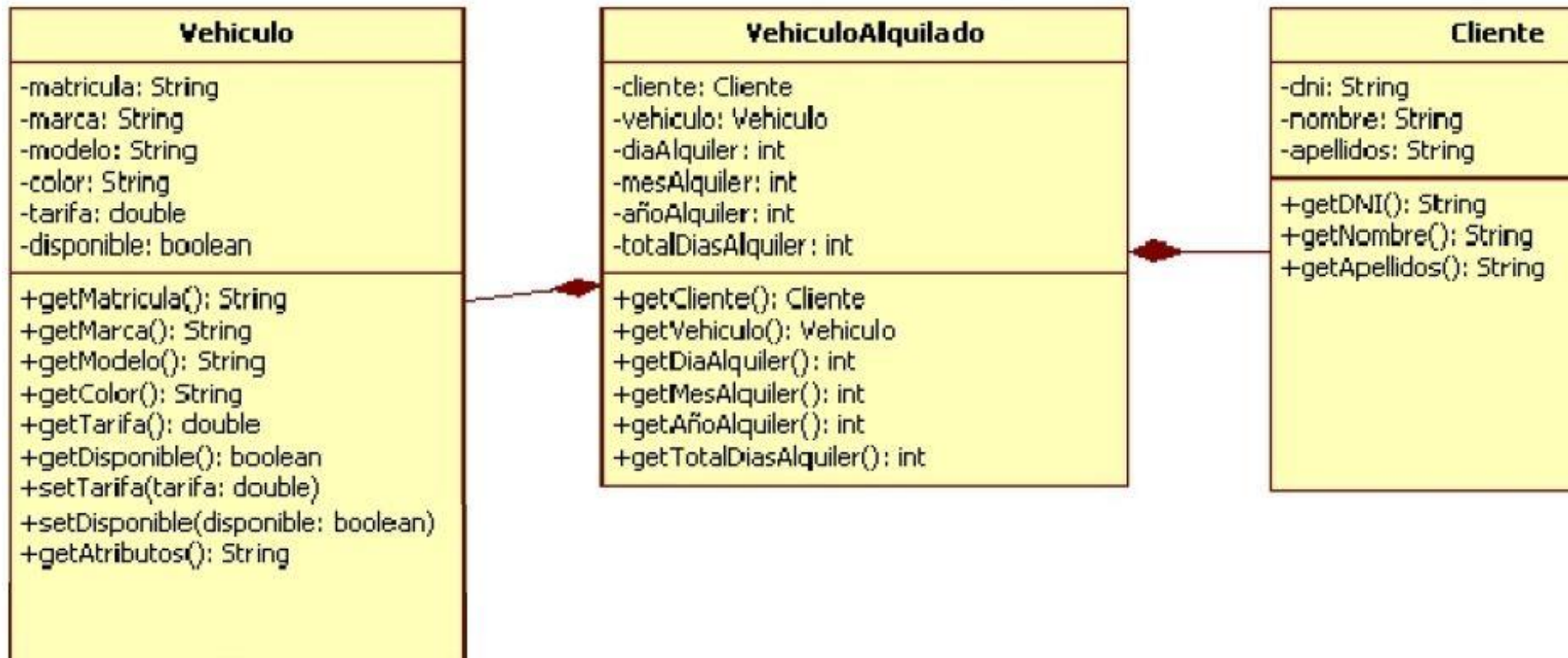
La composició és una capacitat molt potent de la POO que permet **dissenyar programari com un conjunt de classes que col·laboren entre si**, facilitant la modularitat i reutilització del codi → Cada classe s'especialitza en una tasca concreta

## 2. COMPOSICIÓ



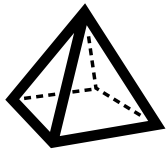
**Exemple 3:** Defineix una composició que declare un objecte de la classe *Vehicle* i un objecte de la classe *Client*. La nova classe *VehicleLlogat* relaciona una instància de la classe *Vehicle* amb una instància de la classe *Client* i crea objectes que emmagatzenen relacions entre clients i vehicles de lloguer ... (continua)







## 2. COMPOSICIÓ



### Exemple 3: ... (continuació)

Instància un objecte *VehicleLlogat* a partir d'instàncies de *Vehicle* i *Client* (inventa les dades)

Un objecte de la classe contenidora pot accedir als mètodes públics de les classes contingudes

En la declaració de la *VehicleLlogat* s'han definit dos mètodes *get* per als atributs de tipus objecte. El mètode *getClient()* retorna un objecte de tipus *Client* i el mètode *getVehicle()* retorna un objecte de tipus *Vehicle*.

Modifica el *main* per a mostrar per pantalla un missatge com aquest, utilitzant l'objecte *VehicleLlogat*:

```
run:
Vehicle Llogat
Client : 30435624X Juan Pérez
Vehicle: 4050 ABJ
BUILD SUCCESSFUL (total time: 0 seconds)
```

# POO - II

## ÍNDEX DE CONTINGUTS

1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. **HERÈNCIA**
4. POLIMORFISME
5. CLASSES ABSTRACTES
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES

### 3. HERÈNCIA

#### INTRODUCCIÓ



- És una de les capacitats més importants i distintives de la POO
- Consisteix en derivar o **estendre una nova classe a partir d'una altra ja existent de manera que la nova classe hereta tots los atributs i mètodes de la classe ja existent**
- A la classe ja existent se la denomina **superclasse**, classe **base** o classe **pare**
- A la nova classe se la denomina **subclasse**, classe **derivada** o classe **filla**



Quan derivem (o estenem) una nova classe, aquesta hereta totes les dades i mètodes membre de la classe existent.

### 3. HERÈNCIA

#### INTRODUCCIÓ



#### EXEMPLE:

Si tenim un programa que treballa amb *alumnes* i *professors*, tindran atributs comuns com:

- Nom
- Dni
- Adreça ...

Però tant *alumnes* com *professors* tindran atributs específics que no tindran els altres.

Per exemple l'alumnat tindrà:

- Número d'expedient
- Cicle
- Curs ...

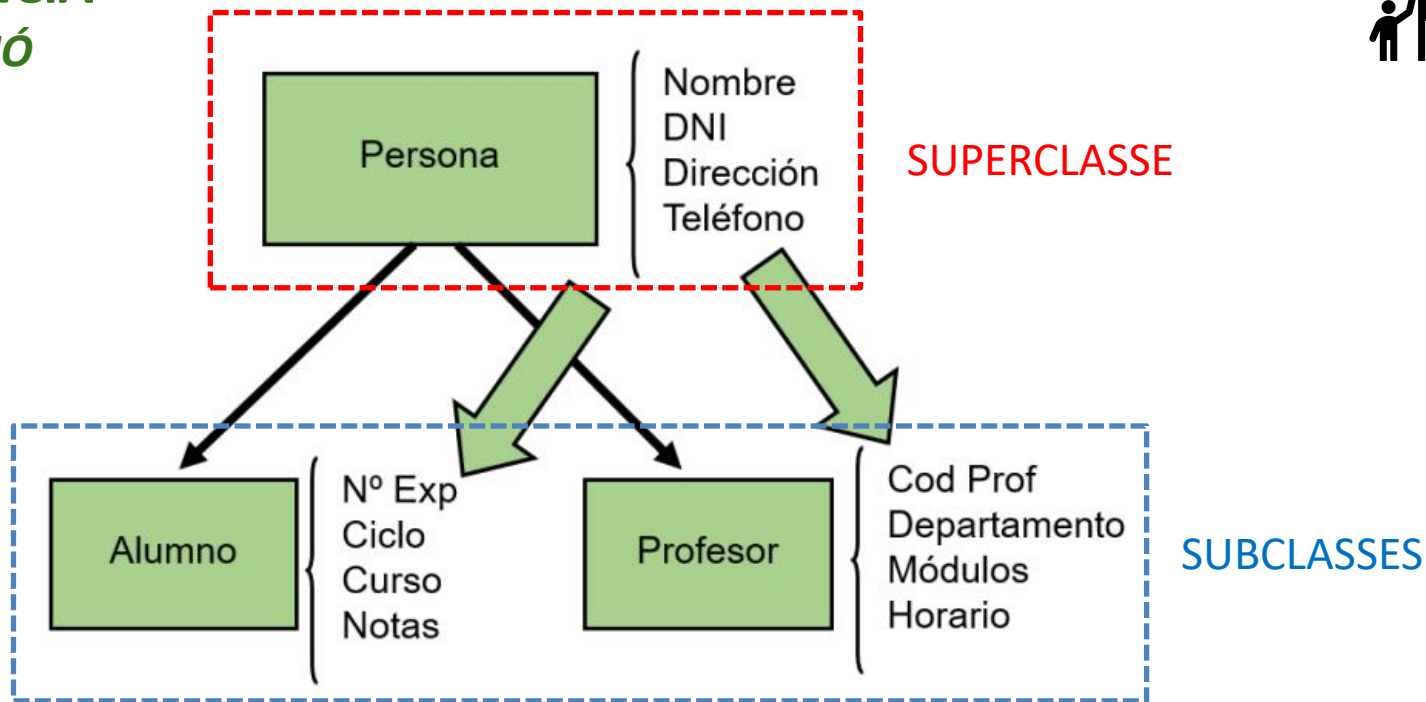
i per la seua part el professorat tindrà:

- Codi de professor
- Departament
- Mòduls que imparteix ...

Per tant, en aquest cas, el millor és declarar una **superclasse *Persona*** amb els atributs comuns (Nom, DNI, Adreça ...) i dos **subclasses *Alumne* i *Professor*** que hereten de ***Persona*** (a més de tindre els seus propis atributs)

### 3. HERÈNCIA

#### INTRODUCCIÓ



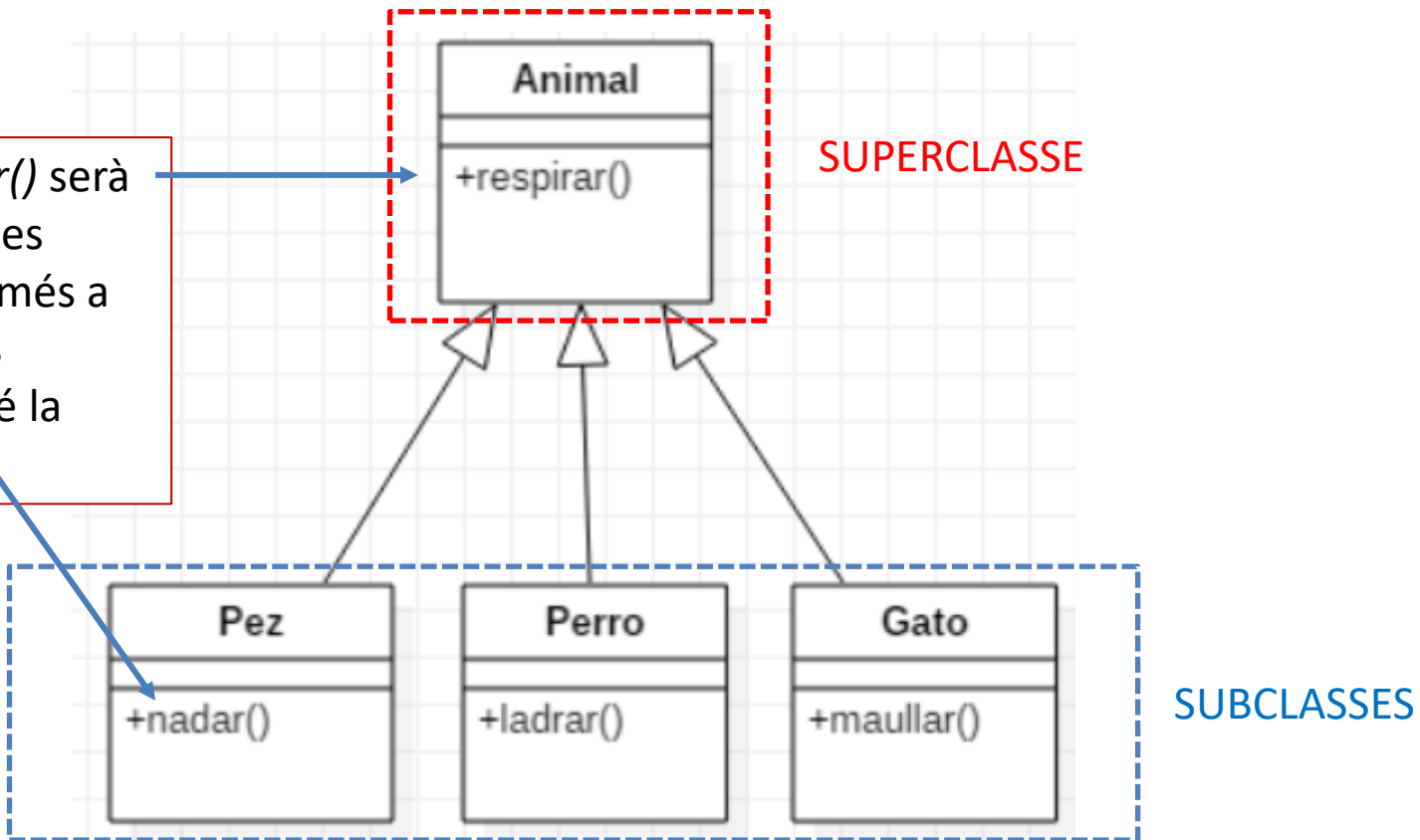
Es important remarcar que *Alumne* i *Professor*, a més d'heretar els atributs, **també heretaran tots els mètodes de *Persona***

### 3. HERÈNCIA

#### INTRODUCCIÓ



El mètode *respirar()* serà heretat per totes les subclasses, que a més a més tindran altres mètodes que no té la superclasse

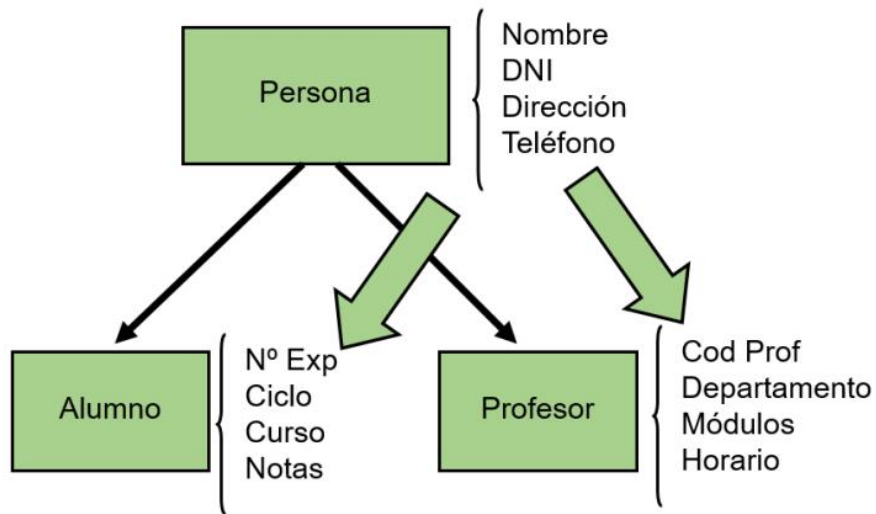


### 3. HERÈNCIA

#### INTRODUCCIÓ



En Java s'utilitza la paraula reservada ***extends*** per a indicar l'herència



```
public class Persona{
    ...
}

public class Alumno extends Persona{
    ...
}

public class Profesor extends Persona{
    ...
}
```

### 3. HERÈNCIA

#### CONSTRUCTORS DE CLASSES DERIVADES



- El constructor d'una **subclasse** ha d'**encarregar-se de construir els atributs que estiguen definits en la superclasse**, a més dels seus propis atributs
- Es fa utilitzant el mètode reservat ***super()***, passant-li com a argument els paràmetres que necessite
- Si no es crida explícitament al constructor de la superclasse ***super()***, el compilador cridarà automàticament al constructor per defecte
- Si no hi ha constructor per defecte → generarà un error de compilació



### 3. HERÈNCIA

#### CONSTRUCTORS DE CLASSES DERIVADES



```
public class Persona{  
    private String nom;  
    private String DNI;  
  
    public Persona(String nom, String DNI){  
        this.nom = nom;  
        this.DNI = DNI;  
    }  
}
```

Constructor de la  
superclasse *Persona*

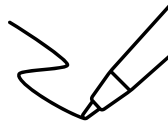
Constructor de la subclasse *Alumne* que  
ha de construir els atributs de la  
superclasse ... i els seus propis

```
public class Alumne extends Persona{  
    private String assignatura;  
    private double nota;  
  
    public Alumne(String nom, String DNI, String assignatura, double nota){  
        super(nom, DNI);  
        this.assignatura = assignatura;  
        this.nota = nota;  
    }  
}
```

DIY

### 3. HERÈNCIA

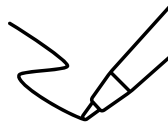
#### MÈTODES HERETATS I SOBRESCRITS



- Hem vist que una subclasse hereta els atributs i mètodes de la superclasse i a més, **es poden incloure nous atributs i nous mètodes**
- Pot ocórrer que un mètode d'una superclasse no ens servisca per a la subclasse (tal com està programat) i **necessitem adequar el mètode a les característiques de la subclasse**
- Això pot fer-se mitjançant la sobreescritura de mètodes
- Un mètode està sobreescrit o reimplementat **quan es programa de nou en la classe derivada**
- En Java podem accedir a mètodes definits en la superclasse mitjançant ***super.mètode()***

### 3. HERÈNCIA

#### MÈTODES HERETATS I SOBRESCRITS



El mètode *mostrarPersona()* de la classe *Persona* s'ha de sobreescriure en *Alumne* per a mostrar també els atributs d'*Alumne*

```
public class Persona{
    private String nom;
    private String DNI;

    public Persona(String nom, String DNI){
        this.nom = nom;
        this.DNI = DNI;
    }

    public void mostrarPersona(){
        System.out.println("Nom: " + this.getNom());
        System.out.println("DNI: " + this.getDNI());
    }
}
```

```
public class Alumne extends Persona{
    private String assignatura;
    private double nota;

    public Alumne(String nom, String DNI, String assignatura, double nota){
        super(nom, DNI);
        this.assignatura = assignatura;
        this.nota = nota;
    }

    public void mostrarPersona(){
        super.mostrarPersona(); // Cridem al mètode de la Superclasse

        System.out.println("Assignatura: " + this.getAssignatura());
        System.out.println("Nota: " + this.getNota());
    }
}
```

Què mostrarà el mètode  
*Alumne.mostrarPersona()* ?

DIY

### 3. HERÈNCIA

#### CLASSES I MÈTODES FINAL



p.e. *String* (intenta-ho)



Una classe **final** no pot ser heretada.

[\(més info\)](#)



Un mètode **final** no pot ser sobreescrit per les subclasses.

```
public class Persona {
    private String nom;
    private String DNI;

    public Persona(String nom, String DNI) {
        this.nom = nom;
        this.DNI = DNI;
    }

    final public void mostrarPersona() {
        System.out.println("Nom: " + this.getNom());
        System.out.println("DNI: " + this.getDNI());
    }
}
```

```
public class Alumne extends Persona {
    private String assignatura;
    private double nota;

    public Alumne(String nom, String DNI, String assignatura, double nota) {
        super(nom, DNI);
        this.assignatura = assignatura;
        this.nota = nota;
    }

    public void mostrarPersona() {
        super.mostrarPersona(); // Cridem al mètode de la Superclasse

        System.out.println("Assignatura: " + this.getAssignatura());
        System.out.println("Nota: " + this.getNota());
    }
}
```

### 3. HERÈNCIA

#### ACCÉS A MEMBRES DERIVATS



- Encara que una subclasse inclou tots els membres de la seua superclasse, **no podrà accedir a aquells que hagen sigut declarats com *private***
- Si declarem els atributs com *protected*, podran ser accedits des de les classes heretades (però mai des d'altres classes)



Els atributs declarats com *protected* son públics per a les classes heretades i privats per a les altres classes.

### 3. HERÈNCIA

#### ACCÉS A MEMBRES DERIVATS



DIY

... i juga canviant les visibilitats observant l'efecte

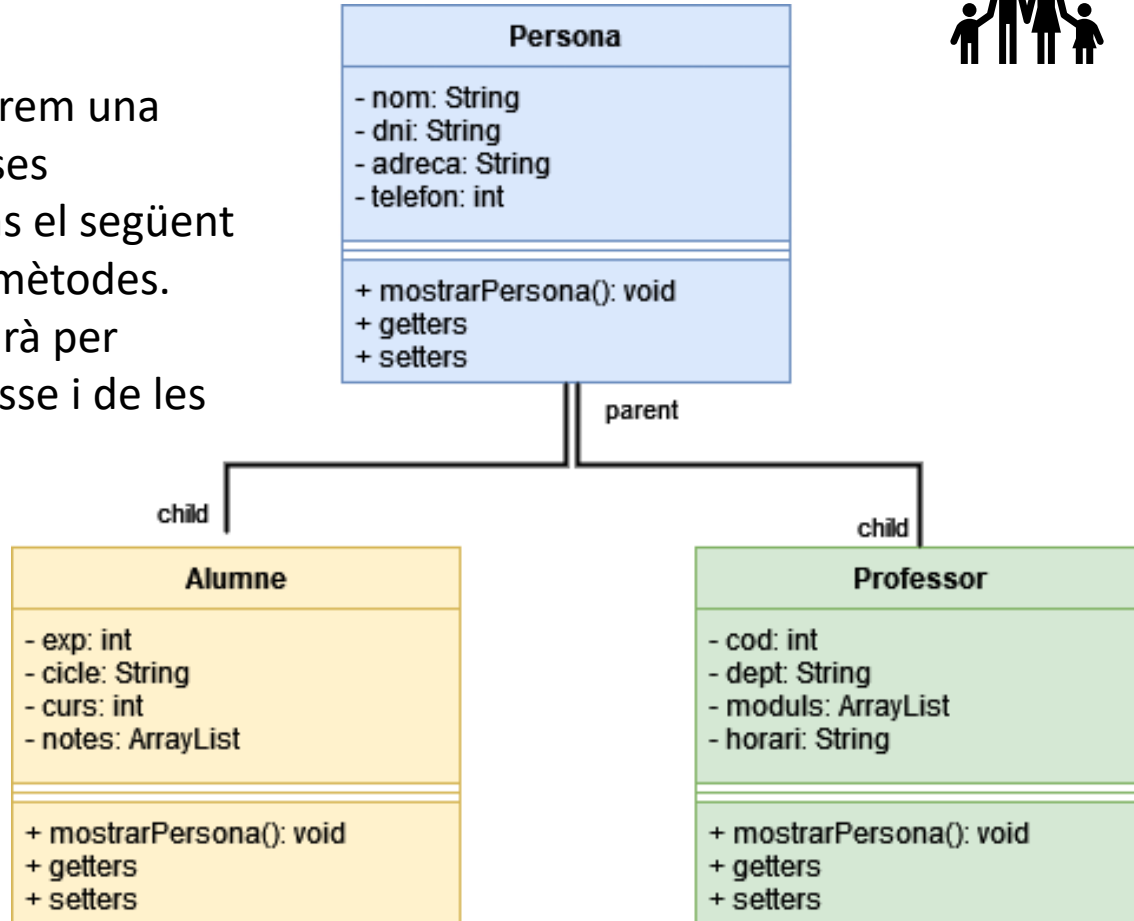
```
class Persona2 {  
    private String nom;  
    protected String DNI;  
  
    private void mostrarPersona() {  
        System.out.println("Nom: " +  
        System.out.println("DNI: " +  
    }  
}
```

```
class Alumne extends Persona2 {  
    private String assignatura;  
    private double nota;  
    public Alumne(String nom, String DNI, String assignatura, double nota) {  
        super(nom, DNI);  
        this.assignatura = assignatura;  
        this.nota = nota;  
    }  
  
    super.nom = "Pere";  
    super.DNI = "12345678s";  
  
    public void mostrarPersona() {  
        super.mostrarPersona(); // Cridem al mètode de la Superclasse  
        System.out.println("Assignatura: " + this.getAssignatura());  
        System.out.println("Nota: " + this.getNota());  
    }  
}
```

### 3. HERÈNCIA



**Exemple 4:** En aquest exemple crearem una nova classe *Persona* i les seues classes heretades *Alumne* i *Professor* segons el següent diagrama de classes amb atributs i mètodes. El mètode *mostrarPersona()* mostrarà per pantalla els atributs de la pròpia classe i de les ascendents (incloent els *ArrayList*).



Al programa principal s'instanciarà:

- una *Persona*
  - un *Alumne* amb 3 notes
  - un *Professor* amb 3 mòduls
- i es mostraran tots per pantalla

# POO - II

## ÍNDEX DE CONTINGUTS

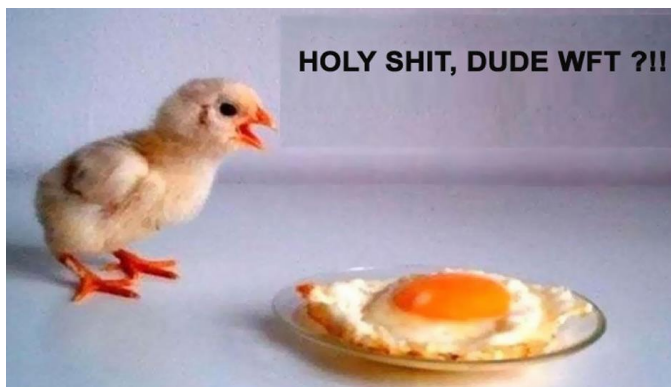
1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. HERÈNCIA
4. **POLIMORFISME**
5. CLASSES ABSTRACTES
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES



## 4. POLIMORFISME



- La sobreescritura de mètodes constitueix la base de la **selecció dinàmica de mètodes** ...
- ... que significa que la crida a un mètode sobreescrit es resol en temps d'execució (no durant compilació)
- El **polimorfisme** permet que una **classe general especifique mètodes que seran comuns a totes les classes que es deriven d'aquesta** ...
- ... i **les subclasses podran definir o modificar la seua implementació**
- D'aquesta manera, combinant l'herència i la sobreescritura de mètodes, una superclasse pot definir la forma general dels mètodes que s'usaran en totes les seues subclasses



S'entén millor amb un exemple



## 4. POLIMORFISME



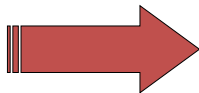
### Exemple 5:

Crear la **classe *Mare*** amb un mètode *crida()*. A continuació crearem **2 classes derivades** d'aquesta: ***Filla1*** i ***Filla2***, sobreescrivint el mètode *crida()*.

En el *main* crearem un objecte de cada classe i posteriorment els assignarem a una 4a variable de tipus *Mare* (anomenada *mare2*) amb la qual utilitzarem el mètode *crida()* dels tres objectes per a voler les diferències.



Pas a pas



## 4. POLIMORFISME



### Exemple 5:

Crear la **classe Mare** amb un mètode *crida()*. A continuació crearem **2 classes derivades** d'aquesta: **Filla1** i **Filla2**, sobreescrivint el mètode *crida()*

```
class Mare{  
    public void crida(){  
        System.out.println("Estic a la classe Mare");  
    }  
}
```

```
class Filla1 extends Mare{  
    public void crida(){  
        System.out.println("Estic a la classe Filla1");  
    }  
}  
  
class Filla2 extends Mare{  
    public void crida(){  
        System.out.println("Estic a la classe Filla2");  
    }  
}
```

## 4. POLIMORFISME



### Exemple 5:

En el *main* crearem un objecte de cada classe i posteriorment els assignarem a una 4a variable de tipus *Mare* (anomenada *mare2*) amb la qual utilitzarem el mètode *crida()* dels tres objectes per a voler les diferències.

```
public static void main(String[] args) {  
    Mare mare1 = new Mare();  
    Filla1 filla1 = new Filla1();  
    Filla2 filla2 = new Filla2();  
  
    System.out.print("Variable mare1: ");  
    mare1.crida();  
  
    // Creem una altra variable de tipus Mare  
    Mare mare2;  
    mare2 = mare1;  
    System.out.print("Variable mare2: ");  
    mare2.crida();  
}
```

```
//Assignem a mare2 un objecte de tipus Filla1  
mare2 = filla1;  
System.out.print("Variable mare2: ");  
mare2.crida();  
  
//Assignem a mare2 un objecte de tipus Filla2  
mare2 = filla2;  
System.out.print("Variable mare2: ");  
mare2.crida();
```

Explicació del que ha passat



## 4. POLIMORFISME



### Exemple 5:

En el *main* crearem un objecte de cada classe i posteriorment els assignarem a una 4a variable de tipus *Mare* (anomenada *mare2*) amb la qual utilitzarem el mètode *crida()* dels tres objectes per a vorer les diferències.

```
//Assignem a mare2 un objecte de tipus Filla1
mare2 = filla1;

System.out.print("Variable mare2: ");
mare2.crida();

//Assignem a mare2 un objecte de tipus Filla2
mare2 = filla2;
System.out.print("Variable mare2: ");
mare2.crida();
```

*mare2* es pot assignar a objectes de classe *Filla1* i *Filla2* perquè *Filla1* i *Filla2* també són de tipus *Mare*  
→ gràcies a l'**Herència**

*mare2* cridarà al mètode *crida()* de la classe de l'objecte al qual fa referència (*Filla1* o *Filla2*) → gràcies al **Polimorfisme**

## 4. POLIMORFISME



### Exemple 5:

Crea a continuació 3 nous mètodes crida() en la classe Filla2:

- El 1r mètode crida tindrà un atribut d'entrada 'edat' per a passar l'edat de la filla i mostrar-la per pantalla
- El 2n mètode tindrà el mateix atribut 'edat' i un String 'feina' per a definir la feina que fa. Es mostrarà igualment tota aquesta info a l'utilitzar el mètode crida()
- El 3r mètode tindrà l'atribut 'edat' i un double anomenat també 'feina' per a definir el seu sou i mostrar-lo per pantalla

Modifica el main per a utilitzar les noves implementacions de crida()

## 4. POLIMORFISME



### Exemple 5:

Crea a continuació 3 nous mètodes crida() en la classe Filla2:

- El 1r mètode crida tindrà un atribut d'entrada 'edat' per a passar l'edat de la filla i mostrar-la per pantalla
- El 2n mètode tindrà el mateix atribut 'edat' i un String 'feina' per a definir la feina que fa. Es mostrarà igualment tota aquesta info a l'utilitzar el mètode crida()
- El 3r mètode tindrà l'atribut 'edat' i un double anomenat també 'feina' per a definir el seu sou i mostrar-lo per pantalla

Modifica el main per a utilitzar les noves implementacions de crida()

**QUI M'EXPLICA EL QUE PASSA?**

*El mètode crida() te diferents implementacions i EN TEMPS D'EXECUCIÓ es triarà quina implementació utilitzar*

# POO - II

## ÍNDEX DE CONTINGUTS

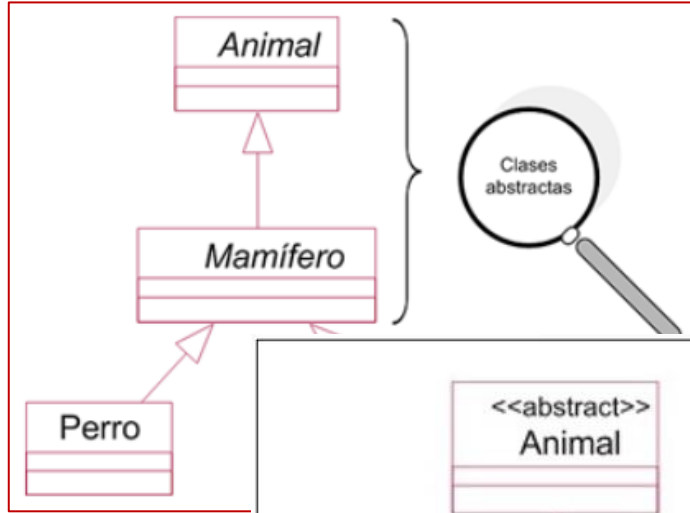
1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. HERÈNCIA
4. POLIMORFISME
5. **CLASSES ABSTRACTES**
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES



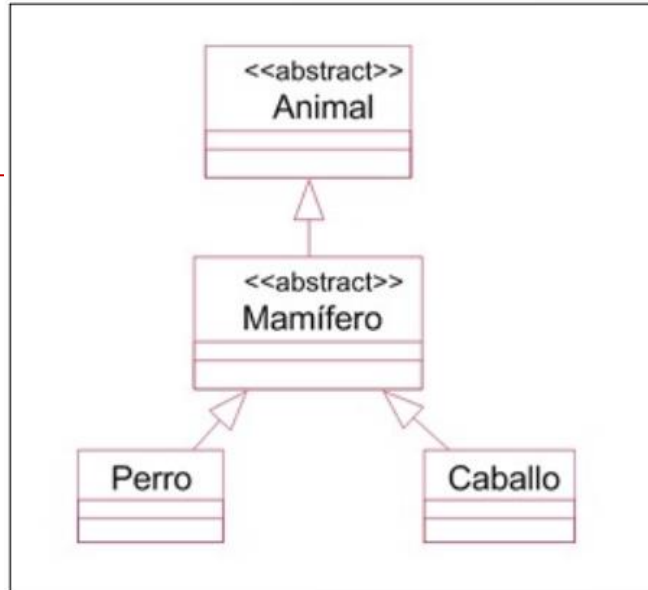
## 5. CLASSES ABSTRACTES



## 5. CLASSES ABSTRACTES



En el món real hi ha Gossos, Cavalls ... però el concepte d'animal pròpiament dit és abstracte, igual que el de mamífer



La classe *Animal* no seria suficient per a definir completament a un animal concret  
→ seria una classe Abstracta

La finalitat de les classes Abstractes és factoritzar atributs i mètodes comuns a les subclasses que les concreten

## 5. CLASSES ABSTRACTES



- ❑ Una classe abstracta és **una classe que declara l'existència d'alguns mètodes però no la seua implementació**
- ❑ Conté la capçalera del mètode però no el seu codi
- ❑ Una classe abstracta pot contindre tant mètodes abstractes (sense implementar) com no abstractes (implementats), però **almenys un mètode ha de ser abstracte**
- ❑ **Per a declarar una classe o mètode** com a abstracte s'utilitza el **modificador abstract**

⚡ Una classe abstracta **no es pot instanciar**, però **sí es pot heretar**. Les subclasses hauran d'implementar obligatòriament el codi dels mètodes abstractes (llevat que també es declaren com a abstractes).

⚡ No poden declarar-se constructors o mètodes estàtics abstractes.

## 5. CLASSES ABSTRACTES



- ❑ Les classes abstractes són útils quan necessitem definir una forma generalitzada de classe que serà compartida per les subclasses, deixant **part del codi en la classe abstracta (mètodes “normals”)** i delegant una altra **part en les subclasses (mètodes abstractes)**
- ❑ En la pràctica **és obligatori aplicar herència**, si no, la classe abstracta no serveix per a res
- ❑ Pot una classe ser *final + abstract* ?

```
abstract class Principal {  
    // Mètode concret amb implementació  
  
    public void imprimirNom(String nom) {  
        System.out.println("El nom és: " + nom);  
    }  
  
    // Mètode abstracte sense implementació  
    public abstract void imprimirClasse();  
}
```

Sense implementar

```
class Secundaria extends Principal {  
    // Implementació concreta  
    @Override  
    public void imprimirClasse() {  
        System.out.println("Classe Secundària");  
    }  
}
```

Implementat

## 5. CLASSES ABSTRACTES



Msc, Alex Narváez

## 5. CLASSES ABSTRACTES



- ❑ Les classes abstractes són útils quan necessitem definir una forma generalitzada de classe que serà compartida per les subclasses, deixant **part del codi en la classe abstracta (mètodes “normals”)** i delegant una altra **part en les subclasses (mètodes abstractes)**
- ❑ En la pràctica **és obligatori aplicar herència**, si no, la classe abstracta no serveix per a res
- ❑ Pot una classe ser *final + abstract* ?

```
abstract class Principal {  
    // Mètode concret amb implementació  
  
    public void imprimirNom(String nom) {  
        System.out.println("El nom és: " + nom);  
    }  
  
    // Mètode abstracte sense implementació  
    public abstract void imprimirClasse();  
}
```

Sense implementar

```
class Secundaria extends Principal {  
    // Implementació concreta  
    @Override  
    public void imprimirClasse() {  
        System.out.println("Classe Secundària");  
    }  
}
```

Implementat

DIY ... i juga amb  
els modificadors



# POO - II

## ÍNDEX DE CONTINGUTS

1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. HERÈNCIA
4. POLIMORFISME
5. CLASSES ABSTRACTES
6. **INTERFÍCIES**
7. ENTITATS GENÈRIQUES

## 6. INTERFÍCIES



Una interfície és una declaració d'atributs i mètodes sense implementació.

- ❑ Si una classe és una plantilla per a crear objectes, una interfície és una plantilla per a crear classes
- ❑ S'utilitzen per a **definir el conjunt mínim d'atributs i mètodes** de les classes que implementen aquesta interfície
- ❑ En certa manera, és paregut a una **classe abstracta amb tots els seus membres abstractes**
- ❑ En una interfície també **es poden declarar constants** que defineixen el comportament que han de suportar els objectes que vulguen implementar la interfície

```
public interface Figura {  
    float PI = 3.1416f;  
  
    public float area();  
}
```

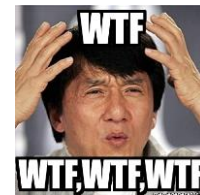
```
public class Circulo implements Figura{  
    private float diametro;  
    public Circulo(float diametro){  
        this.diametro = diametro;  
    }  
    public float area(){  
        return PI*diametro*diametro/4f;  
    }  
}
```



## 6. INTERFÍCIES



- ❑ Avantatges de l'ús de les interfícies:
  - **Afavorir el manteniment i l'extensió de les aplicacions** → en definir interfícies permetem l'ús de variables polimòrfiques i la invocació polimòrfica de mètodes
  - **Separar l'especificació d'una classe** (què fa) **de la implementació** (com ho fa) → dona lloc a programes més robustos i amb menys errors
- ❑ Una interfície **no es pot instanciar en objectes**, només serveix per a implementar classes
- ❑ Una classe pot implementar diverses interfícies (separades per comes)
- ❑ Una classe que implementa una interfície **ha de proporcionar implementació per a tots i cadascun dels mètodes** definits en la interfície
- ❑ Les classes que implementen una interfície que té definides constants poden usar-les en qualsevol part del codi de la classe



→  
Exemple

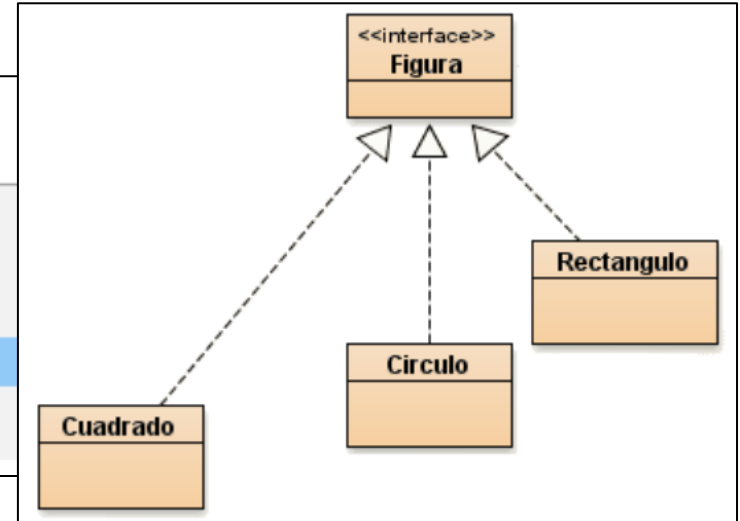
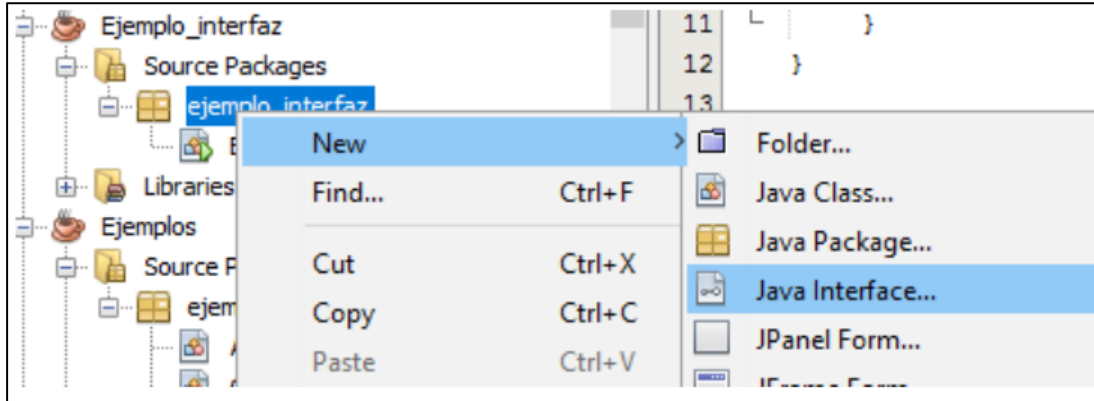
## 6. INTERFÍCIES



### Exemple 6:

Crea una interfície *Figura* i posteriorment implementa les classes: *Quadrat*, *Rectangle* i *Cercle*. La interfície ha de definir una constant *PI* i un mètode *area()* que s'ha d'implementar en cada classe segons corresponga.

Les classes han de disposar d'un constructor per a recollir els atributs necessaris per al mètode *area()*.



Crea ara les 4 classes ...

## 6. INTERFÍCIES



### Exemple 6:

Al programa principal, crea un *ArrayList* de *Figures* i afegeix 3 quadrats, 2 cercles i 2 rectangles. Finalment mostra un missatge com el següent:

“Tenim un total de XX figures i la seua àrea total és de YY unitats quadrades”

```
cuad1 = new Cuadrado(3.5f);  
cuad2 = new Cuadrado(2.2f);  
cuad3 = new Cuadrado(8.9f);  
  
cir1 = new Circulo(3.5f);  
cir2 = new Circulo(4f);  
  
rect1 = new Rectangulo(2.25f,2.55f);  
rect2 = new Rectangulo(12f,3f);
```

```
ArrayList serieDeFiguras = new ArrayList();  
  
serieDeFiguras.add(cuad1);  
serieDeFiguras.add(cuad2);  
serieDeFiguras.add(cuad3);  
serieDeFiguras.add(cir1);  
serieDeFiguras.add(cir2);  
serieDeFiguras.add(rect1);  
serieDeFiguras.add(rect2);
```

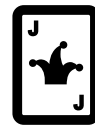
## Fes l'Exercici 1 - Aliment

# POO - II

## ÍNDEX DE CONTINGUTS

1. LA CLASSE *ARRAYLIST*
2. COMPOSICIÓ
3. HERÈNCIA
4. POLIMORFISME
5. CLASSES ABSTRACTES
6. INTERFÍCIES
7. ENTITATS GENÈRIQUES

## 7. ENTITATS GENÈRIQUES



- Una **classe**, una **interfície** o un **mètode** que opera amb un **tipus parametritzat** es coneix com a **entitat genèrica**
- La idea és **permetre que el tipus siga un paràmetre** per a mètodes, classes i interfícies
- Amb l'ús de genèrics és possible crear classes que funcionen amb diferents tipus de dades
- Per exemple, classes com *ArrayList* utilitzen molt bé els genèrics

```
// To create an instance of generic class  
BaseType <Type> obj = new BaseType <Type>()
```

```
ArrayList<String> Usuaris = new ArrayList<String>( )
```

```
ArrayList<String> Usuaris = new ArrayList<>( )
```

```
ArrayList<Aliment> Llista_Compra = new ArrayList<Aliment>( )
```

## 7. ENTITATS GENÈRIQUES



```
public class Empleat<T> {    // La classe genèrica Empleat accepta qualsevol tipus
    T sou;    // El paràmetre sou accepta també qualsevol tipus
```

```
    // Constructor
```

```
    public Empleat(T sou) {
        this.sou = sou;
    }
```

```
    // Mètode print
```

```
    public void print() {
        System.out.println("El sou de l'empleat és: " + sou);
    }
}
```

DIY

```
public static void main(String[] args) {
    int sou = 1000;
    Empleat empleat1 = new Empleat(sou);
    empleat1.print();
```

```
    double sou2 = 2500.55;
    Empleat empleat2 = new Empleat(sou2);
    empleat2.print();
```

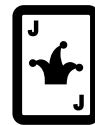
```
    String sou3 = "sense sou";
    Empleat empleat3 = new Empleat(sou3);
    empleat3.print();
}
```

run:

```
El sou de l'empleat és: 1000
El sou de l'empleat és: 2500.55
El sou de l'empleat és: sense sou
```



## 7. ENTITATS GENÈRIQUES



### Característiques de l'ús de genèrics:

- l'argument de tipus passat ha de ser un tipus de referència. No podem utilitzar primitius (*int, double, char ...*)

```
Test<int> obj = new Test<int>(20);
```

← però si la classe *Integer*

- Reutilització del codi: podem escriure un mètode/classe/interfície una vegada i utilitzar-lo per a qualsevol tipus que vulguem
- Seguretat de tipus: utilitzant genèrics, els errors apareixen en temps de compilació i no en temps d'execució (sempre és millor conèixer els problemes del vostre codi en temps de compilació en lloc de fer que el codi falle en temps d'execució)



Exemple



## 7. ENTITATS GENÈRIQUES



Suposem que voleu crear una *ArrayList* que emmagatzeme el nom dels estudiants.  
Si per error el programador afegeix un objecte *int* en lloc d'un *String*, el compilador ho permet  
→ però, quan recuperem aquestes dades, es produeixen problemes en temps d'execució

DIY

```
ArrayList al = new ArrayList();

al.add("Sachin");
al.add("Rahul");
al.add(10); // Compiler allows this

String s1 = (String)al.get(0);
String s2 = (String)al.get(1);

// Causes Runtime Exception
String s3 = (String)al.get(2);
```

El compilador s'ho "traga"

Però el programa  
fallarà en execució

## 7. ENTITATS GENÈRIQUES



SOLUCIÓ: definim que la llista només pot incloure *Strings* i el compilador no permetrà afegir un *int*

```
ArrayList<String> al = new ArrayList<String> ();  
  
al.add("Sachin");  
al.add("Rahul");  
  
// Now Compiler doesn't allow this  
al.add(10);  
  
String s1 = (String)al.get(0);  
String s2 = (String)al.get(1);  
String s3 = (String)al.get(2);
```

El compilador donarà error

DIY

## 7. ENTITATS GENÈRIQUES



### Característiques de l'ús de genèrics:

- No cal fer *type casting*: si no fem servir genèrics, a l'exemple anterior, cada vegada que recuperem dades de l'*ArrayList*, haurem de fer *casting*

```
ArrayList al = new ArrayList();

al.add("Sachin");
al.add("Rahul");
al.add(10); // Compiler allows this

String s1 = (String)al.get(0);
String s2 = (String)al.get(1);
```

```
ArrayList<String> al = new ArrayList<String>();

al.add("Sachin");
al.add("Rahul");

// Typecasting is not needed
String s1 = al.get(0);
String s2 = al.get(1);
```

DIY

Si ja sabem que la nostra llista només conté *Strings*, no caldrà fer *casting* cada vegada ... i en el cas d'objectes, a més a més podrem accedir directament als seus mètodes i variables

## 7. ENTITATS GENÈRIQUES



### Característiques de l'ús de genèrics:

- No cal fer *type casting*: si no fem servir genèrics, a l'exemple anterior, cada vegada que recuperem dades de l'*ArrayList*, haurem de fer *casting*

Recupera l'exemple de interfícies (Figura) i modifica'l per a utilitzar entitats genèriques

## 7. ENTITATS GENÈRIQUES



### Característiques de l'ús de genèrics:

- Es poden crear classes amb múltiples tipus parametritzats

```
class Test<T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```
public static void main (String[] args)
{
    Test <String, Integer> obj =
        new Test<String, Integer>("GfG", 15);

    obj.print();
}
```

## BONUS TRACK → Arguments variables



Els arguments variables simplifiquen la creació de **mètodes que necessiten prendre un nombre variable d'arguments** (incloent el cas de cap argument)

```
public static void fun(int... a){  
    System.out.println("Nombre d'arguments: " + a.length);  
    for (int i : a)  
        System.out.print(i + " ");  
    System.out.println();  
}
```

DIY

```
fun(100);  
fun(1, 2, 3, 4);  
fun();
```



```
Nombre d'arguments: 1  
100  
Nombre d'arguments: 4  
1 2 3 4  
Nombre d'arguments: 0
```

BUILD SUCCESSFUL (total time: 0 seconds)

## BONUS TRACK → Arguments variables



Un mètode pot combinar **paràmetres variables** amb altres paràmetres, però s'ha d'assegurar que només existeix un paràmetre *varargs* que s'ha d'escriure l'últim a la llista de paràmetres

```
public static void fun2(String str, double doub, int... a){  
    System.out.println("String: " + str);  
    System.out.println("Double: " + doub);  
    System.out.println("Nombre d'arguments: "+ a.length);  
  
    for (int i : a)  
        System.out.print(i + " ");  
    System.out.println("\n");  
}
```

DIY

```
fun2("funció 2",55.5, 100, 200);  
fun2("funció 2", 33.3, 1, 2, 3, 4, 5);  
fun2("funció 2",22.2);
```

**Fes la resta d'Exercicis**



## Autor:

Àngel Olmos Giner

segons el material de Carlos Cacho i Raquel Torres  
i el portal [geeksforgeeks](#)



## Llicència:

**CC BY-NC-SA 3.0 ES Reconeixement – No Comercial – Compartir Igual (by-nc-sa)**

No es permet un ús comercial de l'obra original ni de les possibles obres derivades, la distribució de les quals s'ha de fer amb una llicència igual a la que regula l'obra original. Aquesta és una obra derivada de l'obra original de Carlos Cacho i Raquel Torres (Programació)