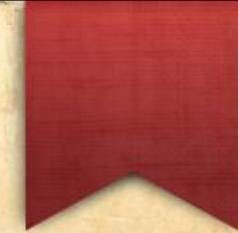


Programació



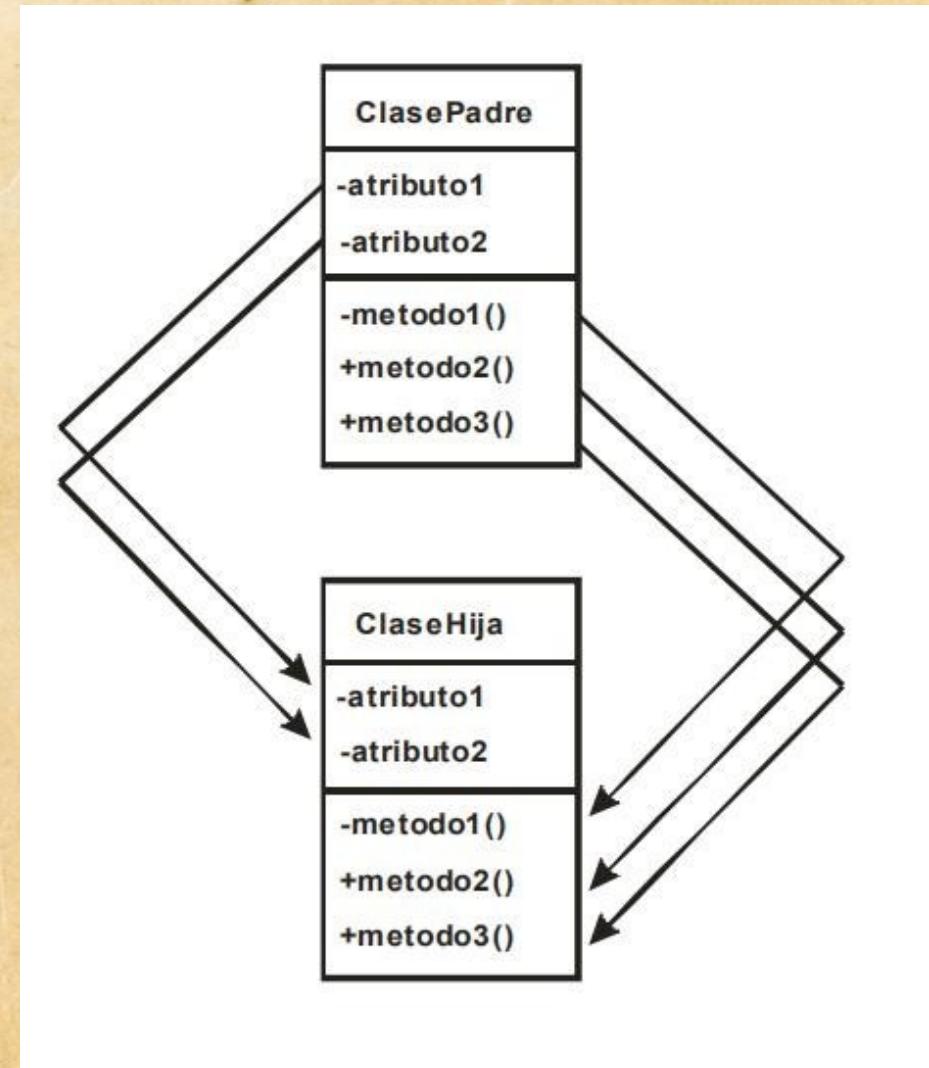
UT6.1. Herència

Introducció

- L'herència en tots els àmbits és TRANSMISIÓ
 - **Exemples**
 - Un parent llunyà mor i tots els seus béns, així com les seues obligacions es transmeten.
 - L'home i el mico segons Darwin hereten d'un ancestre comú, després cadascú afegeix les seues característiques pròpies

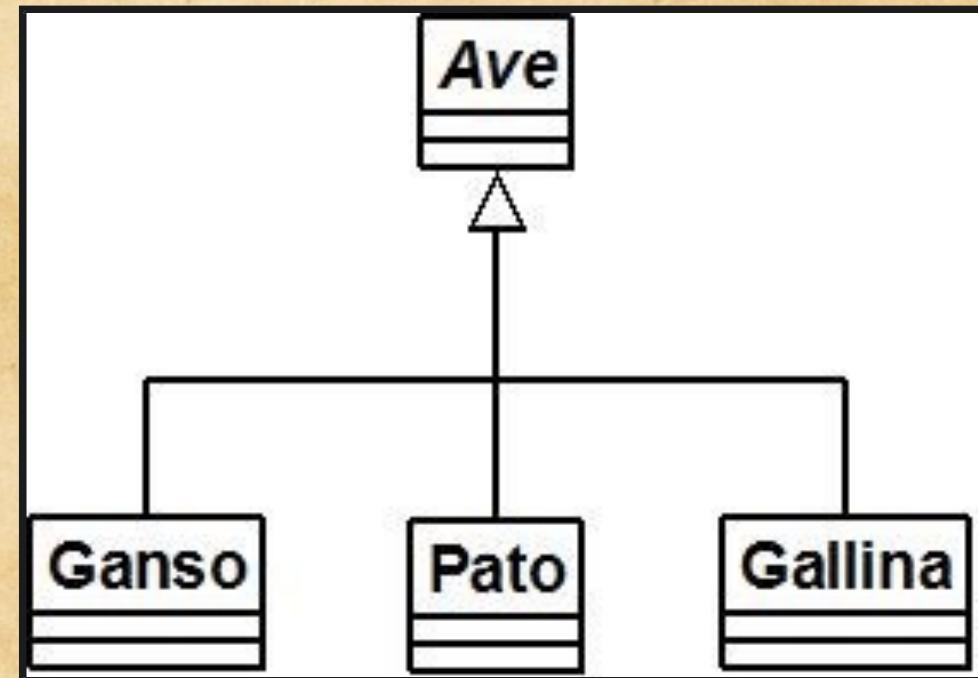
Relació d'herència

- En la POO, que una classe herete d'una altra significa que la classe heretada **transmet** a la classe hereva tot el que té.
- Es transmet tant la vista **pública** com la vista **privada**.
- Una definició poc rigorosa seria determinar que una herència seria un “copy-paste” d'una classe a una altra, però de manera **dinàmica**. (acoplament!)



Terminologia

- Per a fer referència a les classes implicades s'utilitza:
 - Terminologia genealògica:**Classe pare** front a **classe filla**
 - **Classe base** front a **classe derivada**.
 - **Superclasse** front a **subclasse**



Col·laboració entre objectes

- La **composició, agregació i associació** són relacions binàries que succeeixen de la **col·laboració entre objectes**
- Per altra banda, l'herència és una relació binària **entre classes**
- Encara que no siga habitual, és possible una col·laboració entre els objectes de les classes implicades en l'herència.

Exemple:

Classe pare *Animal*

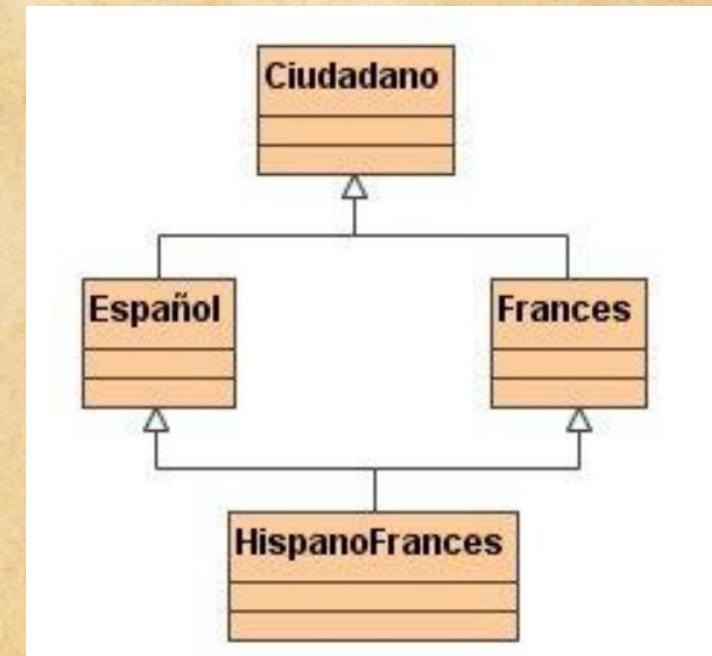
Classe filla *Persona*

- *En una app d'evolució d'espècies NO col·laboren.*
- *En una app d'una granja SÍ que col·laboren.*

Tipus de herència

- **Herència simple:** quan una classe derivada hereta d'una única classe base
- **Herència múltiple:** quan una classe derivada hereta de diverses classes base

Llenguatges com C++ o Python admeten l'herència múltiple mentre que altres com Java o PHP només admeten l'herència simple.

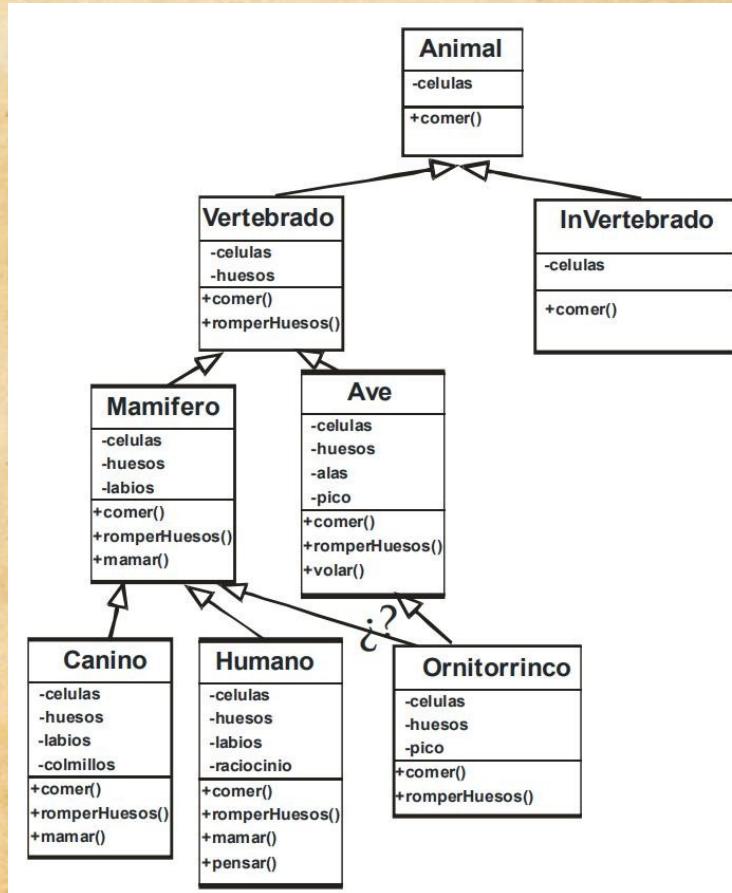


Jerarquies de classificació

L'objectiu principal de tota herència és realitzar **jerarquies de classificació**

Característiques:

- **Subjectes a subjectivitat**
(ex. pacients d'hospital)
- Poden contemplar elements **difícilment categoritzables**
(ex. ornitorrinc)
- És **impossible trobar la perfecció**



Regles de construcció

- **Regla de Generalització/Especificació:** les classes derivades no només han d'heretar TOTES les característiques de la classe base, sinó que deuen especialitzar-se en alguna cosa més.
- **Regla ÉS UN?**: respondre afirmativament a que un objecte de la classe filla és també un objecte de classe pare

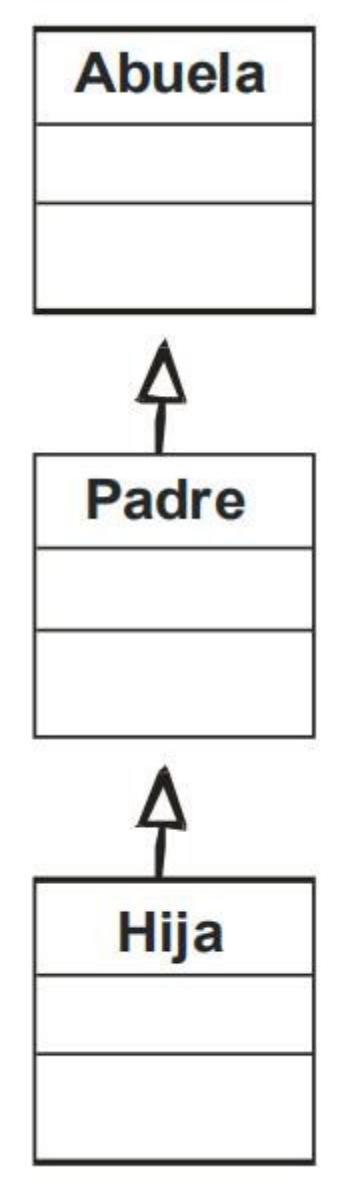
Herència en Java

```
public class <NomClasse> extends <NomClasseBase>
{
    <CosDeLaClasse>
}
```

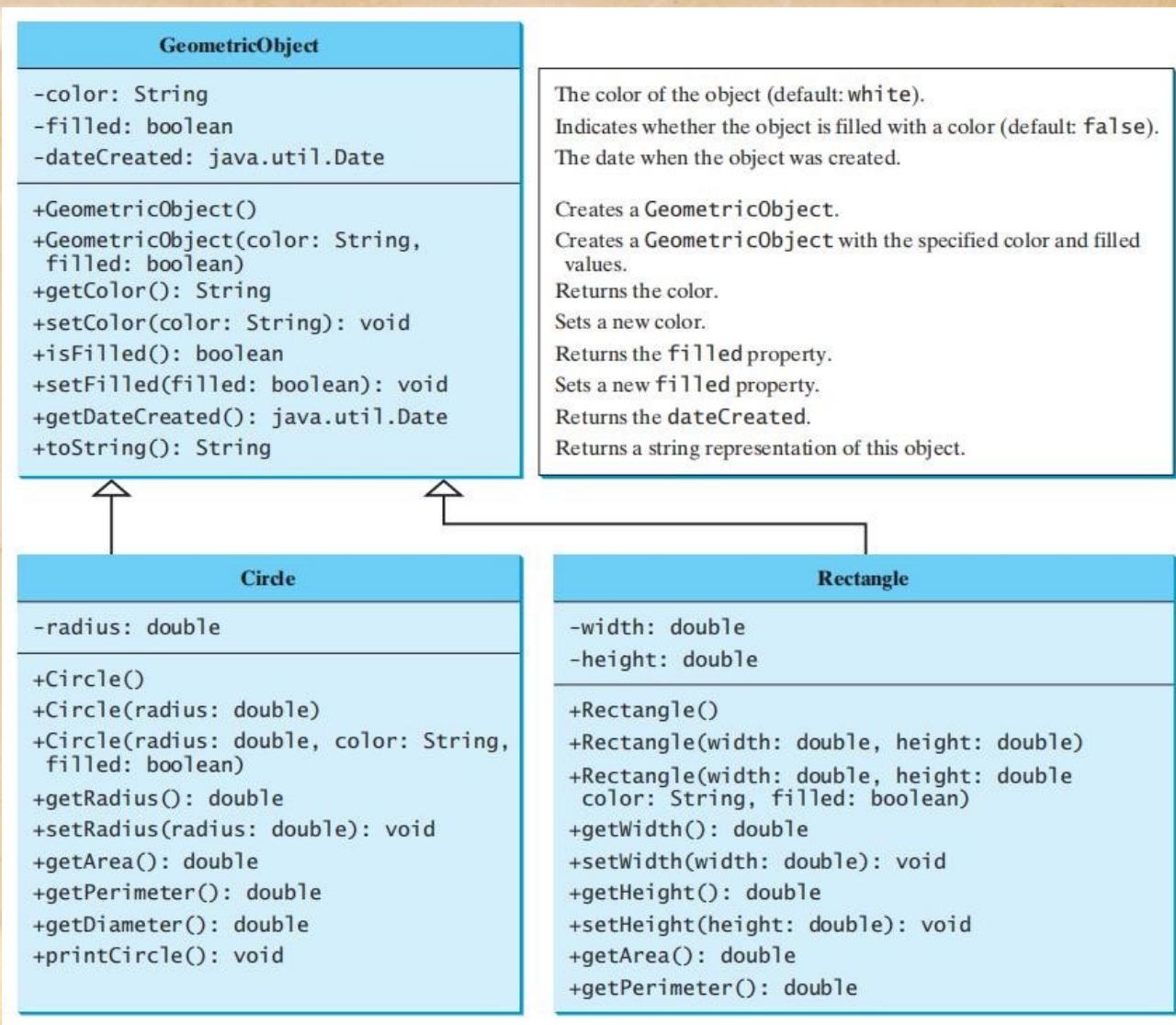
- Tots els membres (**dades i mètodes**) que té la classe base resideixen ara també en la classe derivada.
- Ara bé, només es podrà accedir des de la classe derivada als membres que en la classe base siguen **public**, **protected** i, en el seu cas, als de visibilitat **package**.

Herència en Java. Exemple

```
Ej.: class Abuela {  
    ...  
}  
  
class Padre extends Abuela {  
    ...  
}  
  
class Hija extends Padre {  
    ...  
}
```



Un altre esempio



Pràctica 1

Transforma el diagrama de la diapositiva anterior a codi

- Fes una classe `TestGeometricObject` on crearàs diversos cercles, i diversos rectangles, els llançaràs diversos missatges i imprimiràs el seu estat.
- Pots accedir als atributs de la classe pare `GeometricObject` des de la classe filla `Circle` o `Rectangle`?

Com especialitzar una classe?

```
public class <NomClasse> extends <NomClasseBase>
{
    <atributAfegit>
    ....
    <metodeAfegit>
}
```

1. Afegir **atributs** que tindràn les mateixes regles sintàctiques i semàntiques que qualsevol altra classe no derivada.
2. Afegir **mètodes**, similar als atributs afegits, excepte que NO tenen accés als atributs i mètodes privats transmesos des de la classe pare.

Implicacions de l'herència

- Els objectes de la classe pare NO pateixen cap alteració
 - Els objectes de la classe filla:
 - Disposen de tots els atributs de la classe pare, junt als propis
 - Responen a missatges dels mètodes públics transmesos des de la classe pare junt amb els propis
- Per tant:
- 
- Els objectes de la classe filla **contenen els atributs privats** transmesos des de la classe pare **però no es té accés a ells**
 - D'eixa manera, es manté un **mínim acoblament** entre la classe pare i la filla.

Problema

- La classe pare no disposa de mètodes públics necessaris per a manipular els atributs privats de la manera que es necessita des de la classe derivada
- Eixos mètodes no es van necessitar realitzar a la classe pare per el principi d'**encapsulació de la POO**
- Per tant, si no es fa res, la classe filla no podria manipular de la manera que només ella vol els atributs de la classe pare.

1a Solució

- **Usar la visibilitat “public” per als mètodes que manipulen directament els atributs privats.**

DITA SOLUCIÓ, ENCARA QUE ÉS POSSIBLE DE REALITZAR **NO ÉS UNA BONA OPCIÓ** JA QUE TRENCA EL PRINCIPI D'ENCAPSULACIÓ(els objectes de la classe pare donen a conèixer informació que no es proporcionava abans de l'existència de la classe derivada).

2a Solució

- Es proposa una nova visibilitat: **protected**
- Els membres **protected** són **accessibles des de la propia classe on es defineixen, així com des de les seues classes derivades**
- El més “formal” és realitzar **mètodes “get/set” de tipus protected** que obtinguen/modifiquen els atributs de classes pare
- D’eixa manera, si es modifiquen els atributs de la classe pare, no es veurien afectats a la classe filla (mínim acoblament)

3a i solució més estesa

- Degut a la falta de practicitat de la segona opció, es va optar, encara de una manera menys formal, per **fer directament els atributs de les classes pare de tipus protected.**
- Això sí, si es modifiquen els atributs de la classe pare, ara sí que caldrà modificar també la implantació de les classes filles on s'accedeix a eixos atributs protegits.
- **Optem per un augment de l'acoblament a canvi d'un augment de la practicitat.**

Una altra forma d'especialització

- Hem vist que afegir atributs i mètodes eren dos maneres d'especialitzar una classe derivada.
- Una altra manera és la **SOBREESCRIPTURA DE MÈTODES**

```
public class <NomClasse> extends <NomClasseBase> {  
    <mètodeSobreescrit>  
}
```

On la capçalera del mètode és exactament igual que la de la classe base, excepte la seu visibilitat, que pot ampliar-se.

La sobreescritura també es coneix com a “**redefinició**”
Podem diferenciar entre “**refinament**” i “**reemplaçament**”

Implicacions de la redefinició de mètodes

1. La **visibilitat del mètode de la classe pare NO POT SER PRIVAT.**
2. S'**anul·la la transmissió del mètode** de la classe pare (a partir d'eixe moment es transmet el mètode redefinit)
3. Els objectes de la classe pare responen al missatge amb el comportament de la classe pare
4. Els objectes de la classe filla responen al missatge amb el comportament de la classe filla

Reutilització del mètode de la classe pare sobreescrit

- En este cas, passa que tot i la redefinició que **ANUL·LA la transmissió** del mètode a la classe derivada, es pot executar el codi del mètode anul·lat.
- Per això, es disposa de la paraula reservada **super**, que és una **referència** constant que guarda la direcció de l'objecte que rep el missatge corresponent al mètode que s'està redefinint, però **amb el comportament de la classe pare**.
- Per tant, permet la **REUTILITZACIÓ** del mètode del que es va anul·lar la seu **transmissió** a la classe derivada

this vs super

- **this**: referència al suposat objecte que rep el missatge corresponent al mètode que s'està codificant.
- **super**: referència al suposat objecte que rep el missatge corresponent al mètode que es està codificant **però que es comporta com la classe pare i no com la classe filla**.

Especialització dels constructors

- La classe filla ha d'inicialitzar els seus atributs, però té el problema de que no té accés directe a ells si són de visibilitat **private**
- Si són de visibilitat **protected**, es **pot tendir a repetir el codi que ja es realitza a la classe pare.**
- Precisament, l'herència pretén **reutilitzar codi** i per tant que es puga invocar als constructors de la classe pare des de la classe filla.

Reutilització de constructors de classes pare

```
public class <NomClasse> extends <NomClasseBase> {  
    <visibilitat> <NomClasse> (<paràmetres>){  
        super(<arguments>);  
        ...  
    }  
}
```

- Es pot ometre per al cas del constructor de la classe pare amb una llista buida de paràmetres

- La invocació al constructor de la classe pare, ha de ser **la primera sentència** del constructor.
- Els arguments han de coincidir en quantitat i tipus amb el constructor de la classe padre.

Què imprimeix el següent codi?

```
class A {  
    public A() {  
        System.out.println(  
            "A's no-arg constructor is invoked");  
    }  
}  
  
class B extends A {  
}  
  
public class C {  
    public static void main(String[] args) {  
        B b = new B();  
    }  
}
```

Quin problema hi ha al codi?

```
class A {  
    public A(int x) {  
    }  
}  
  
class B extends A {  
    public B() {  
    }  
}  
  
public class C {  
    public static void main(String[] args) {  
        B b = new B();  
    }  
}
```

Classe Persona v 2.0

- Es defineixen ara els seus atributs com a "protected"

```
protected String dni;  
protected String nom;  
protected int edat;
```

Classe Persona v2.0

- Constructors que tenia la classe Persona (cap canvi):

```
public Persona() {}

public Persona(String dni, String nom, int edat) {
    this.dni = dni;
    this.nom = nom;
    if (edat >= 0 && edat < 150)
        this.edat = edat;
}

public Persona(Persona p) {
    this(p.dni, p.nom, p.edat);
}
```

Classe Persona v 2.0

- Els mètodes getters, setters i els mètodes visualitzar i clonar queden igual que a la versió anterior.
- Afegim el mètode `toString`:

```
public String toString() {  
    return this.dni + " - " + this.nom + " - " + this.edat;  
}
```

Classe Alumne

- Afegirem una classe Alumne, on afegir també el nivell d'estudis que cursa actualment (eixe serà el seu atribut específic).

Classe Alumne

- La definició de la classe seria així:

```
public class Alumne extends Persona{  
    // Aquí es definirán els membres de la classe  
}
```

Atributs de la classe Alumne

- S'ha determinat que el nivell de l'alumne s'emmagatzemarà en una variable de tipus "char". Els valors possibles seran B (batxillerat), M (grau mitjà), S(grau superior) i ? (desconegut).
- L'únic atribut seria:

```
private char nivell
```

Constructors de la classe Alumne

```
public Alumne() {  
    this.nivell = '?';  
}  
  
public Alumne(char nivell, String dni, String nom, int edat) {  
    super(dni, nom, edat);  
    this.nivell = validarNivell(nivell);  
}  
  
public Alumne(char nivell, Persona p){  
    super(p);  
    this.nivell = validarNivell(nivell);  
}
```

```
public Alumne(Persona p){  
    this('?', p);  
}  
  
public Alumne(String dni, String nom, int edat) {  
    this('?', dni, nom, edat);  
}  
  
public Alumne(Alumne a){  
    this(a.nivell, a.dni, a.nom, a.edat);  
}
```

Mètodes de la classe Alumne

```
public void visualitzar() {
    super.visualitzar();
    System.out.println("Nivell: " + this.nivell);
}

private static char validarNivell(char nivell) {
    if (nivell == 'B' || nivell == 'M' || nivell == 'S') {
        return nivell;
    } else {
        return '?';
    }
}
```

Altres mètodes d'Alumne

- Implementem també el mètode “`toString`” per a l’Alumne, que entre altres coses invocarà el mètode `toString` de “`Persona`” per a retornar la següent informació:

dni - nom - edat - nivell

Pràctica 2

- Realitza els canvis indicats sobre la classe Persona i crea la classe Alumne.
- Des d'un mètode main, crea objectes Persona i objectes Alumne.
- Invoca els mètodes `visualitzar` de cada objecte i comprova la informació obtinguda per consola.

Herència d'*Object* en Java

- Independentment de la utilització de la clàusula "extends" **QUALSEVOL CLASSE** definida, heretarà directa o indirectament de la classe Object.
- Object és la superclasse més general de qualsevol classe Java.
- Alguns mètodes d'Object són:

```
public boolean equals(Object object)  
protected Object clone()  
public String toString()  
protected void finalize()  
public int hashCode()
```

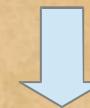
Sobreescritura del mètode equals de Object

- Recordem que quan es treballa amb objectes, utilitzem referències.

```
public static void main (String args[]) {  
    String dni1 = "00000000";  
    String dni2 = "00000000";  
    String dni3 = new String("00000000");  
    char t[] = {'0','0','0','0','0','0','0','0'};  
    String dni4 = new String(t);  
    System.out.println("dni1 : " + dni1);  
    System.out.println("dni2 : " + dni2);  
    System.out.println("dni3 : " + dni3);  
    System.out.println("dni4 : " + dni4);  
    System.out.println("dni1 == dni2 : " + (dni1 == dni2));  
    System.out.println("dni1 == dni3 : " + (dni1 == dni3));  
    System.out.println("dni1 == dni4 : " + (dni1 == dni4));  
    System.out.println("dni3 == dni4 : " + (dni3 == dni4));  
}
```

Sobreescritura del mètode equals de Object

```
dnil : 00000000  
dni2 : 00000000  
dni3 : 00000000  
dni4 : 00000000  
dnil == dni2 : true  
dnil == dni3 : false  
dnil == dni4 : false  
dni3 == dni4 : false
```



```
public boolean equals (Object obj)
```

Sobreescritura d'equals per a Persona

```
public boolean equals (Object obj) {  
    if (obj == this) return true;  
    if (obj == null) return false;  
    if (obj.getClass() != this.getClass()) return false;  
    return dni.equals(((Persona)obj).dni);  
}
```

Sobreescritura d'equals per a Persona

```
public boolean equals (Object obj) {  
    if (obj == this) return true;  
    if (obj == null) return false;  
    if (obj instanceof Persona) return dni.equals(((Persona)obj).dni);  
    return false;  
}
```

```
public      boolean equals (Persona obj) {  
    if (obj == this) return true;  
    if (obj == null) return false;  
    return dni.equals(obj.dni);  
}
```

Quins errors hi ha en el codi?

```
1 public class Circle {  
2     private double radius;  
3  
4     public Circle(double radius) {  
5         radius = radius;  
6     }  
7  
8     public double getRadius() {  
9         return radius;  
10    }  
11  
12    public double getArea() {  
13        return radius * radius * Math.PI;  
14    }  
15 }
```

```
17    class B extends Circle {  
18        private double length;  
19  
20        B(double radius, double length) {  
21            Circle(radius);  
22            length = length;  
23        }  
24  
25        @Override  
26        public double getArea() {  
27            return getArea() * length;  
28        }  
29    }
```

Preguntes

- Quina diferència hi ha entre la sobrecàrrega i la sobreescritura d'un mètode?
- Si un mètode en una classe derivada té el mateix nom que un mètode definit en la seu classe pare i els dos tenen establert el mateix valor de retorn, és una sobreescritura o una sobrecàrrega?
- Si un mètode en una classe derivada té el mateix nom que un mètode definit en la seu classe pare i tenen establert diferent valor de retorn, suposaria un problema?

Preguntes

- Si un mètode en una classe derivada té el mateix nom que un mètode definit en la seu classe pare, i els dos tenen un número de paràmetres diferent, seria una sobrecàrrega o una sobreescritura?

Resum de modificadors de visibilitat

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No