

GRAU D'ENGINYERIA INFORMÀTICA

PROGRAMACIÓ II

Bloc 2:

Programació Orientada a Objectes (3)

Laura Igual

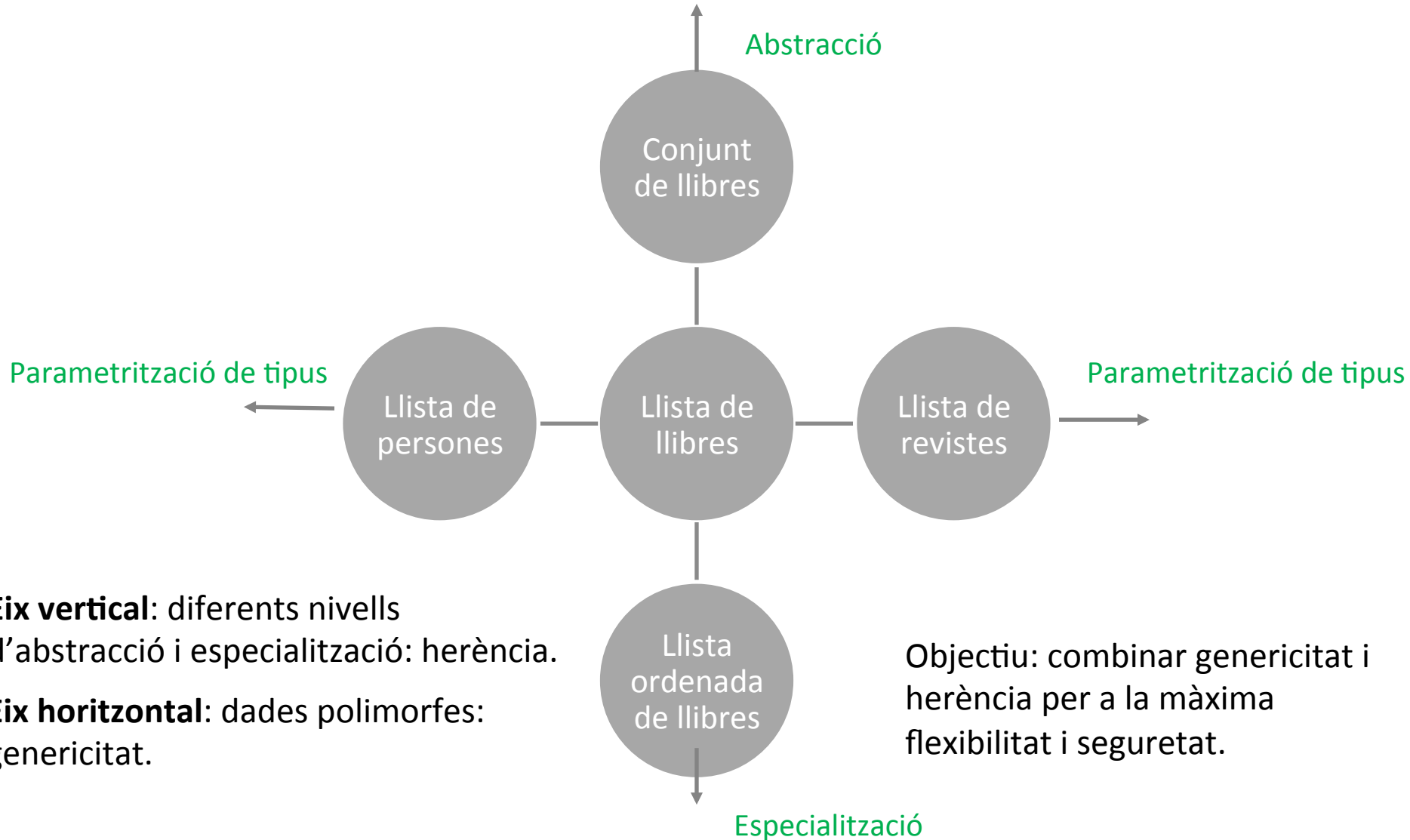
Departament de Matemàtica Aplicada i Anàlisi

Facultat de Matemàtiques

Universitat de Barcelona

HERÈNCIA I JERARQUIA DE CLASSES

Generalització



Herència

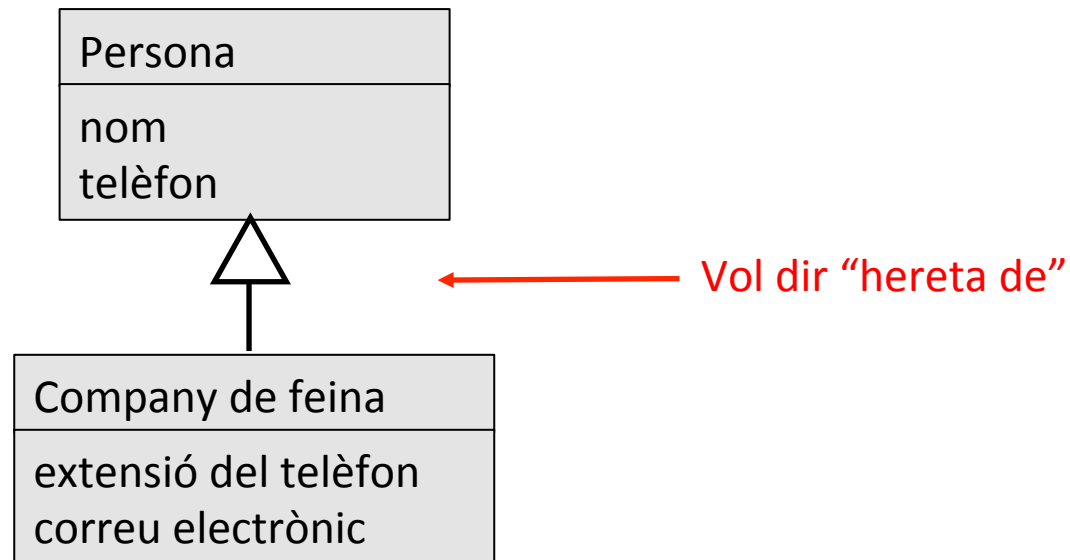
- L'herència és un mecanisme que permet definir una classe nova a partir d'una d'anterior descrivint les diferències entre elles.
- Característica pròpia de la programació orientada a objectes.
- Facilita la reutilització
- Concepte de relacions de generalització i especialització

Tipologies d'herència

- Depenent de la manera d'arribar-hi a l'herència, s'anomenen:
 - Herència per especialització
 - Herència per generalització

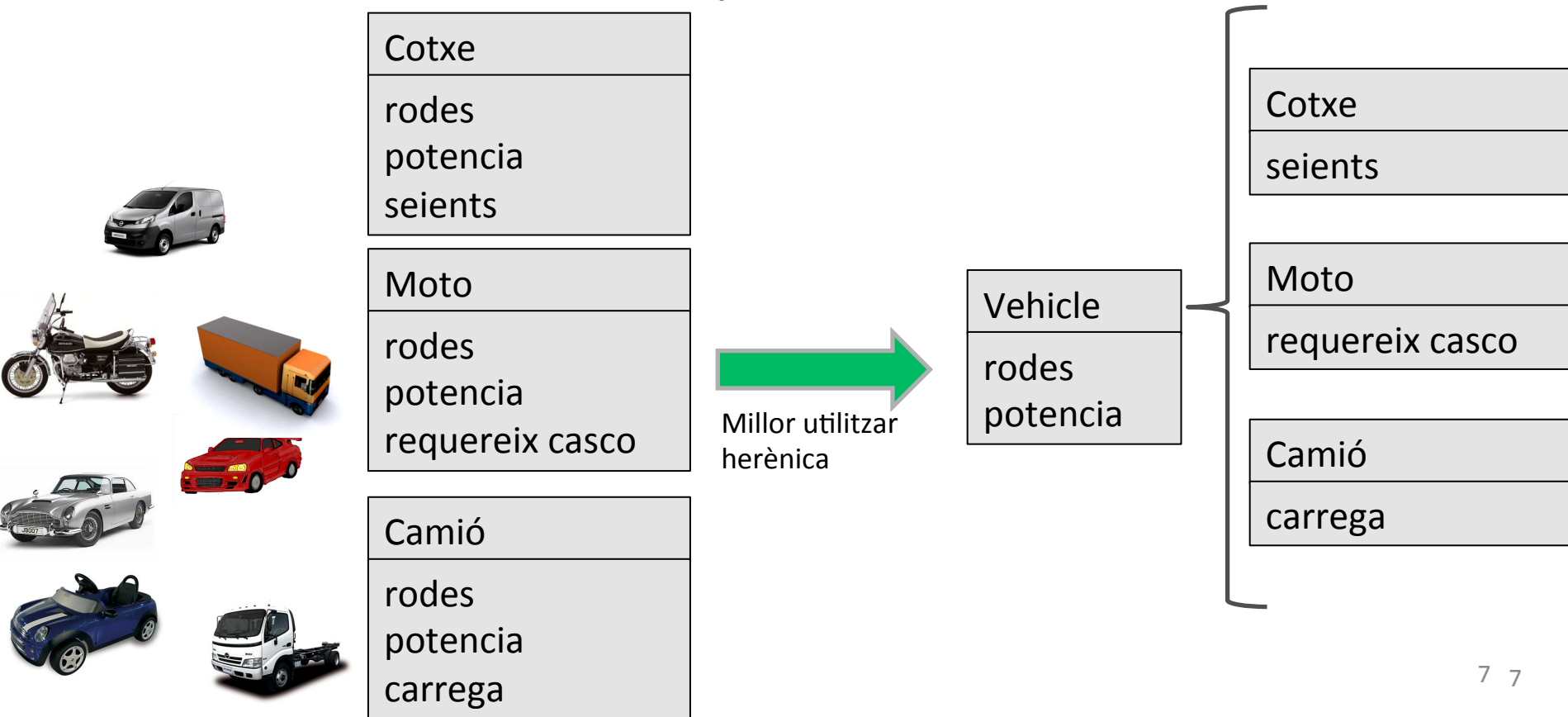
Herència per especialització

- Sorgeix de la necessitat de crear una classe nova que afegeixi unes propietats i un comportament a una altra classe del domini ja existent.
- Quan afegim funcionalitat a un disseny ja donat.
- Exemple:



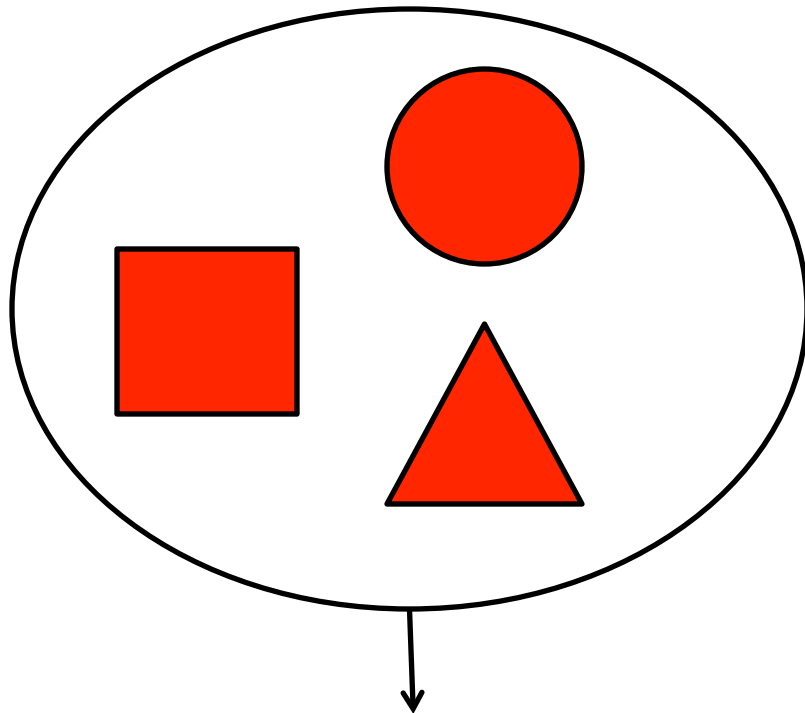
Herència per generalització

- Apareix amb la finalitat d'homogeneïtzar el comportament de les parts comunes a certes classes.
- Quan es crea el disseny de classes.



Exemple

- Herència → Jerarquia de classes
- Classe abstracta: Figura geomètrica
- Classe: Triangle, quadrat, cercle, ...



Figures geomètriques → Dibuixar

Figura geomètrica
color posició a pantalla àrea perímetre
calcula àrea calcula perímetre retorna color assigna color

Quadrat
color posició a pantalla àrea perímetre dimensió costats

Circumferència
color posició a pantalla àrea perímetre Radi

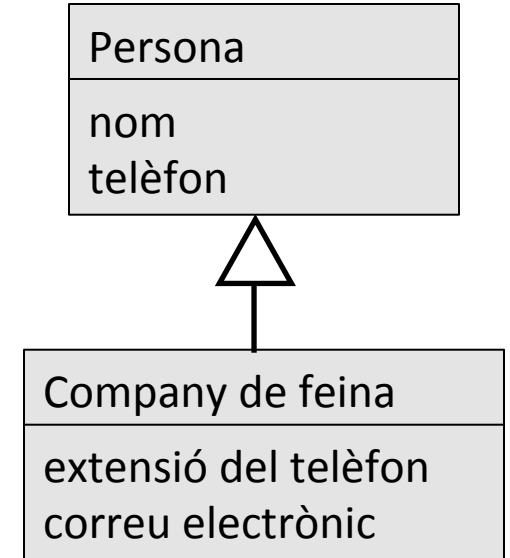
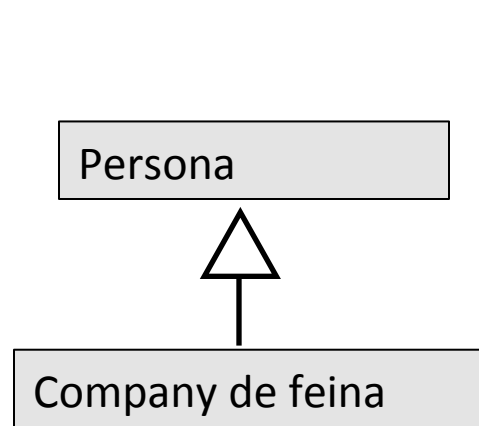
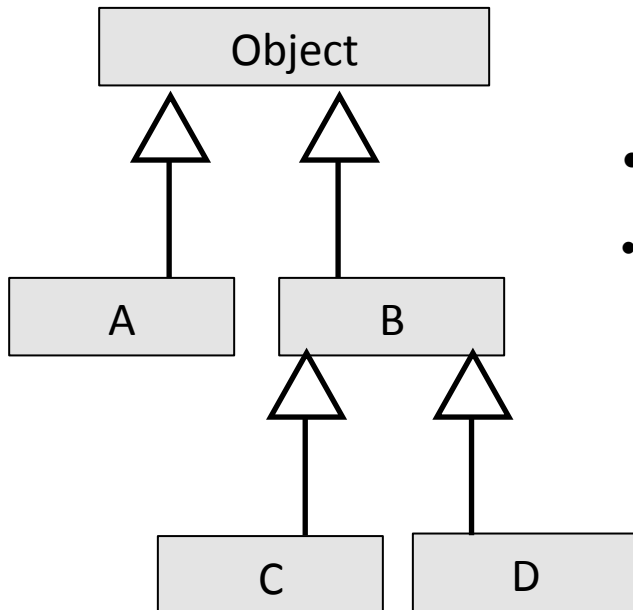
Triangle
color posició a pantalla àrea perímetre dimensió 3costats

Herència

- Terminologia:



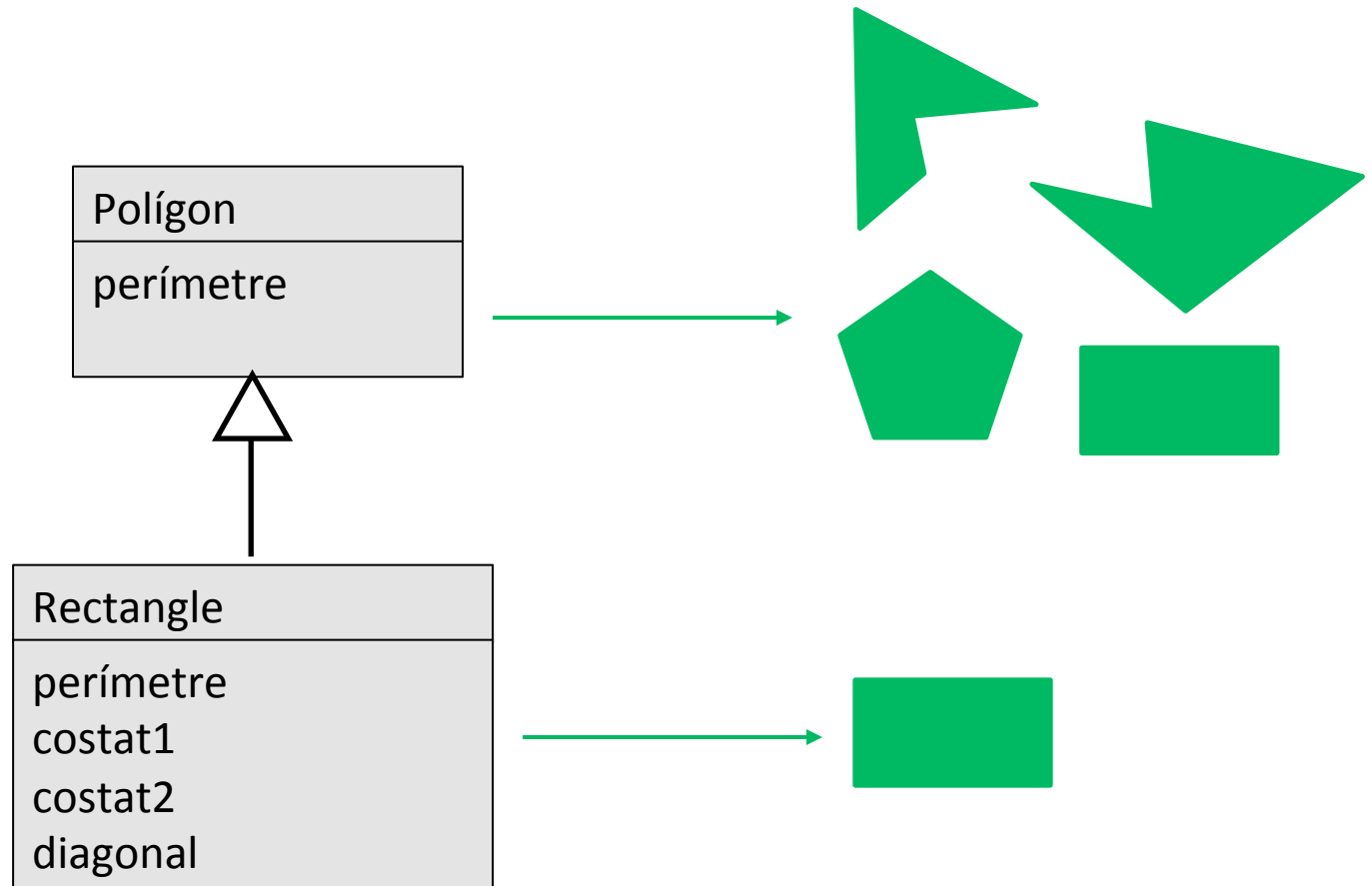
- Java:



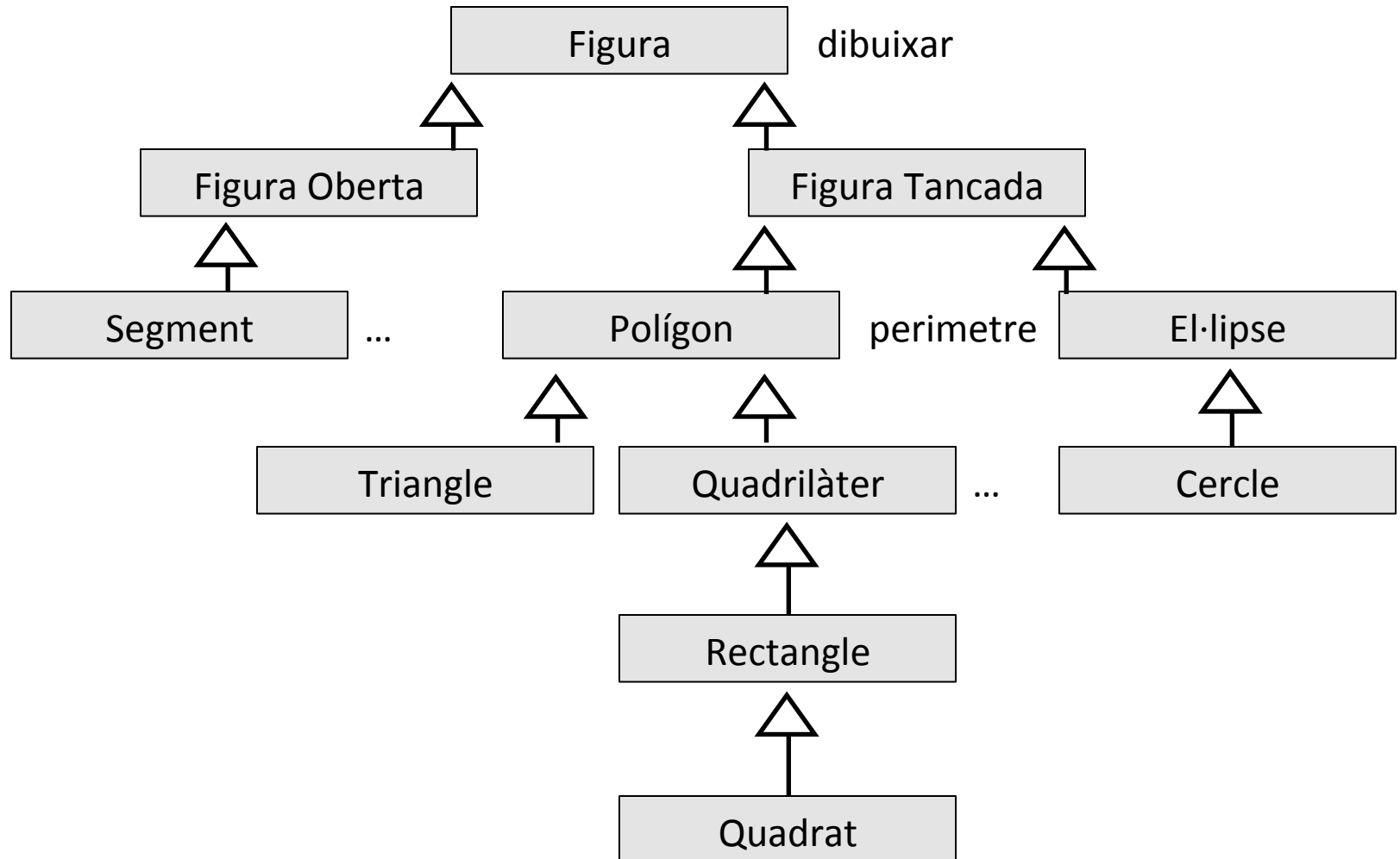
- Object** és la classe arrel (paquet `java.lang`)
- Object** descriu les propietats comunes a tots els objectes
 - C i D són **subclasses** de B
 - B és la **superclasse** de C i D

Herència

- Exemple



Jerarquia de classes



Consideracions sobre l'herència

- Els atributs i mètodes de la superclasse estaran **sempre definits en la subclasse**.
 - Aquesta restricció només s'aplica als atributs i mètodes definits amb la **visibilitat public o protected**.
 - Les classes filles no tenen accés als atributs i mètodes definits com a **private**.

Consideracions sobre l'herència

Atributs:

- En una classe filla, podem afegir **nous atributs**.
- S'ha de vigilar a l'hora de **triar els noms** dels nous atributs.

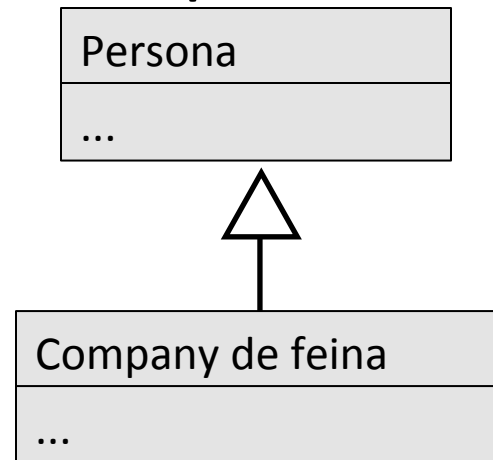
Consideracions sobre l'herència

Mètodes:

- Podem realitzar 3 tasques diferents:
 - Afegir mètodes nous
 - Implementar mètodes declarats prèviament com abstractes
 - Tornar a implementar mètodes.

Afegir mètodes nous

- Atributs nous → necessitem mètodes nous
 - Mètodes que realitzin tasques específiques de la classe filla.
- Els mètodes definits en la subclasse es consideraran mètodes d'aquesta i només s'hi podrà accedir des d'instàncies d'aquesta o de les seves classes filla.



Tipus de classes

- **Abstract**
- **Final**
- **Public**
- **Synchronizable**

Tipus de classes

- **Classe abstracta:**

No s'instancia, sinó que s'utilitza com classe base per a l'herència.

- Exemple:

Classificació animal:

- Mamífer,
- Bípede,
- Quadrúpede,

→ D'aquests conjunts no hi ha instàncies concretes

La balena és un mamífer, però de la subespècie dels cetacis

El cavall és un quadrúpede, de la subespècie dels equins

Tipus de classes

- **Classe final**

Se declara com la classe que termina una cadena d'herència. No es pot heretar d'ella.

- Exemple:

La classe **Math** és una classe final.

Tipus de classes

- **Classes public**

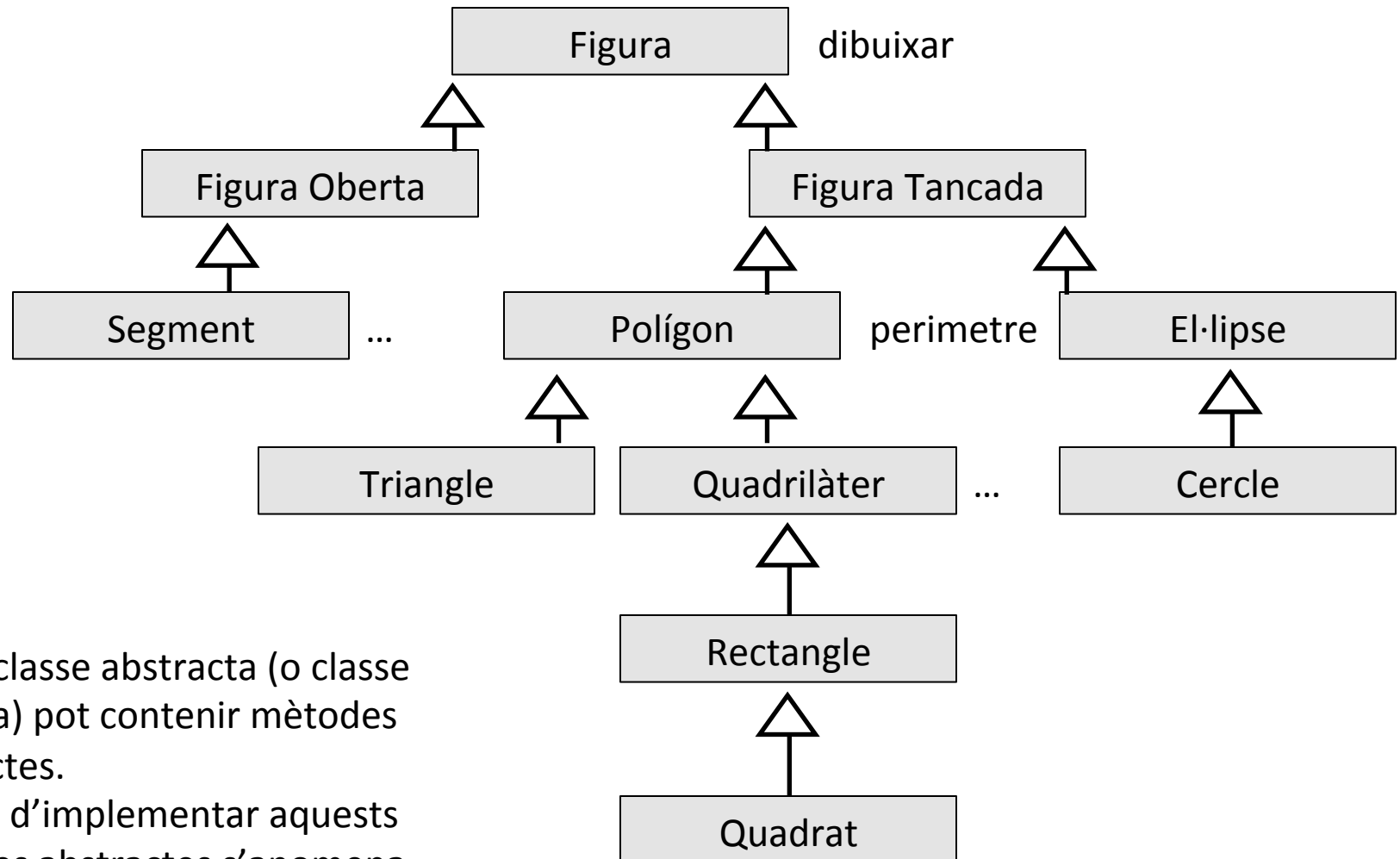
Són accessibles des d'altres classes, o directament o per herència.

- Són accessibles dins del mateix paquet en el que s'han declarat.
- Per a accedir des d'altres paquets, primer tenen que ser importades.

Tipus de classes

- **Classe synchronizable**
- Aquest modificador especifica que tots els mètodes definits en la classe són sincronitzats, es a dir, que no es poden accedir al mateix temps a ells des de diferents threads;
- El sistema s'encarrega de col·locar els flags necessaris per a evitar-ho.
- Aquest mecanisme fa que des de threads diferents es puguin modificar les mateixes variables sense que hagi problemes de sobreescritura.

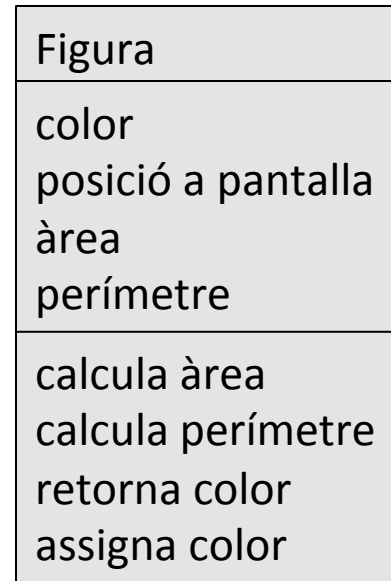
Exemple de jerarquia de classes



- Una classe abstracta (o classe diferida) pot contenir mètodes abstractes.
- El fet d'implementar aquests mètodes abstractes s'anomena **fer efectiu** un mètode.

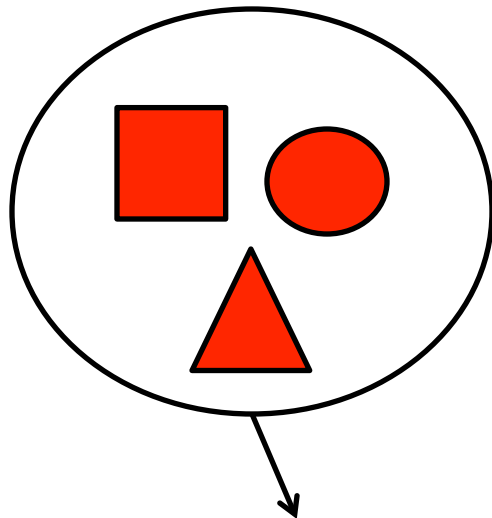
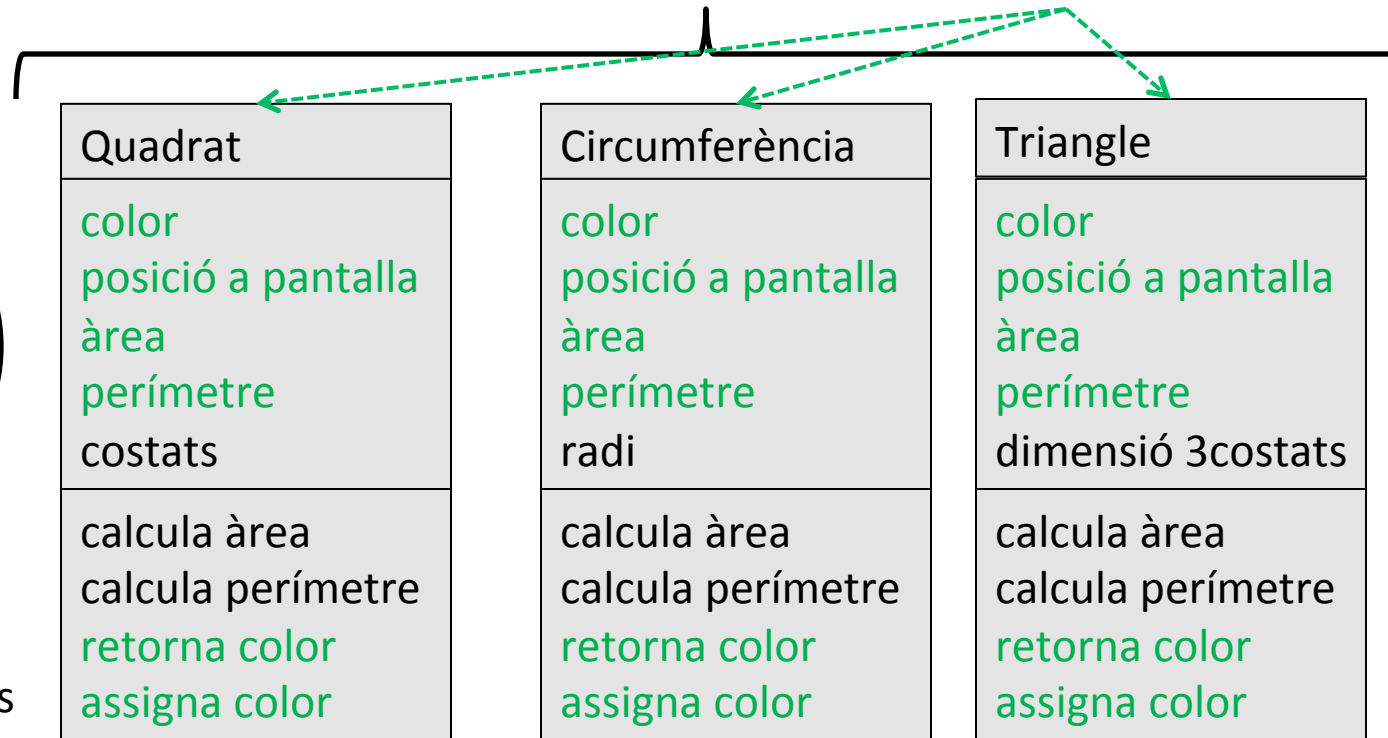
Exemple

- **Classe abstracta:** Figura geomètrica
- Classe: Triangle, quadrat, cercle, ...



← No pot haver-hi una instància d'una classe abstract

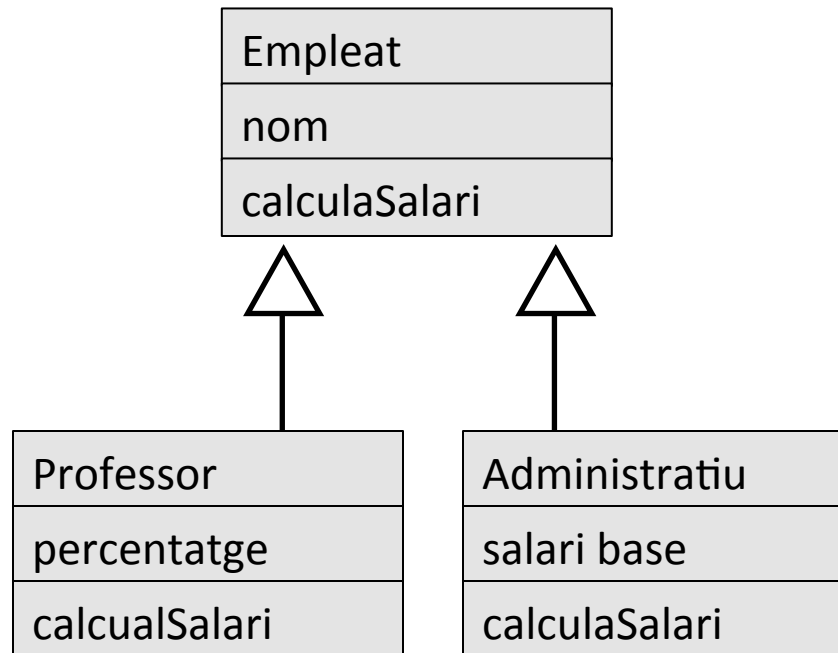
Atributs i mètodes heretats



Figures geomètriques

Implementar mètodes abstractes

- **Mètode abstracte** és aquell que té definida la seva interfície (nom, tipus, nombre de paràmetres i valor de retorn), però no té implementat el codi que atindrà les peticions.
- Exemple:



Implementar mètodes abstractes

- Si una classe té declarat com a mínim un mètode abstracte, es diu que la **classe és abstracta**.
- Les classes abstractes obliguen les classes que hereten d'aquesta a implementar els mètodes no implementats.

En cas que una classe filla continuï sense implementar un mètode abstracte, aquesta ha de ser també abstracta.

Sobreescritura de mètodes

- Ens permet modificar el comportament d'un mètode definit prèviament en la classe mare per que realitzi altres tasques.
- Cal tornar a definir-lo i implementar-lo amb una **signatura igual o diferent**.

Amb Java

- Per definir una herència:
paraula reservada ***extends***
+ nom de la classe de la qual s'hereta
- Per accedir als mètodes definits a la classe mare:
paraula reservada ***super***

Ús d'herència

```
public class MiClase {  
    int i;  
    public MiClase() {  
        i = 10;  
    }  
    public void suma_a_i( int j ) {  
        i = i + j;  
    }  
}
```

```
import MiClase;  
public class MiNuevaClase extends MiClase {  
    public void suma_a_i( int j ) {  
        i = i + ( j/2 );  
        super.suma_a_i( j );  
    }  
}
```

sobreescriptura

Fa referència al mètode
de la classe mare

Ús d'herència

```
public static void main(String[] args) {  
    MiNuevaClase mnc;  
    mnc = new MiNuevaClase();  
    mnc.suma_a_i( 10 );  
  
    System.out.println(mnc.i);  
}
```

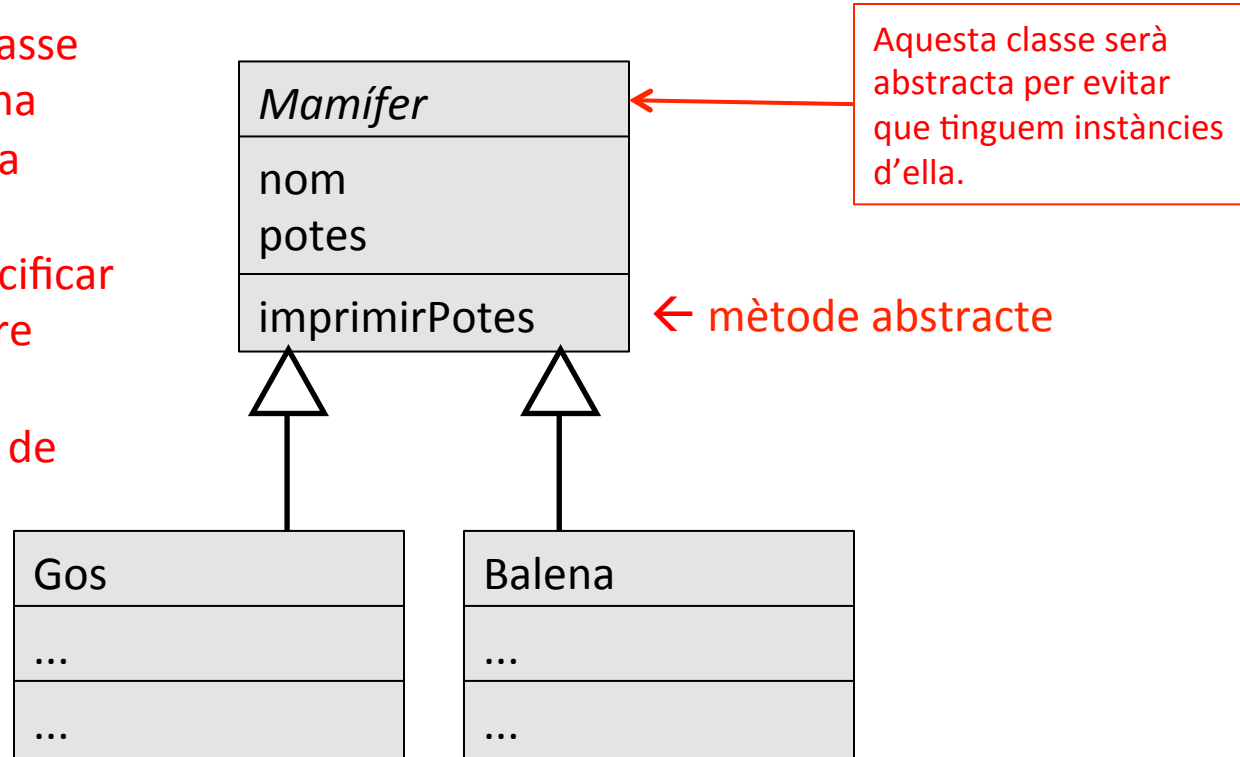


Resultat: 25

Exemple: Classe abstracta

Mamífer serà una classe abstracta, ja que hi ha informació d'aquesta classe que no es pot conèixer sense especificar més (saber més sobre l'animal).

Exemple: el número de potes de l'animal.



```

public abstract class Mamifer {
    private String nom;
    private int potes;
    public void imprimirPotes() {
        System.out.println(nom + " té " + potes + " potes\n");
    }
    public Mamifer(String nom, int potes) {
        this.nom = nom;
        this.potes = potes;
    }
}

public class Gos extends Mamifer {
    public Gos(String nom){
        super(nom, 4);
    }
}

public class Balena extends Mamifer {
    public Balena(String nom){
        super(nom, 0);
    }
}

public class CreaGos {
    public static void main(String [] args) {
        Gos bobí = new Gos("Bobí");
        bobí.imprimirPotes(); /*Està a la classe mamífer*/
    }
}

```

Mamifer.java

Gos.java

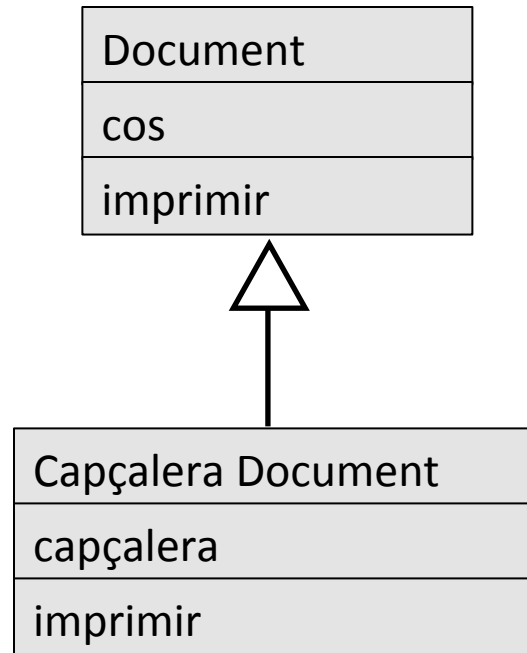
Balena.java

CreaGos.java

S'invoca a la versió del mètode de la primera superclasse que el continga

Exemple

- Herència amb sobreescritura de mètodes:

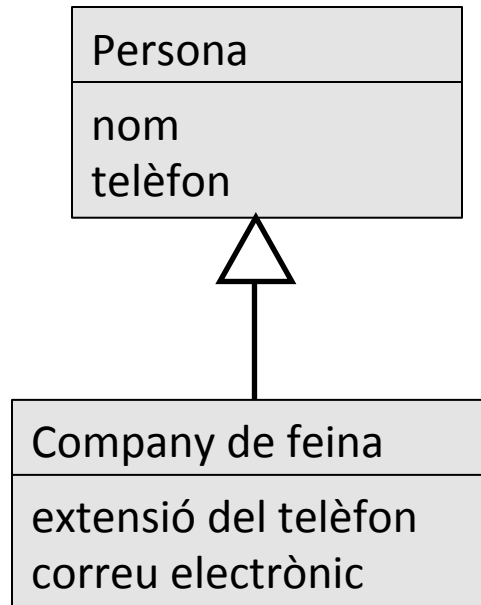


Entenem DocumentCapçalera com un tipus específic de document que té a més d'un cos de document una capçalera.

Les funcions a realitzar pel mètode imprimir ara seran diferents, ja que tenim una informació diferent emmagatzemada.

Exemple

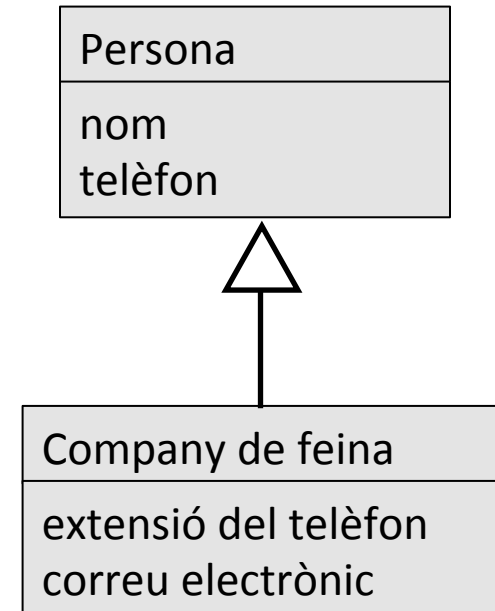
- Implementar l'exemple:



Solució:

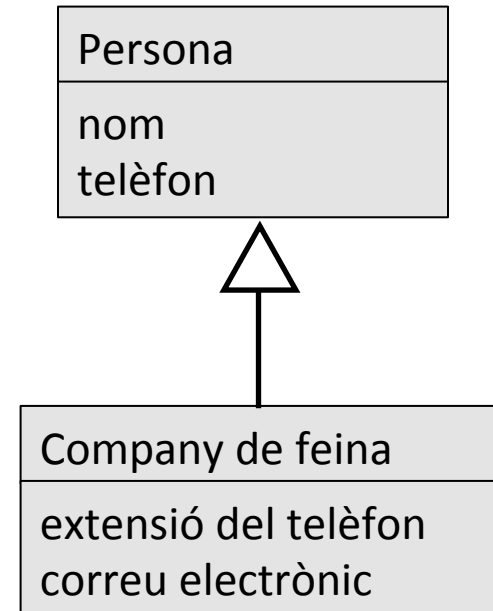
Persona.java

```
public class Persona {  
    private String nom;  
    private String telefon;  
    // constructor  
    public Persona (String pNom, String pTelefon) {  
        nom = pNom;  
        telefon = pTelefon;  
    }  
    // Getters i setters  
    public String getNom() {  
        return nom;  
    }  
    public String getTelefon() {  
        return telefon;  
    }  
    public void setNom(String pNom) {  
        nom = pNom;  
    }  
    public void setTelefon(String pTelefon) {  
        telefon = pTelefon; }  
}
```



Company.java

```
public class Company extends Persona {  
    private String extTel;  
    private String email;  
    // constructors:  
    public Company(String pNom, String pTelefon) {  
        super(pNom, pTelefon);  
        extTel = "";  
        email = "";  
    }  
    public Company(String pNom, String pTelefon, String pExtTel, String pEmail) {  
        super(pNom, pTelefon);  
        extTel = pExtTel;  
        email = pEmail;  
    }  
  
    // Getters i setters  
    public String getExtTel() {  
        return extTel;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setExtTel(String pExtTel) {  
        extTel = pExtTel;  
    }  
  
    public void setEmail(String pEmail) {  
        email = pEmail;  
    }  
}
```



Exercici:

Donat el codi anterior de les classe Persona i Company indicar si hi ha errors de compilació en les següents classes del mateix paquet:

1. Classe TestCompanys1

```
public class TestCompanys1 {  
    public static void main(String[] args){  
        Company nouCompany = new Company();  
    }  
}
```

2. Classe TestCompanys2

```
public class TestCompanys2 {  
    public static void main(String[] args){  
        String nom="Joan";  
        String telefon="931111111";  
        String telefonActual;  
        Company nouCompany = new Company(nom, telefon);  
        System.out.println(nouCompany.getNom());  
        telefonActual = "93222222";  
        nouCompany.setTelefon(telefonActual);  
    }  
}
```

Solució Exercici:

Donat el codi de les classe Persona i Company indicar si hi ha errors de compilació en les següents classes del mateix paquet:

1. Classe TestCompanys1

```
public class TestCompanys1 {  
    public static void main(String[] args){  
        Company nouCompany = new Company();  
    }  
}
```

← Error de compilació:
La classe Company no té
constructor sense
paràmetres.

2. Classe TestCompanys2

```
public class TestCompanys2 {  
    public static void main(String[] args){  
        String nom="Joan";  
        String telefon="931111111";  
        String telefonActual;  
        Company nouCompany = new Company(nom, telefon);  
        System.out.println(nouCompany.getNom());  
        telefonActual = "93222222";  
        nouCompany.setTelefon(telefonActual);  
    }  
}
```

No. Encara que la classe
Company no té els
mètodes getNom i
setTelefon implementats,
la superclasse Persona si
que els té.

Donarà error?

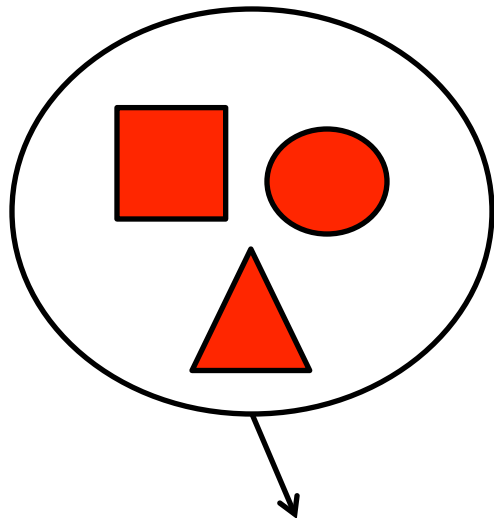
Donarà error?

Exercici

- **Classe abstracta:** Figura geomètrica
- Classe: Triangle, quadrat, cercle, ...

Figura
color posició a pantalla àrea perímetre
calcula àrea calcula perímetre retorna color assigna color

Atributs i mètodes
heretats



Figures geomètriques

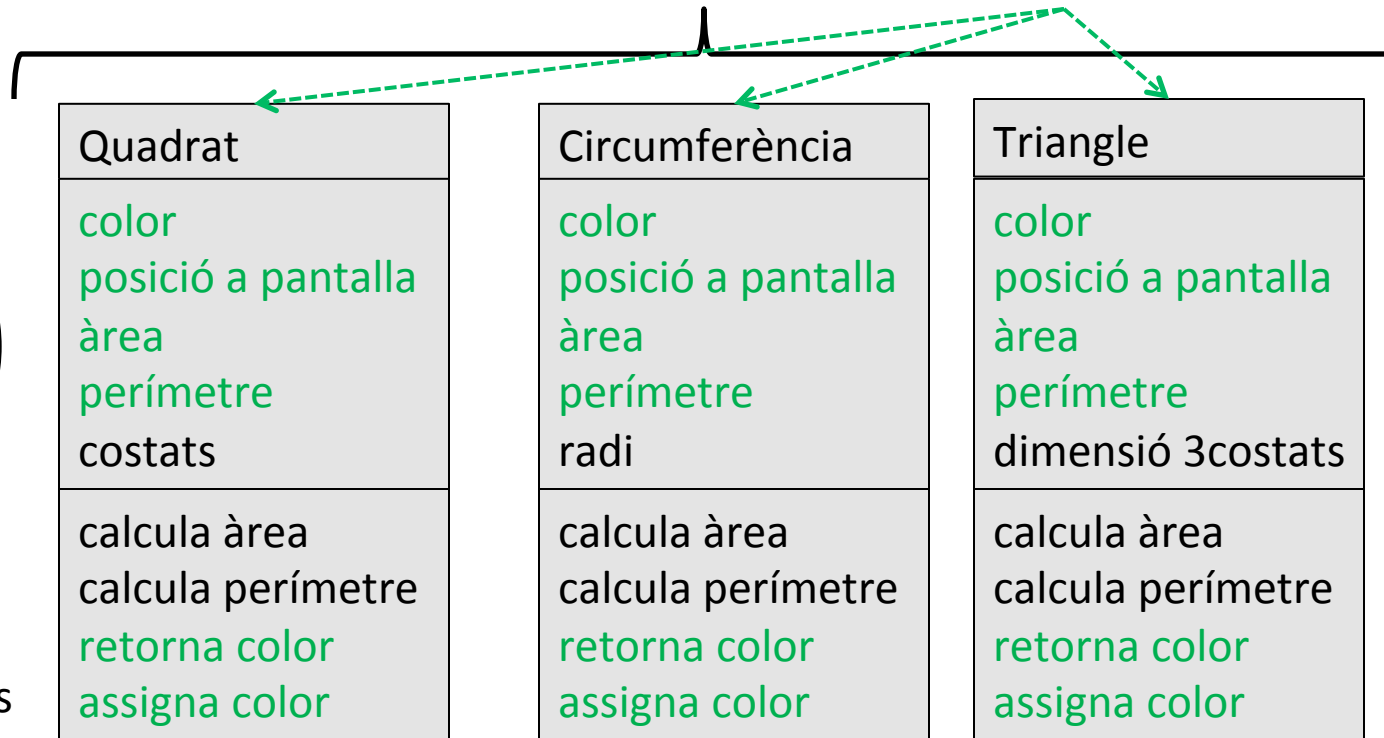


Figura.java

```
public abstract class Figura {  
    protected String color;  
    protected double x, y;  
    protected double area;  
    protected double perimetre;
```

```
    // Mètodes abstractes:  
    public abstract double calculaArea();  
    public abstract double calculaPerimetre();
```

```
    //Retorna el Color  
    public String getColor(){  
        return color;  
    }  
    //Assigna el Color  
    public void setaColor(String color){  
        this.color=color;  
    }
```

```
    //Retorna la posició de la Figura  
    public double [] getPosicion(){  
        double [] posicioxy = {x, y};  
        return posicioxy;  
    }  
    //Assigna la posició de la Figura  
    public void setPosicio(double[] posicioxy){  
        x=posicioxy[1];  
        y=posicioxy[2];  
    }  
} // Final de la classe Figura
```

Quadrat.java

```
public class Quadrat extends Figura {  
    private double costat; // longitud dels costats  
    // constructors  
    public Quadrat() {  
        costat=0.0;  
    }  
    public Quadrat(double costat) {  
        this.costat = costat;  
    }  
    // Calcula l'àrea del quadrat:  
    public double calculaArea() {  
        area = costat * costat ;  
        return area;  
    }  
    // Calcula el valor del perímetre:  
    public double calculaPerímetre(){  
        perímetre = 4 * costat;  
        return perímetre;  
    }  
}
```

Cercle.java

```
public class Cercle extends Figura {
    public static final double PI=3.14159265358979323846;
    public double  radi;
    // constructors
    public Cercle(double x, double y, double radi) { crearCercle(x,y,radi); }
    public Cercle (double radi) { crearCercle(0.0,0.0,radi); }
    public Cercle (Cercle c){ crearCercle(c.x,c.y,c.radi); }
    public Cercle() { crearCercle(0.0, 0.0, 1.0); }
    // Mètode de suport
    private void crearCercle(double x, double y, double radi) {
        this.x=x; this.y=y; this.radi =radi;
    }
    // calcula l'area del cercle
    public double calculaArea() {
        area = PI * radi * radi;
        return area;
    }
    // calcula el valor del perímetre
    public double calculaPerimetre() {
        perimetre = 2 * PI * radi;
        return perimetre;
    }
} // fi de la classe Cercle
```


Exercici

- Amplia la implementació de la classe **Cercle** que hereta de la classe abstracta **Figura** amb
 - Un contador de cercles,
 - Dos mètodes propis,
 - Un mètode d'objecte per comparar cercles i
 - Un mètode de classe per comparar cercles.

```
public class Cercle extends Figura {
```

```
    static int numCercles = 0;
```

```
    public static final double PI=3.14159265358979323846;
```

```
    public double radi;
```

```
    // constructors
```

```
    public Cercle(double x, double y, double radi) {
```

```
        this.x=x; this.y=y; this.radi=radi;
```

```
        numCercles++;}
```

```
    public Cercle(double radi) { this(0.0, 0.0, radi); }
```

```
    public Cercle(Cercle c) { this(c.x, c.y, c.radi); }
```

```
    public Cercle() { this(0.0, 0.0, 1.0); }
```

```
    // calcula l'area del cercle
```

```
    public double calculaArea() {
```

```
        area = PI * radi * radi;
```

```
        return area;
```

```
    }
```

```
    // calcula el valor del perímetre
```

```
    public double calculaPerimetre(){
```

```
        perimetre = 2 * PI * radi;
```

```
        return perimetre;
```

```
    }
```

```
    // mètode d'objecte per a comparar cercles
```

```
    public Cercle elMajor(Cercle c) {
```

```
        if (this.radi>=c.radi)
```

```
            return this;
```

```
        else return c;
```

```
    }
```

```
    // mètode de classe per a comparar cercles
```

```
    public static Cercle elMajor(Cercle c, Cercle d) {
```

```
        if (c.radi>=d.radi)
```

```
            return c;
```

```
        else return d;
```

```
    }
```

```
} // fi de la classe Cercle
```

```
public class TestCercles {  
    public static void main(String[] args){  
        Cercle cercleGran;  
        System.out.println("número de cercles = " + Cercle.numCercles);  
        Cercle cercle1 = new Cercle(1.5);  
        System.out.println("número de cercles = " + Cercle.numCercles);  
        Cercle cercle2 = new Cercle(2.5);  
        System.out.println("número de cercles = " + Cercle.numCercles);  
        Cercle cercle3 = new Cercle(3.5);  
        System.out.println("número de cercles = " + Cercle.numCercles);  
  
        cercleGran = Cercle.elMajor(cercle1, cercle2);  
        System.out.println("El radi del cercle gran és = " + cercleGran.getRadi());  
  
        cercleGran = cercle3.elMajor(cercle2);  
        System.out.println("El radi del cercle gran és = " + cercleGran.getRadi());  
    }  
} // fi de la classe
```

Sortida per pantalla →

```
número de cercles = 0  
número de cercles = 1  
número de cercles = 2  
número de cercles = 3  
El radi del cercle gran és = 2.5  
El radi del cercle gran és = 3.5
```

```

public class Cercle extends Figura {
    // quantitat d'objectes d'aquesta classe que existeixen.
    static int numCercles = 0;
    // constant PI
    private static final double PI=3.14159265358979323846;
    // radi del cercle
    private double radi;

    // constructors
    public Cercle(double x, double y, double radi) {
        this.x=x; this.y=y; this.radi =radi;
        // actualitza la quantitat d'objectes d'aquesta classe
        sumarCercle();
    }
    public Cercle(double radi) { this(0.0, 0.0, radi); }
    public Cercle(Cercle c) { this(c.x, c.y, c.radi); }
    public Cercle() { this(0.0, 0.0, 1.0); }

    // calcula l'area del cercle
    public double calculaArea() {
        area = (double) (PI * radi * radi);
        return area; }

    // calcula el valor del perímetre
    public double calculaPerimetre(){
        perimetre = (double) (2 * PI * radi);
        return perimetre;
    }

    // mètode d'objecte per a comparar cercles
    public Cercle elMajor(Cercle c) {
        if (this.radi>=c.radi) return this;
        else return c; }

```

```

// mètode de classe per a comparar cercles
public static Cercle elMajor(Cercle c, Cercle d) {
    if (c.radi>=d.radi) return c; else return d;
}

public double getRadi(){
    return this.radi;
}

// destructor
protected void finalize() {
    // actualitza la quantitat d'objectes d'aquesta classe que existeixen:
    restarCercle();
}

// mètode de classe que incrementa en un la quantitat d'objectes d'aquesta
classe que existeixen.
private static void sumarCercle(){
    numCercles++;
}

// mètode de classe que decrementa la quantitat d'objectes creats dins
d'aquesta classe.
private static void restarCercle(){
    numCercles--;
}

} // fi de la classe Cercle

```

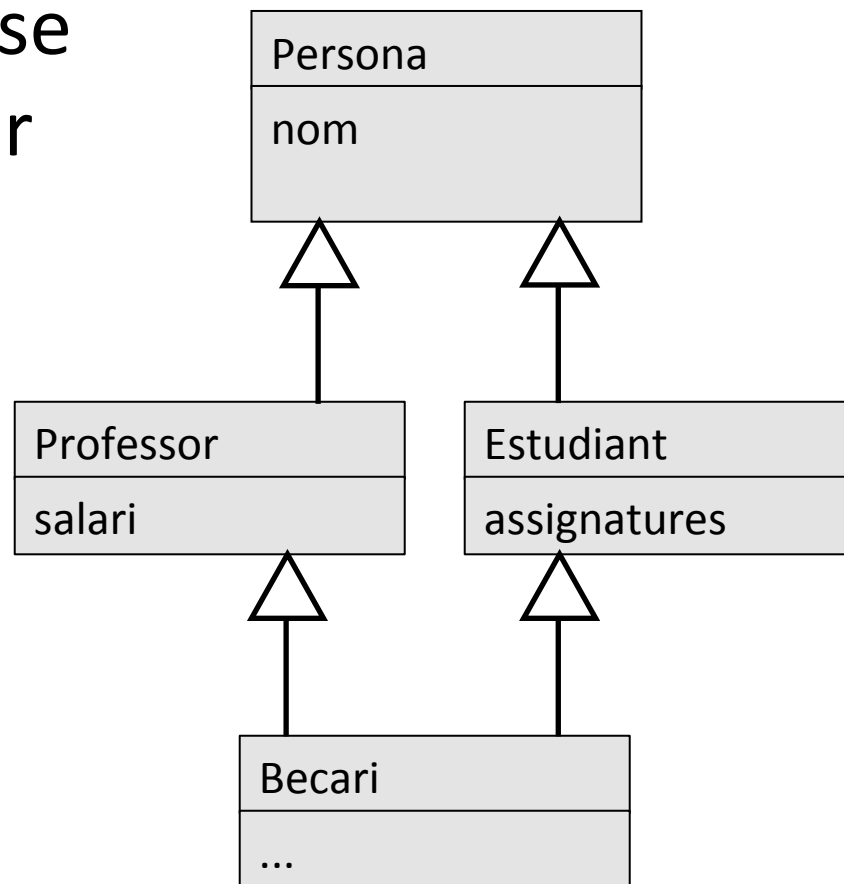
Una altra possible implementació de la classe Cercle.

Herència múltiple

- L'herència en què la classe nova és generada a partir de dues o més classes alhora.
- Exemple:

Quin pot ser el problema?

Problema: el becari té dos atributs nom



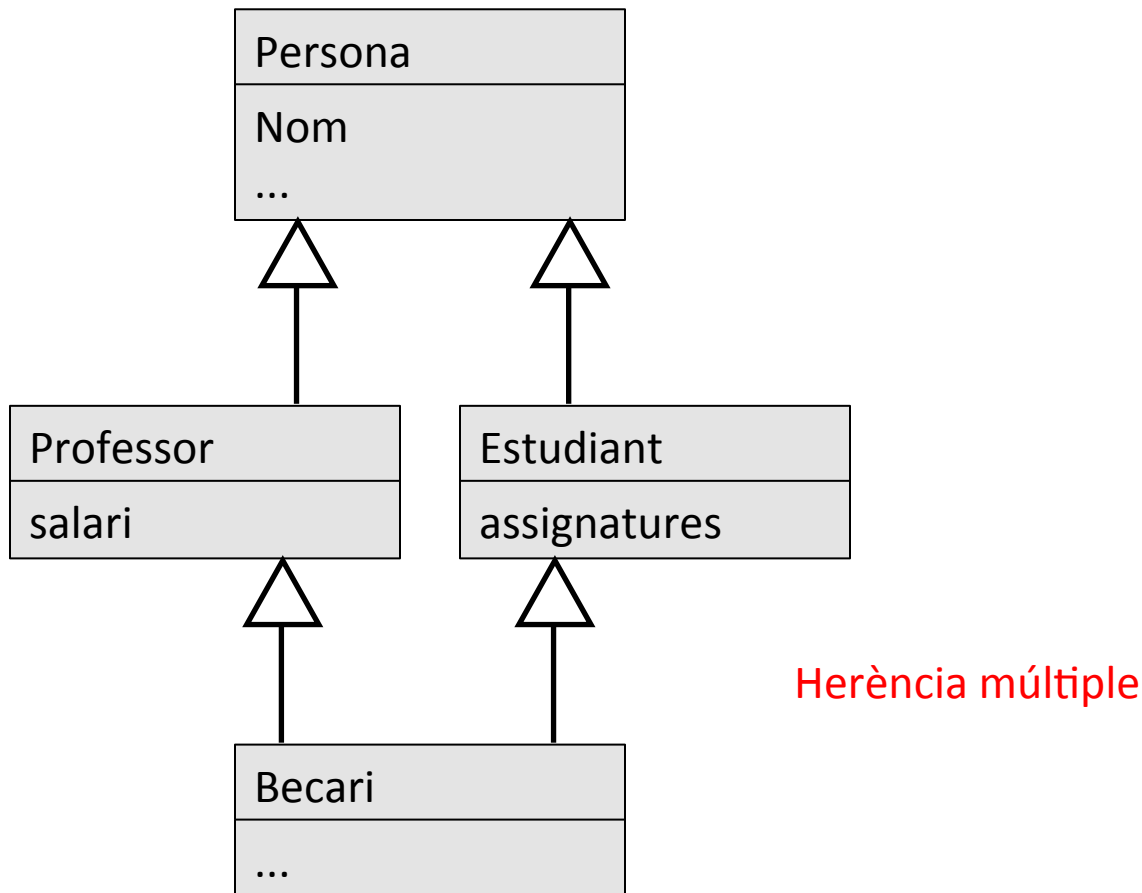
Herència múltiple:

Duplicitat d'atributs i mètodes

- Podem trobar que una classe té un atribut o mètode repetit perquè hereta de classes que contenen el mateix atribut o mètode.
- Calen mecanismes per a pal·liar aquesta problemàtica.

Duplicitat d'atributs i mètodes

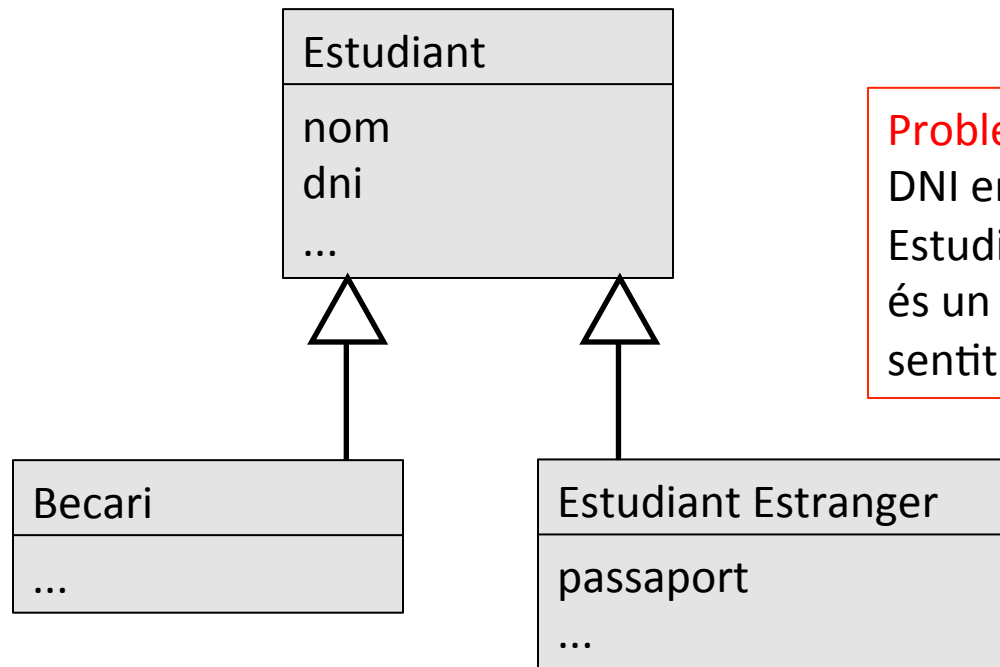
- Cas en que sempre es produiran duplicitats:



Errors típics de l'herència

1. Creació de superclasses poc generals
2. Ús de subclasses en comptes d'una superclasse

Creació de superclasses poc generals

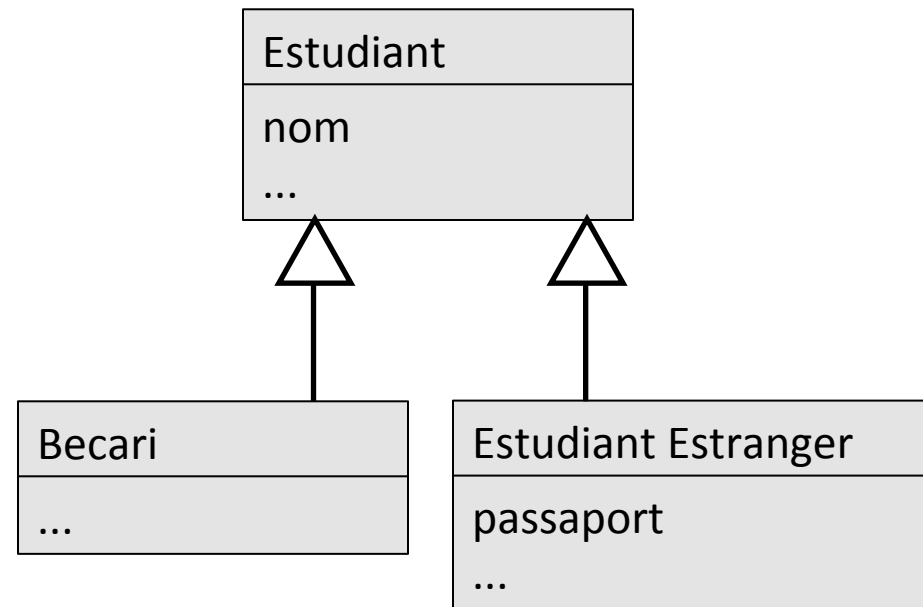


Problema: apareix el DNI en la classe Estudiant Estranger, que és un atribut que no té sentit.

Evitar-ho al disseny.

Ús de subclasses en comptes d'una superclasse

- Si volem emmagatzemar una relació d'estudiants, podem definir un vector per a emmagatzemar les instàncies de la classe Becari i Estudiant Estranger;
- En recuperar-los, com que poden estar barrejats, hem d'utilitzar la superclasse Estudiant, però si volem accedir a mètodes definits en la subclasse hem d'utilitzar el **càsting**.



Conversió de tipus

- El càsting (o conversió de tipus) ens permet utilitzar una instància d'una classe com si es tractés d'una instància d'un altre tipus.

Tot i que la definició anterior es completament certa, cal matitzar-la, ja que podrem realitzar el procés de càsting sempre que la conversió sigui possible.

Conversió de tipus

- **Conversió implícita:** (automàtica)

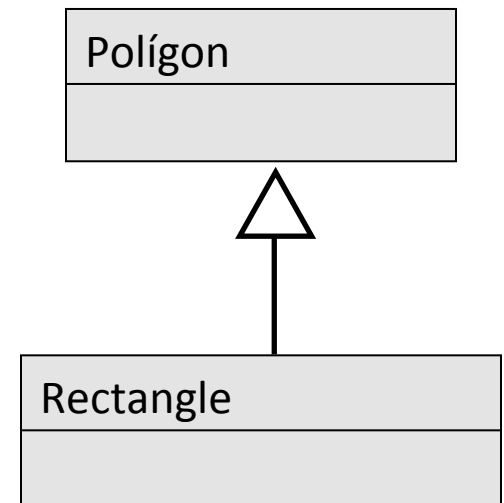
- *Tipus primitius* a un que suporti un rang major de valors

```
float saldo = 300;    //podem assignar-li un enter  
int codi = 3.7;       //Donarà ERROR
```

- *Referències*: tot objecte conté una instància de les seves superclasses

- ***cast-up***
 - sempre vàlid

```
Poligon poligon;  
Rectangle rectangle = new Rectangle();  
poligon = rectangle;
```



Conversió de tipus

- **Conversió explícita:**

- *Tipus primitius*: perden informació

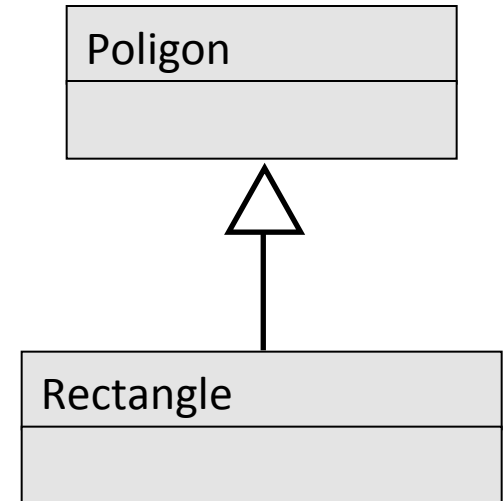
```
long l = 200;  
int i = (int)l;
```

- *Referències*: assignar a un objecte d'una subclasse un de la superclasse
 - ***cast-down*** o *narrowing*
 - No sempre vàlid
 - L'error es pot produir:
 - en temps d'execució (**ClassCastException**)
 - en temps de compilació si no és ni tan sols una subclasse.

Conversió explícita de referències

- Pot donar un error en execució:

```
Poligon [] poligons = new Poligon [30];  
...  
Rectangle r = (Rectangle)poligons[i];
```



- Donaria error en compilació:

```
Compte c = (Compte)poligons[i];
```

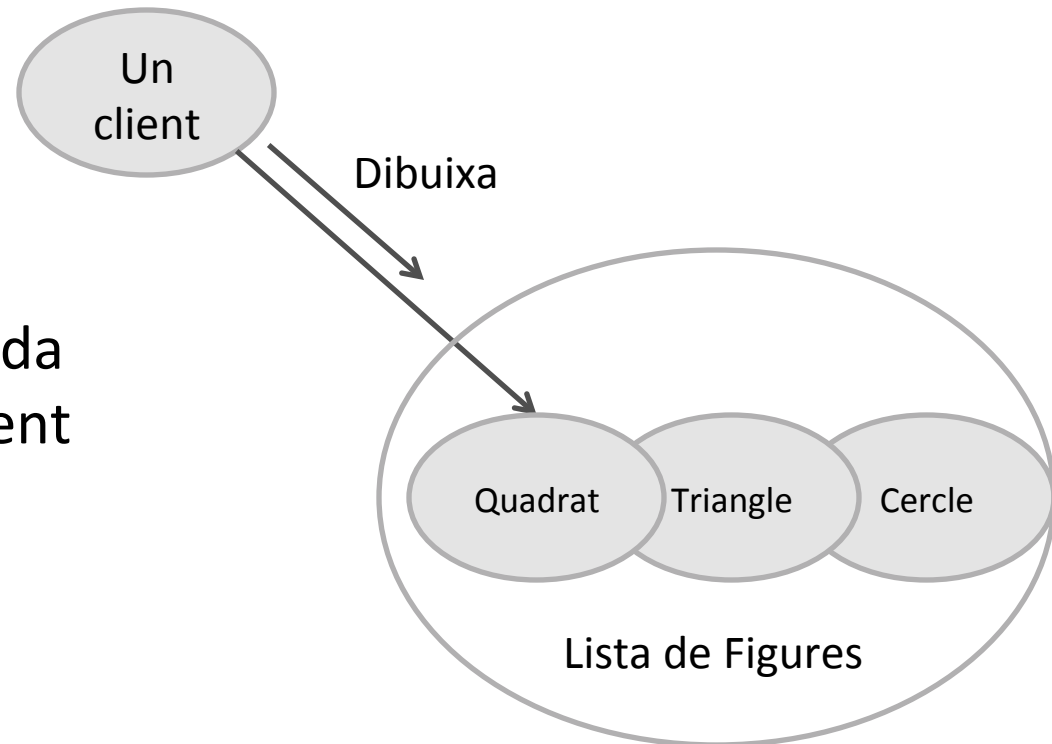
POLIMORFISME

Polimorfisme

- *Origen: poli ('diversos') i morfos ('forma')*
- “Capacitat per a adoptar diverses formes”
- El polimorfisme està lligat estretament amb l’herència
- És la propietat per la qual es poden realitzar tasques diferents invocant la mateixa operació, segons el tipus d’objecte sobre el qual s’invoca.

Polimorfisme

- El Polimorfisme provocarà un canvi de comportament d'una operació depenent de l'objecte al qual s'aplica.
- L'operació és única, però cada classe defineix el comportament d'aquella operació.



Polimorfisme

- És la propietat d'ocultar l'estructura interna d'una jerarquia de classes implementant un conjunt de mètodes de manera independent i diferenciada en cada classe de la jerarquia.
- El polimorfisme **apareix** quan definim un mètode en una classe de la jerarquia (generalment la superclasse) i el reescrivim en, com a mínim, alguna de les classes que formen la jerarquia.
- La reescriptura de mètodes només pot existir en subclasses de la classe en què es defineix o implementa el mètode per primera vegada.

Polimorfisme

- El concepte de polimorfisme es pot aplicar tant a **mètodes** com a **tipus de dades**.
- Així neixen els conceptes de:
 - *Mètodes polimòrfics*, són aquells mètodes que poden avaluar-se o ser aplicats a diferents tipus de dades de forma indistinta.
 - *Tipus polimòrfics*, són aquells tipus de dades que contenen al menys un element amb tipus no especificat.

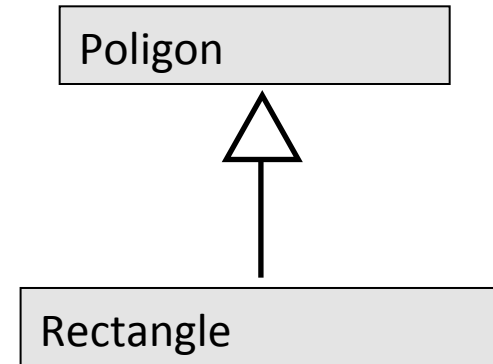
Polimorfisme

- Assignació polimorfa:

Dues implementacions:

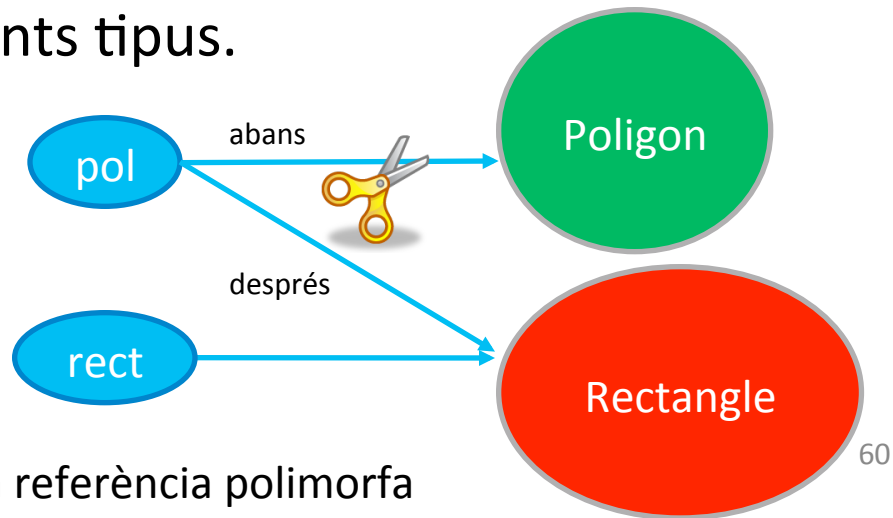
```
Poligon pol = new Poligon();  
Rectangle rect = new Rectangle();  
pol = rect;
```

```
Poligon pol = new Rectangle();
```



- Connexió polimorfa (assignació i passo de paràmetres): quan l'origen i el destí tenen diferents tipus.

- Que passa durant una connexió polimorfa?



Reconnexió de la referència polimorfa

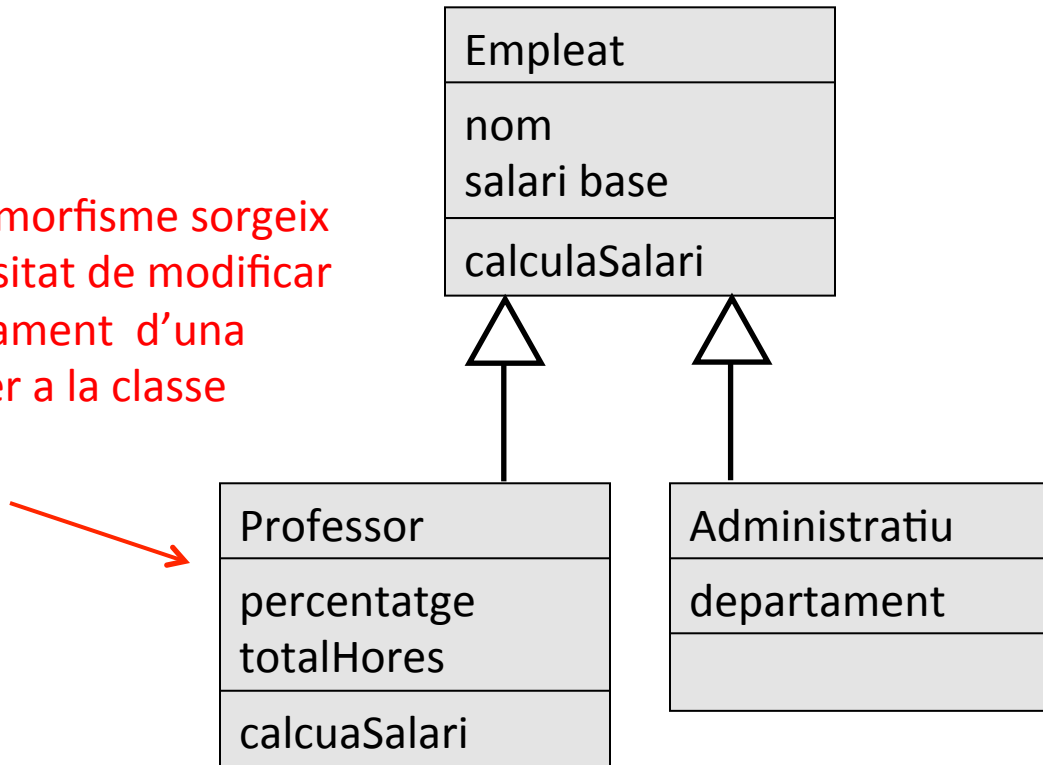
Polimorfisme

- El polimorfisme permet que es decideixi en temps d'execució i de manera automàtica quin dels mètodes cal executar: el mètode heretat o, en cas que existeixi, el mètode sobreescrit.

Exemple 1

- Implementació de l'exemple:

Aquí el polimorfisme sorgeix de la necessitat de modificar el comportament d'una operació per a la classe Professor



```
public abstract class Empleat{
    private String nom;
    private float salariBase;
    public Empleat( String nom, float salariBase) {
        this.nom = nom;
        this.salariBase = salariBase;
    }
    public String getNom() {
        return nom; }
    public float getSalariBase() {
        return salariBase; }
    public void setNom(String nom) {
        this.nom = nom; }
    public void setsalariBase(float salariBase ) {
        this.salariBase = salariBase; }
    public float calculaSalari() {
        float salari = (float) (salariBase * 1.5);
        return salari;
    }
}
```

Empleat.java

Administratiu.java

```
public class Administratiu extends Empleat {  
    private String department;  
    public Administratiu (String nom, float salariBase, String department) {  
        super(nom, salariBase);  
        this.department = department;  
    }  
    public String getDepartment() {  
        return department;  
    }  
    public void setDepartment(String department) {  
        this.department = department;  
    }  
}
```


Professor.java

```
public class Professor extends Empleat {  
    private float percentatge;  
    private float totalHores;  
    // constructor  
    public Professor(String nom, float salariBase, float percentatge, float totalHores) {  
        super(nom, salariBase);  
        this.percentatge = percentatge;  
        this.totalHores = totalHores;}  
    // Getters i setters  
    public float getPercentatge() {  
        return percentatge;}  
    public float getTotalHores() {  
        return totalHores;}  
    public void setPercentatge(float percentatge) {  
        this.percentatge = percentatge;}  
    public void setTotalHores(float totalHores) {  
        this.totalHores = totalHores;}  
    // Reescriptura del mètode calculaSalari  
    public float calculaSalari() {  
        return super.calculaSalari() + (percentatge * totalHores);}  
}
```

Test.java

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

```
        Empleat admin;
```

```
        admin = new Administratiu("Joana",1000, "dep");
```

```
        System.out.println("salari administratiu = " + admin.calculaSalari());
```

→ 1.500.0

```
        Empleat empleat;
```

```
        // Preguntar a l'usuari que vol introduir a l'aplicació:
```

```
        Scanner sc=new Scanner(System.in);
```

```
        System.out.println("Indica 1 per introduir un professor i 2 per introduir un administratiu: ");
```

```
        int resposta = sc.nextInt();
```

```
        if(resposta==1){
```

```
            empleat = new Professor("Joana",1000, 10, 200);
```

```
        }else{
```

```
            empleat = new Administratiu("Joana",1000, "dep");
```

```
        }
```

```
        System.out.println("salari professor = " + empleat.calculaSalari());
```

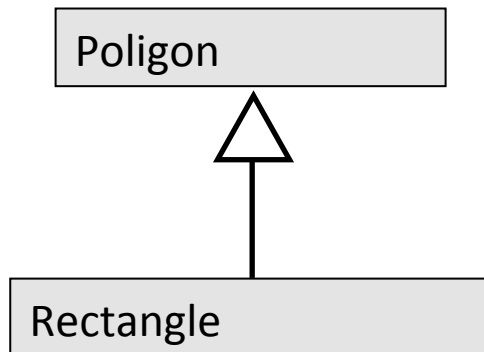
→ Depenent de la resposta de l'usuari, sortirà:
1.500.0 o
3.500.0

```
    }  
}
```

Example 2

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

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



```
public class Test {  
    public static void main(String[] args){  
        Poligon[] pol = new Poligon[2];  
  
        Poligon elemA = new Poligon();  
        Rectangle elemB = new Rectangle();  
  
        pol[0] = elemA;  
        pol[1] = elemB;  
  
        pol[0].imprimiIdentitat();  
        pol[1].imprimiIdentitat();  
    }  
}
```

Sortida per pantalla →

Sóc Poligon
Sóc Rectangle

Polimorfisme vs. Sobrecàrrega

- És important diferenciar entre sobrecàrrega i el polimorfisme (sobreescriptura).

La sobrecàrrega consisteix a definir un mètode nou amb una signatura diferent (nombre i/o tipus de paràmetres).

La sobrecàrrega es pot detectar en temps de compilació.

El polimorfisme és la substitució d'un mètode per un altre en una subclasse mantenint la signatura original.

El polimorfisme es resol en temps d'execució.

Exemple 3

- És important diferenciar entre sobrecàrrega i el polimorfisme (sobreescritura).

```
public class ExempleSobrecarrega{  
    public void metodeExemple(){  
        System.out.println("mètode sense  
        parametres");  
    }  
    public void metodeExemple(int x){  
        System.out.println("mètode amb els  
        parametres" + x);  
    }  
}
```

```
public class ExPolimorfismeMare{  
    public void metode(){  
        System.out.println("mètode original");  
    }  
}
```

```
public class ExemplePolimorfisme extends ExPolimorfismeMare{  
    public void metode(){  
        System.out.println("mètode sobreescrit");  
    }  
}
```

Exemple 3

```
public class ExPolimorfismeMare{  
    public void metode(){  
        System.out.println("mètode original");  
    }  
}
```

```
public class ExemplePolimorfisme extends ExPolimorfismeMare{  
    public void metode(){  
        System.out.println("mètode sobreescrit");  
    }  
}
```

```
public class TestPolimorfisme {  
    public static void main(String[] args) {  
        ExPolimorfismeMare exPol = new ExPolimorfisme();  
        exPol.metode();  
    }  
}
```

→ mètode sobreescrit

Com has d'implementar el mètode per que aparegui per pantalla el missatge?:

mètode original

mètode sobreescrit

Exemple 3

- Com has d'implementar el mètode per que aparegui per pantalla el missatge?
mètode original
mètode sobreescrit

Solució:

```
public class ExPolimorfismeMare{  
    public void metode(){  
        System.out.println("mètode original");  
    }  
}
```

```
public class ExemplePolimorfisme extends ExPolimorfismeMare{  
    public void metode(){  
        super.metode();  
        System.out.println("mètode sobreescrit");  
    }  
}
```

Exercicis

1. Genereu dues classes, A i B, amb els constructors per defecte (és a dir, sense cap argument), que tornin per pantalla algun missatge per saber quin és el constructor de cada una de les classes. Després, genereu una nova classe C que hereti de A, definiu un objecte de B dins de C i un constructor que no faci res. En acabar, definiu un objecte de la classe C i mostreu els resultats.

Exercicis

2. Modifiquen l'exercici anterior per tal que les classes A i B tinguin un constructor amb arguments. Escriviu un constructor per a la classe C i executeu totes les inicialitzacions necessàries dins el constructor de C.

Exercicis

3. Genereu una classe anomenada `Root` que contingui una instància de cada una de les tres classes `Comp1`, `Comp2` i `Comp3`. Deriveu una classe nova, `Node` a partir de la classe `Root` que contingui una instància de les tres classes “component”. En el constructor de totes les classes, col·loqueu un missatge de manera que retorni per pantalla un identificador de la classe on és i mireu el resultat.

Exercicis

4. Genereu una classe que tingui un mètode sobrecarregat. Genereu una classe A que tingui un mètode per a tornar per pantalla un paràmetre que, si és enter, n'hi sumi 1; si és real, n'hi sumi 3,5, i si és una cadena de caràcters, hi afegixi un guió davant i un altre al darrere.

Exercicis

5. Creeu tres classes, A, B i C, de manera que formin una jerarquia de classes. La classe A és la superclasse i les classes B i C hereten de la classe A. Implementeu un mètode en la classe A anomenat toString que no rebi paràmetres i que retorni una cadena que indiqui que és un mètode de la classe A i, posteriorment, sobreescribiu aquest mètode en les classes B i C. Creeu una instància de cada classe i executeu aquest mètode.

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.