

Tema 11

Programació Orientada a Objectes

- 1 Introducció
- 2 Objectes i classes
- 3 Estructura d'un programa OO
- 4 Definició d'una classe
- 5 Creació i ús dels objectes d'una classe
- 6 Propietats dels llenguatges OO:
Abstracció, encapsulament, herència i polimorfisme
- 7 Atributs i mètodes estàtics
- 8 Miscel·lània
9. La classe ArrayList i la classe Vector

1 Introducció

Fins ací hem treballat amb JAVA sense utilitzar pràcticament la Programació Orientada a Objectes (POO). Però per a aprofitar la potència de JAVA és imprescindible conèixer els conceptes i característiques particulars de la POO.

La POO és una metodologia que basa l'estructura dels programes en objectes. En el tema anterior veiérem que un objecte era un conjunt d'atributs. Doncs ara ampliem este concepte: un objecte estarà format per:

- ✓ Una sèrie de característiques (atributs).
- ✓ Unes operacions que es poden realitzar sobre eixe objecte (mètodes).

Com ja veiérem, crearem una classe i, a partir d'ella, crearem objectes. És a dir, cada classe descriurà un conjunt d'objectes que tenen les mateixes propietats.

Els llenguatges de POO es classifiquen com a llenguatges de 5a generació.

2 Objectes i classes

2.1 Objectes

Definició informal d'objecte

Un objecte és qualsevol persona, animal o cosa que veiem al nostre voltant. Un objecte es distingeix d'altres objectes per tindre unes determinades **característiques** i “servir” per a alguna cosa, o dit d'una altra forma, es poden realitzar distintes **operacions** amb/sobre eixe objecte.

Exemple: Una casa qualsevol és un objecte. Una altra casa és altre objecte.

- ✓ **Característiques:** Quantitat de pisos, alçària total en metres, color de la façana, quantitat de finestres, quantitat de portes, ciutat, carrer i número on està ubicada, etc.
- ✓ **Operacions:** Construir-la, destruir-la, pintar la façana, obrir una nova finestra, pintar habitacions, etc.

Evidentment, un objecte pot definir-se en funció de multitud de característiques i es poden realitzar innumerables operacions sobre ell. El programador determinarà quines característiques i quines operacions cal definir en un objecte.

Per exemple, sobre l'objecte casa pot no ser necessari conèixer la quantitat de finestres, i, per tant, eixa característica no formarà part de l'objecte definit pel programador. Igual podria passar amb l'operació d'obrir una nova finestra.

Terminologia formal

En terminologia de POO, a les característiques de l'objecte se'ls denomina **atributs** i a les operacions que es fan sobre l'objecte, **mètodes**. A més, cada objecte ha de tindre un nom (casa1, casa2, ...), que és l'**identificador** de l'objecte.

TERMINOLOGIA INFORMAL	TERMINOLOGIA FORMAL
Nom	Identificador
Característiques o dades	Atributs
Operacions (funcions i procediments)	Mètodes

identificador
atributs
mètodes

És a dir: un objecte és un conjunt d'**atributs** i **mètodes** interrelacionats i que s'identifica per un nom o **identificador**.

Exemples d'objectes:



casa1

3 plantes, 1 porta,
5 finestres,
color blanc, ...

construir,
pintar, ...



casa

2 plantes, 1 porta,
2 finestres,
color os, ...

construir,
pintar, ...



cotxe1

matríc. DMB 1234,
diesel, 5 portes, ...

fabricar, accelerar,
omplir dipòsit, ...

2.2 Classes

En la POO cal distingir entre dos conceptes íntimament lligats: *classe* i *objectes*.

En els exemples anteriors, teníem 3 objectes: 2 cases i 1 cotxe. És a dir:

OBJECTES

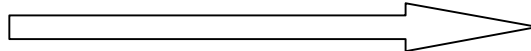
casa 1



casa2



Els objectes *casa1* i *casa2*
pertanyen a la classe *Casa*.



CLASSES

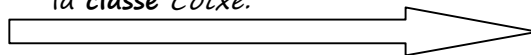
Casa



cotxe1



L'objecte *cotxe1* pertany a
la classe *Cotxe*.



Cotxe



És a dir, la nostra tendència és classificar els objectes segons unes característiques comunes (classe). Per exemple, les persones que van a l'institut es poden classificar com estudiants, docents, conserges, neteja. Fins i tot podem establir subclasses. Per exemple, els estudiants poden ser d'ESO, batxillerat o cicles; els docents poden ser definitius, interins o en pràctiques, etc.

Definició informal de classe

La classe es pot definir com el conjunt d'objectes que compartixen unes característiques comunes i un comportament comú. És un motlle (com ja diguérem al tema dels registres) que conté quina **descripció** cal guardar dels objectes d'eixe tipus i quines **operacions** es poden fer sobre ells.

Tot objecte pertany a alguna classe. Un objecte és una entitat concreta que existix en temps i espai, però una classe representa només una abstracció. Tots els objectes que pertanyen a una classe tenen els mateixos atributs i mètodes. Els objectes que pertanyen a una mateixa classe es diferencien en l'identificador i, possiblement, en els valors que contenen els seus atributs.

Terminologia formal

Hi ha una analogia entre variables i objectes. És a dir: *casa1* i *casa2* són variables, però un tant especials, són **objectes**. De quin tipus? Del tipus *Casa*. Per tant, *Casa* és la **classe** d'eixos objectes.

TERMINOLOGIA INFORMAL	TERMINOLOGIA FORMAL
Tipus	Classe
Variables	Objectes
Valor	Estat

3 Estructura d'un programa OO

La forma de crear un programa OO és la següent:

1r) Definir la classe o classes necessàries.

```
class Casa {  
    // Definició dels atributs  
    // Definició dels mètodes  
}
```

Nota: els noms de les classes cal que siguin representatius, en singular i majúscula.

2n) Crear (en la classe principal) els objectes que necessitem de cada classe.

```
Casa casa1 = new Casa();  
Casa casa2 = new Casa();
```

Al declarar *casa1* i *casa2* com a objectes de la classe *Casa*, s'està indicant que *casa1* i *casa2* tindran els atributs i mètodes de la classe *Casa*.

3r) Utilitzar eixos objectes

3.1 Exemple

Ho vorem
al punt 4

```
// CLASSE Alumne
class Alumne {
    // ATRIBUTS
    String dni;
    String nom;
    String cognoms;
    int edat;
    // MÈTODES
    void setDni(String d)        { dni = d;}
    void setNom(String n)        { nom = n;}
    void setCognoms(String c)    { cognoms = c;}
    void setEdat(int e)          { edat = e;}
    String getDni()              { return dni; }
    String getNom()              { return nom; }
    String getCognoms()          { return cognoms; }
    int getEdat()                { return edat; }
    String nomComplet()          { return nom + " " + cognoms; }
    void incrementarEdat(int anys){ edat += anys; }
    // CONSTRUCTOR d'objectes de la classe Alumne
    Alumne(String d, String n, String c, int e){
        dni = d;
        nom = n;
        cognoms = c;
        edat = e;
    }
}
```

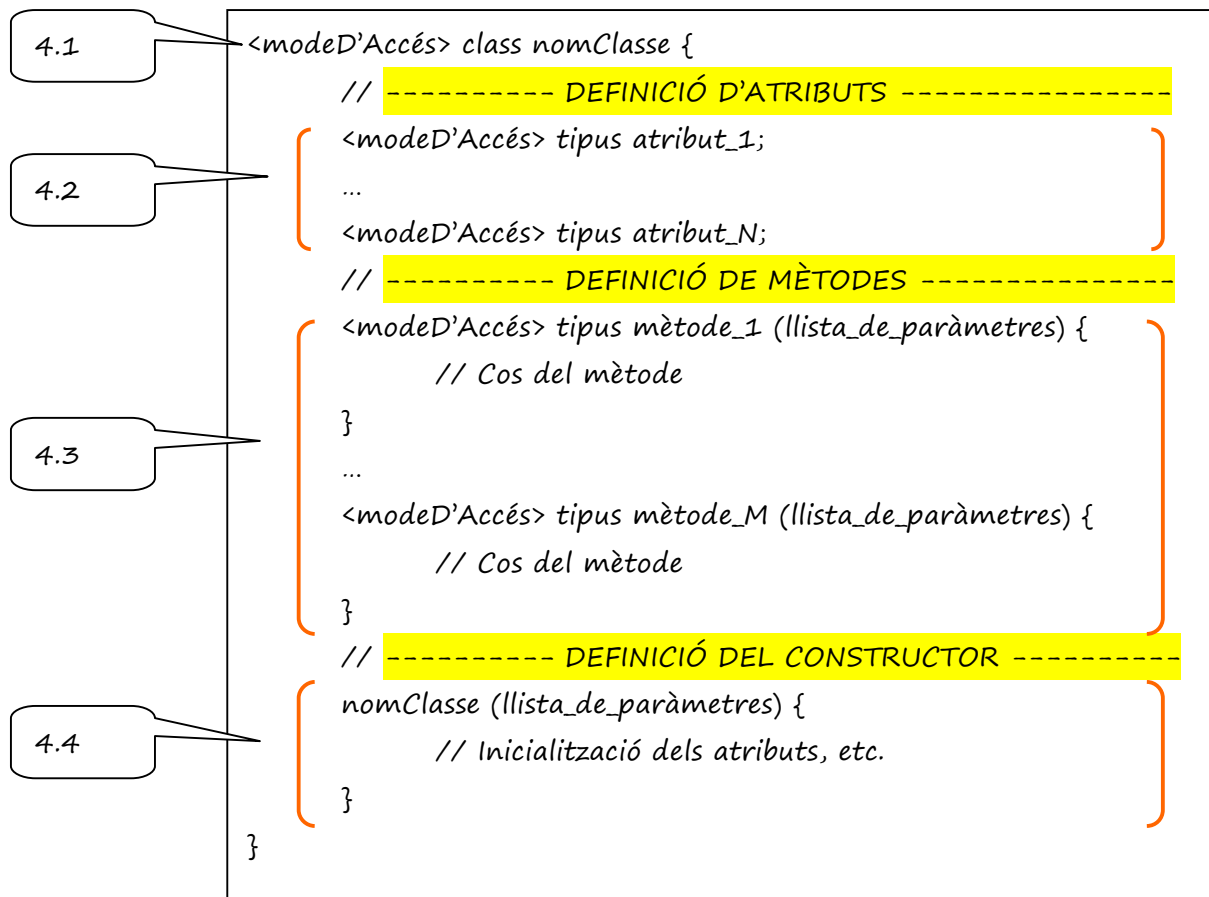
Ho vorem
al punt 5

```
// CLASSE PRINCIPAL
public class Main {

    public static void main(String[] args) {
        // DEFINICIÓ I ÚS D'UN OBJECTE
        Alumne a1 = new Alumne("55555555", "Pep", "Garcia Martí", 18);
        if (a1.getEdat() >= 18)
            System.out.println("L'alumne " + a1.getDni() + " té carnet verd");
        a1.setCognoms("Garcia i Martí");
        System.out.println(a1.nomComplet());
    }
}
```

4 Definició d'una classe

Esta és la sintaxi de Java per a definir una classe:



Exercicis

1. Mira al teu voltant. Escriu una llista d'alguns objectes i posal's un nom a cadascun. Classifica'ls. Finalment, escriu una sèrie de característiques de la classe i d'operacions. Posa també valors a les característiques de cada objecte. Exemple:

<u>Classe:</u>	taula	Característiques: color, estat_de_conservació, coixaSN
		Operacions: arreglar-la, pintar-la, crear-la, destruir-la
<u>Objectes:</u>	taula1(verd,bo,N), taula2(verd,bo,N), taula3(verd,mal,S)	

2. Un concessionari vol una aplicació per a tindre un control dels vehicles que ven. Crea una aplicació anomenada *Concessionari*. Allí crea la classe *Cotxe*. De moment, sense atributs ni mètodes. Ja anirem ampliant-la conforme anem veient més temari.

4.1 Modes d'accés de la classe

El mode d'accés de la classe pot ser un dels següents:

- *public* indica que la classe és pública, que pot ser utilitzada des de qualsevol altra classe. (Si no es posa *public*, només pot ser accedida des del mateix paquet).
- *abstract* / *final* s'utiliza per a l'herència de classes (ja ho vorem).
 - *abstract* per a iniciar una herència. No pot ser instanciada ja que té algun mètode abstracte (està declarat però li falta el codi del cos). Les subclasses són les que proporcionaran codi al mètode.
 - *final* per a finalitzar una herència. No es podran crear subclasses d'una classe final.

4.2 Atributs

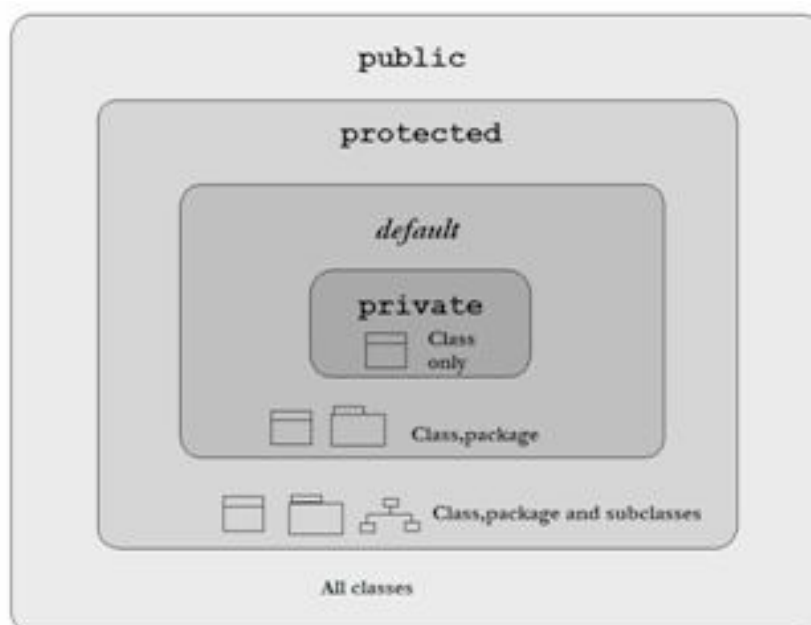
Són les variables membre que caracteritzen l'objecte. El seu valor en un moment donat indica l'estat de l'objecte.

Mitjançant els atributs es pot definir la **informació oculta** dins d'un objecte, és a dir, informació que només pot ser accedida o manipulada pels mètodes definits dins de l'objecte. És el que vorem en l'apartat d'**encapsulament**.

Esta és la sintaxi de Java per a definir un atribut:

```
<nivellD'Accés> <TipusDeDades> nomAtribut ;
```

El nivell d'accés als atributs és opcional. Anem a veure quins nivells poden ser:



Ara ho vorem
detalladament:

Nivell d'accés dels atributs (i dels mètodes)

Els atributs (i mètodes) sempre són accessibles des de la classe que els definix. A banda d'això, podem fer que també puguin ser accessibles des d'altres llocs. Per a fer això, davant de la definició dels atributs posarem un modificador que ens indicarà el nivell d'accés que té cada atribut:

NIVELLS D'ACCÉS ALS ATRIBUTS (I MÈTODES)					
Modificador de l'atribut (o del mètode)	Descripció de l'accés	Des d'on podem accedir?			
		Mateixa Classe	Mateix Paquet	Mateixa Subclasse	Qualsevol altre lloc
public	Des de <u>qualsevol lloc</u>	S	S	S	S
protected	Des del <u>mateix paquet</u> de la classe i en les <u>subclasse</u> s	S	S	S	N
	Des del <u>mateix paquet</u> de la classe	S	S	N	N
private	Des de cap altre lloc. Només des de la <u>pròpia classe</u>	S	N	N	N

Nota: el **protected**, si estem intentant accedir a un atribut des d'una subclasse que està en altre paquet que la classe, podrem fer-ho si l'atribut és d'un objecte de la subclasse, però no si és d'un objecte de la superclasse.

Consell: els atributs solen ser privats, i les classes donen accés a ells mitjançant els mètodes de lectura i modificació d'eixos atributs, que normalment seran públics.

Tipus de dades dels atributs

El tipus de dades dels atributs pot ser qualsevol: enter, real, lògic, caràcter, cadena, vector, matriu, altra classe (ja ho vorem) ...

Exercici

1. Posa els següents atributs a la classe *Cotxe*: *numBastidor*, *matricula*, *marca*, *model*, *color*, *preu*. També l'atribut privat *revisions* (vector de 5 booleans).

4.3 Mètodes

Són les operacions (procediments o funcions) que s'apliquen sobre els objectes. Entre altres coses, permeten crear els objectes, canviar el seu estat (valor dels atributs), consultar-lo, etc. Si des d'elles usem els atributs, convé posar primer el "this".

Exemple: Tenim una classe *Cercle* amb l'atribut *radi* i amb els següents mètodes:

```
class Cercle {  
    // ----- ATRIBUTS -----  
    int radi;  
    // ----- MÈTODES -----  
    int diametre() {  
        return (2 * this.radi);  
    }  
    float area() {  
        return (Math.PI * this.radi * this.radi);  
    }  
    float perímetre() {  
        return (2 * Math.PI * this.radi);  
    }  
    void incrementaRadi(int increment) {  
        this.radi += increment;  
    }  
    // ----- MÈTODE CONSTRUCTOR -----  
    ...  
}
```

Són l'eina bàsica per a comunicar-nos amb els objectes. És a dir, si tenim objectes, voldrem comunicar-nos amb ells. Per a això, si volem fer alguna cosa sobre ells, els enviarem el missatge corresponent. Per exemple, si tinc un objecte *cercle1*, jo li podré enviar un missatge per a que es cree, li podré enviar un missatge per a dir quina és la seua grandària, li podré enviar un missatge per a que em done la seua àrea, etc:

```
public class Main {  
    public static void main(String[] args) {  
        Cercle cercle1 = new Cercle(15);  
        ...  
        System.out.println( "L'àrea és: " + cercle1.area() );  
        cercle1.incrementaRadi(3);  
        ...  
    }  
}
```

Enviem un missatge a l'objecte *cercle1* per a que ens retorne la seua àrea.

Enviem un missatge a l'objecte *cercle1* per a que augmente el seu radi en 3.

Nota: quan encara no havíem vist la POO, posàvem la paraula **static** davant de la definició dels mètodes. Ara no la posem, ja que estos mètodes actuen sobre un objecte en concret. Per exemple, en la crida a `cercle1.area()`, estem cridant al mètode `area()` però de l'objecte `cercle1`. Ara bé, el `static` també pot usar-se en POO. Ja ho vorem més endavant.

Característiques generals dels mètodes

Com els mètodes són funcions o procediments, cada mètode té:

- ✓ un nom
- ✓ zero o més paràmetres d'entrada
- ✓ zero o 1 paràmetre d'eixida
- ✓ una implementació d'algorisme (el cos del mètode)

Exercicis

2. Modifica la classe *Cotxe* que tens creada: afegix-li els següents mètodes:

IDENTIFICADOR	ENTRADA	EIXIDA	COS
<i>matricular</i>	Número de matrícula	(res)	Posar-li el número de matrícula al cotxe.
<i>mostrarRevisions</i>	(res)	(res)	Mostrar per pantalla les revisions
<i>mostrarDades</i>	(res)	(res)	Mostrar per pantalla totes les dades del cotxe.
<i>pintar</i>	Nou color del cotxe	(res)	Assignar el color al cotxe.
<i>augmentarPercentatgePreu</i>	Percentatge	(res)	Augmentar el preu del cotxe en el percentatge corresponent.
<i>Revisar</i>	Número de revisió	(res)	Posar a vertader la revisió corresponent.
<i>quantitatRevisions</i>	(res)	Quantitat de revisions fetes.	Calcular les revisions fetes a partir del vector corresponent.

4.4 Mètode constructor

Definició

Un constructor és un mètode especial que s'executa automàticament quan es crea un objecte (quan li assignem memòria amb el `new`).

Utilitat

Servix, principalment, per a donar valors inicials als atributs de l'objecte.

Exemple

```
// CLASSE Alumne
class Alumne {
    // ATRIBUTS
    String dni;
    String nom;
    String cognoms;
    int edat;
    // MÈTODES
    ...
    // CONSTRUCTORS d'objectes de la classe Alumne
    Alumne(String d, String n, String c, int e){
        this.dni = d;
        this.nom = n;
        this.cognoms = c;
        this.edat = e;
    }
    Alumne(){
        System.out.println("Dni: ");
        System.out.println("Nom: ");
        System.out.println("Cognoms: ");
        this.dni = llegirCadena();
        this.nom = llegirCadena();
        this.cognoms = llegirCadena();
    }
}

// CLASSE PRINCIPAL
public class Main {
    public static void main(String[] args) {
        // DEFINICIÓ D'OBJECTES
        Alumne a1 = new Alumne("55555555", "Pep", "Garcia Martí", 18);
        Alumne a2 = new Alumne();
        Alumne a3 = new Alumne();
        ...
    }
}
```

Constructor amb
paràmetres

Constructor
sense paràmetres

Propietats dels constructors

- ✓ L'identificador del constructor sempre és el nom de la classe.
- ✓ Este mètode rep zero o més paràmetres d'entrada i els utilitza per a inicialitzar els valors dels atributs de l'objecte.
- ✓ No té paràmetres d'eixida (ni tan sols posarem el void).
- ✓ És possible que una classe tinga més d'un constructor (sempre amb el mateix nom) però es diferenciaran en la quantitat o tipus dels paràmetres d'entrada. En eixe cas direm que el constructor està **sobrecarregat** (ja vorem més endavant la sobrecàrrega de mètodes). En eixe cas, des d'un constructor es pot cridar a altre amb: `this(paràmetres);`
- ✓ No podem **invocar als constructors** directament. Només es fa amb l'operador `new`, mentre instanciem la classe.
- ✓ És convenient que tota classe tinga com a mínim un constructor.
- ✓ Si no hem especificat cap constructor, es crea automàticament un constructor sense paràmetres.

Exercicis

3. Crea el constructor de la classe `Cotxe` de forma que es passen com a paràmetres el número de bastidor, la matrícula, marca, model, color i preu. El constructor deu inicialitzar amb estos valors l'estat de l'objecte. El vector de revisions es posarà tot a fals.
4. Crea un altre constructor de la classe `Cotxe`, de forma que reba com a paràmetres el número de bastidor, la marca i el model. Este constructor no inicialitzarà cap variable membre, sinó que cridarà a l'altre constructor per a que ho faça ell, amb l'ús de `this(paràmetres)`.
5. Crea altre constructor a la classe `Cotxe`, de forma que no reba cap paràmetre. El constructor haurà de preguntar per pantalla els valors del número de bastidor, marca i model. Igual que l'exercici anterior, intenta cridar al primer constructor per a que s'encarregue ell d'inicialitzar els atributs. Per què no et deixa? No hi haurà més remei que inicialitzar els atributs directament.

5 Creació i ús dels objectes d'una classe

5.1 Creació d'objectes

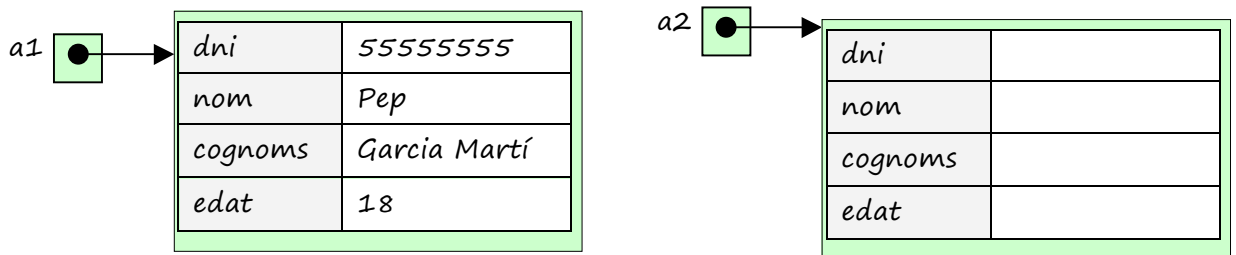
Ja sabem que els objectes es creen amb l'operador `new`:

Sintaxi:

```
NomClasse nomObjecte = new NomClasse();
```

Exemples:

```
Alumne a1 = new Alumne("55555555", "Pep", "Garcia Martí", 18);  
Alumne a2 = new Alumne();  
...
```



Com ja hem vist anteriorment, el `new` fa el següent:

- ✓ Reserva memòria per a l'objecte.
- ✓ Crida al mètode constructor de la classe corresponent (si existix eixe mètode), qui sol omplir eixa memòria.
- ✓ Assigna a l'objecte una referència (adreça) a la memòria reservada.

El fet de crear objectes també s'anomena *instanciar una classe*.

Nota: el `String` no és un tipus sinó una classe (per això es posa en majúscula, a diferència de `int`, `char`, `float`, etc). Per tant, si volem "una variable de tipus cadena", el que hauríem de fer és declarar un objecte de la classe `String`:

```
String nom = new String("Pep");
```

Ara bé, hi ha una forma abreviada de fer-ho per als objectes de la classe `String`:

```
String nom = "Pep";
```

Exercicis

6. Crea l'objecte *cotxeMeu* (de la classe *Cotxe*) sense passar-li cap paràmetre al constructor.
7. Demana per teclat les dades d'un cotxe (número de bastidor, marca, model, color i preu). Crea l'objecte *cotxeTeu* (de la classe *Cotxe*) passant-li com a paràmetres al constructor les dades que s'han arreglat per teclat. Nota: passa una cadena buida per a la matrícula.
8. Intenta crear altre cotxe (*cotxeProva*) passant-li al constructor només un paràmetre (per exemple, la matrícula). Et deixa? Per què?

5.2 Ús dels objectes

Els registres que vam veure al tema anterior ja diguérem que són objectes. I ja diguérem de quina forma podíem utilitzar els atributs dels objectes:

```
alumne1.nom = "Pep";  
System.out.println( alumne1.nom );
```

Però hem dit que els objectes, a més dels atributs, tenen mètodes. Estos ens permeten accedir també als atributs dels objectes. Com?

Per cada atribut de la classe, sol haver un mètode per a consultar el valor i altre per a donar-li valor. Els noms d'eixos mètodes solen ser *get* i *set* respectivament, més el nom de l'atribut.

Per exemple, per a accedir a l'atribut *nom* dels objectes de la classe *Alumne*, haurem de definir els mètodes *setNom* i *getNom* a eixa classe (tal i com s'indica a l'apartat 3.1) i utilitzar-los així:

```
alumne1.setNom("Pep");  
System.out.println( alumne1.getNom() );
```

Forma aconsellable
d'accedir als atributs
d'un objecte

Esta forma d'accedir als atributs mitjançant mètodes de la classe és més aconsellable que l'accés directe als atributs. Els motius els vorem més endavant, a l'apartat de l'encapsulament.

Exercicis

9. Assigna 10000 euros al preu del cotxeMeu.
10. Demana una matrícula per teclat i posa-li-la a cotxeTeu mitjançant el mètode corresponent que ja té la classe Cotxe.
11. Pinta el cotxeMeu de groc i el cotxeTeu de lila (usant el mètode corresponent).
12. Sense usar els mètodes, pinta els 2 cotxes de verd cuquet.
13. Usant el mètode revisar, passa la revisió del 1r any de cotxeMeu i la del 2n any dels 2 cotxes. Passa també la revisió de l'any 7 del cotxeMeu. Si dóna error, corregix el mètode corresponent.
14. Ajudant-te del mètode quantitatRevisions i de mostrarDades, mostra les dades del cotxe que ha passat més revisions.
15. Crea un vector de 5 cotxes, anomenat cotxesAparador. Inicialitza'ls tots usant el constructor sense paràmetres.
16. Sense utilitzar cap mètode, intenta passar la primera revisió d'algun cotxe de l'aparador. Primer, intenta-ho fer amb l'atribut revisions com a públic. Després, privat. Et deixa? Ara fes-ho amb el mètode revisar. Et deixa?

5.3 Eliminació d'objectes

Igual que es creen els objectes (amb *new*), com que ocupen espai en memòria, és convenient alliberar eixa memòria quan ja no vagen a utilitzar-se els objectes durant l'execució del programa. En molts llenguatges de programació és el programador qui ha de posar les instruccions necessàries per a fer-ho, però en altres llenguatges, com Java, hi ha un mecanisme automàtic anomenat el *Garbage Collector* (recolector de fems), que és qui s'encarrega de fer este alliberament de recursos. Per tant, el programador no s'ha de preocupar de res.

Com sap el *Garbage Collector* quins són els objectes que s'han d'eliminar? Doncs perquè l'objecte ja no té referències, degut a que la variable ha quedat fora d'àmbit o perquè manualment se li ha assignat el valor *null*. Aleshores, el *Garbage Collector* periòdicament busca estos objectes i els elimina de la memòria.

6 Propietats dels llenguatges OO

Els llenguatges de programació OO han de complir unes propietats bàsiques:

- Abstracció
- Encapsulament
- Herència
- Polimorfisme

6.1 Abstracció

Definició

Capacitat per a aïllar un conjunt d'informació i/o comportaments relacionats.

Avantatges

- ✓ Reduïx la complexitat dels programes (dividix i guanyaràs)
- ✓ Fomenta la reutilització del codi

Evolució dels mecanismes d'abstracció

MECANISMES D'ABSTRACCIÓ	CARACTERÍSTIQUES
Funcions (i procediments)	És un conjunt d'instruccions, que es pot parametritzar.
Mòduls (o unitats, llibreries, Paquets...)	Són agrupacions de funcions.
Objectes (i classes)	Agrupació d'estructura de dades i de les operacions que volem fer sobre estes. Incorpora tècniques com l'herència, polimorfisme, etc.

Tècnica d'abstracció en la POO

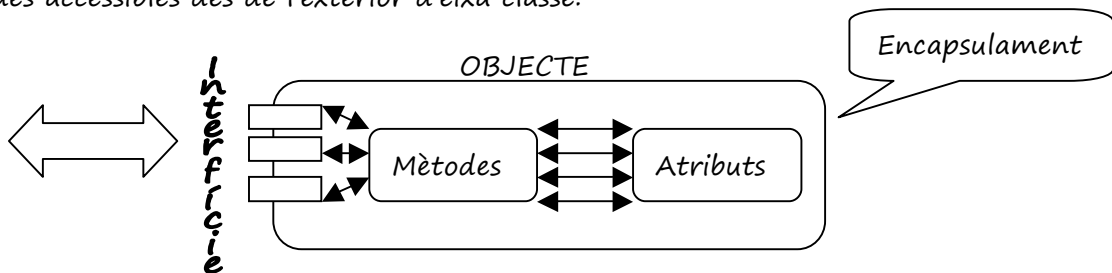
És la capacitat per a encapsular atributs i mètodes comuns a un determinat conjunt d'objectes i emmagatzemar-los en una classe.

6.2 Encapsulament

Hem dit que una classe es un conjunt d'atributs i mètodes. I que un objecte tindrà uns valors (un estat) en eixos atributs. Els usuaris dels objectes poden accedir als atributs directament, però no és aconsellable (per a fer més mantenible el programa).

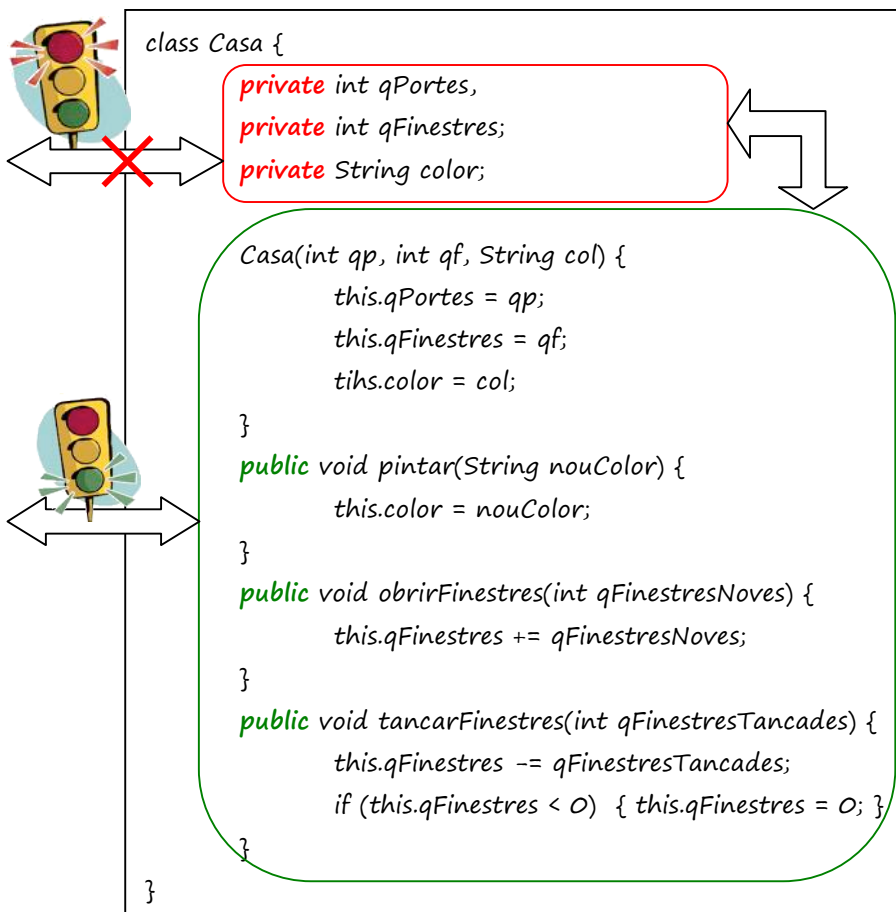
L'encapsulament és la propietat que tenen els objectes d'ocultar els seus atributs i/o mètodes a altres parts del programa.

La forma natural de construir una classe és amb atributs que, en general, no són accessibles fora del mateix objecte, sinó que únicament poden modificar-se a través dels mètodes accessibles des de l'exterior d'eixa classe.



És a dir: l'estructura interna d'un objecte normalment està oculta als usuaris de l'objecte. I l'única forma que l'usuari d'un objecte pot comunicar-se amb ell és a través dels mètodes públics de la classe. Este conjunt de mètodes públics és la interfície.

Exemple



La classe *Casa* té atributs privats i, per tant, no es poden accedir a ells des de fora de la classe.

Però la classe *Casa* té uns mètodes públics i, per tant, sí que poden ser accedits des de fora de la classe. Estos mètodes són els que accediran als atributs privats.

Anem a vore com podem accedir als atributs privats:

Des de fora de la classe (per exemple, en la classe principal, al mètode *main*), ens definim un objecte de la classe *Casa* a qui anomenem *macasa*. Creem *macasa* amb 1 porta, 2 finestres i de color blanc:

```
public class ProvesEncapsulament {  
    public static void main(String[] args) {  
        ...  
        Casa macasa = new Casa(1, 2, "blanc");  
        ...  
    }  
}
```

Ara volem utilitzar eixa casa. Suposem que volem obrir-li una nova finestra:

- ✓ Amb la filosofia tradicional de programació estructurada, fariem:

~~macasa.qFinestres++;~~

Però ara NO podem fer-ho perquè l'atribut *qFinestres* és **privat**.

- ✓ L'altra forma de fer-ho, amb filosofia de POO, és amb un mètode públic:

macasa.obrirFinestres(1);

És a dir, els usuaris dels objectes no tenen per què saber quins atributs tenen els objectes ni quin nom tenen, però sí que necessiten saber la interfície: els noms dels mètodes per a accedir als atributs, els seus paràmetres d'entrada i el paràmetre d'eixida. Per tant, cal cridar al mètode *obrirFinestres()* de l'objecte *macasa*. Li passem com a paràmetre un 1 perquè el mètode s'encarregue d'incrementar en 1 l'atribut *qFinestres*.

Per tant, la forma normal de declarar la classe *Casa* consisteix a definir una sèrie d'atributs (*qFinestres*, *qPortes*, *color*) no accessibles des de fora de la classe, sinó únicament a través de determinats mètodes (*obrirFinestres()*, etc).

Per què usar encapsulament?

És a dir: per què ocultar els atributs als usuaris dels objectes? Un dels motius és per a evitar errors posteriors. Si donàrem permís per a accedir directament als atributs, podria donar-se el cas que, per exemple, llevàrem finestres sense controlar si en tenim suficients, podent-se donar el cas de tindre una quantitat negativa de finestres. Per tant, són els mètodes dels objectes qui controlen els errors. Així, els usuaris de l'objecte no s'han de preocupar si hi ha finestres o no abans de tancar-les.

Exercicis

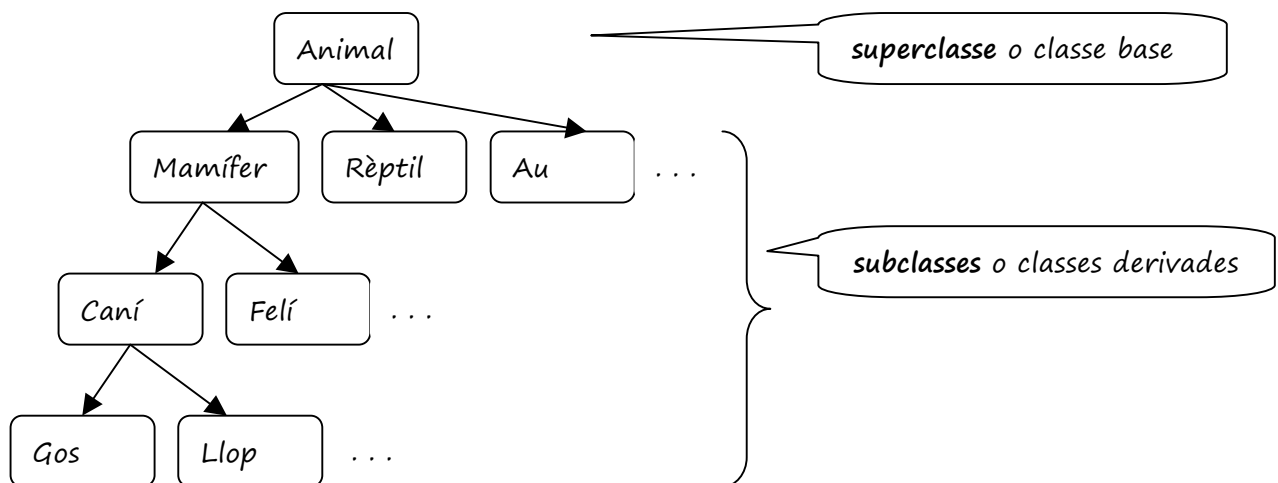
17. Per a aconseguir la propietat d'encapsulament de la nostra classe *Cotxe*, posa tots els seus atributs privats. Per tant, per a poder escriure i llegir d'eixos valors, crea en la classe *Cotxe* els mètodes que cregues que fan falta.

18. Netbeans encapsula automàticament el que tu li digues. Per a provar-ho, crea una classe amb atributs no privats. Estant el cursor dins la classe, en el menú contextual tria Refactor → Encapsulate Fields. Voràs que l'encapsulament ha fet que només es pugui accedir als atributs mitjançant els mètodes: ha deixat com a privats els atributs i ha creat els mètodes set i get per a accedir als atributs des de fora de la classe.

6.3 Herència

Què és l'herència?

L'herència és un dels principals avantatges de la POO. Ens permet establir una jerarquia de classes. Podrem definir classes específiques a partir d'una altra més general.



Propietats de l'herència

Si definim una herència entre 2 classes:

- ✓ La subclasse hereta els atributs i mètodes de la superclasse.
- ✓ La subclasse pot tindre nous atributs i mètodes.
- ✓ La subclasse pot redefinir atributs i mètodes de la superclasse (per exemple: canviar les operacions d'un mètode o el tipus d'un atribut).

Avantatges

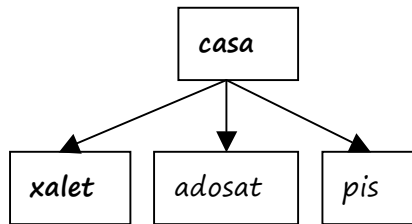
L'avantatge principal de l'herència és la reusabilitat o reutilització del codi. És a dir: una subclasse pot aprofitar el codi d'una superclasse ja existent, modificant-lo mínimament per a adaptar-lo a les seues necessitats.

Exemples

L'herència és la forma natural de definir objectes en la vida real.

Per exemple:

- Què és un **xalet**?
- Una **casa** amb jardí.



Vegem les propietats de l'herència en este exemple del xalet:

- El xalet, com a casa que és, també tindrà el atributs de la casa (portes, finestres, color...) i els mètodes de la casa (construir, pintar...)
- El xalet tindrà nous atributs: m2 de jardí, quantitat d'arbres, etc i també nous mètodes: plantar un arbre, arrancar un arbre, etc.
- A més, amb la propietat de polimorfisme (que vorem més endavant) podrem redefinir algun mètode: podrem fer que el mètode *pintar*, a més de fer el que feia, que pintara també la tanca).

Altre exemple: Un ànec és una au que nada. Manté les mateixes característiques que les aus i únicament hauria que declarar un mètode nou: **nadar**.

Implementació

Volem crear una subclasse a partir d'una superclasse ja creada. Eixe procés es coneix com derivar una classe. Per a derivar una classe usarem la paraula **extends**.

```
class nomSuperclasse {  
    ... // Atributs i mètodes de la superclasse (ho heretaran les subclasses)  
}  
  
class nomSubclasse extends nomSuperclasse {  
    ... // Altres atributs i mètodes només de la subclasse  
    ... // Mètodes de la superclasse que necessitem modificar  
}
```

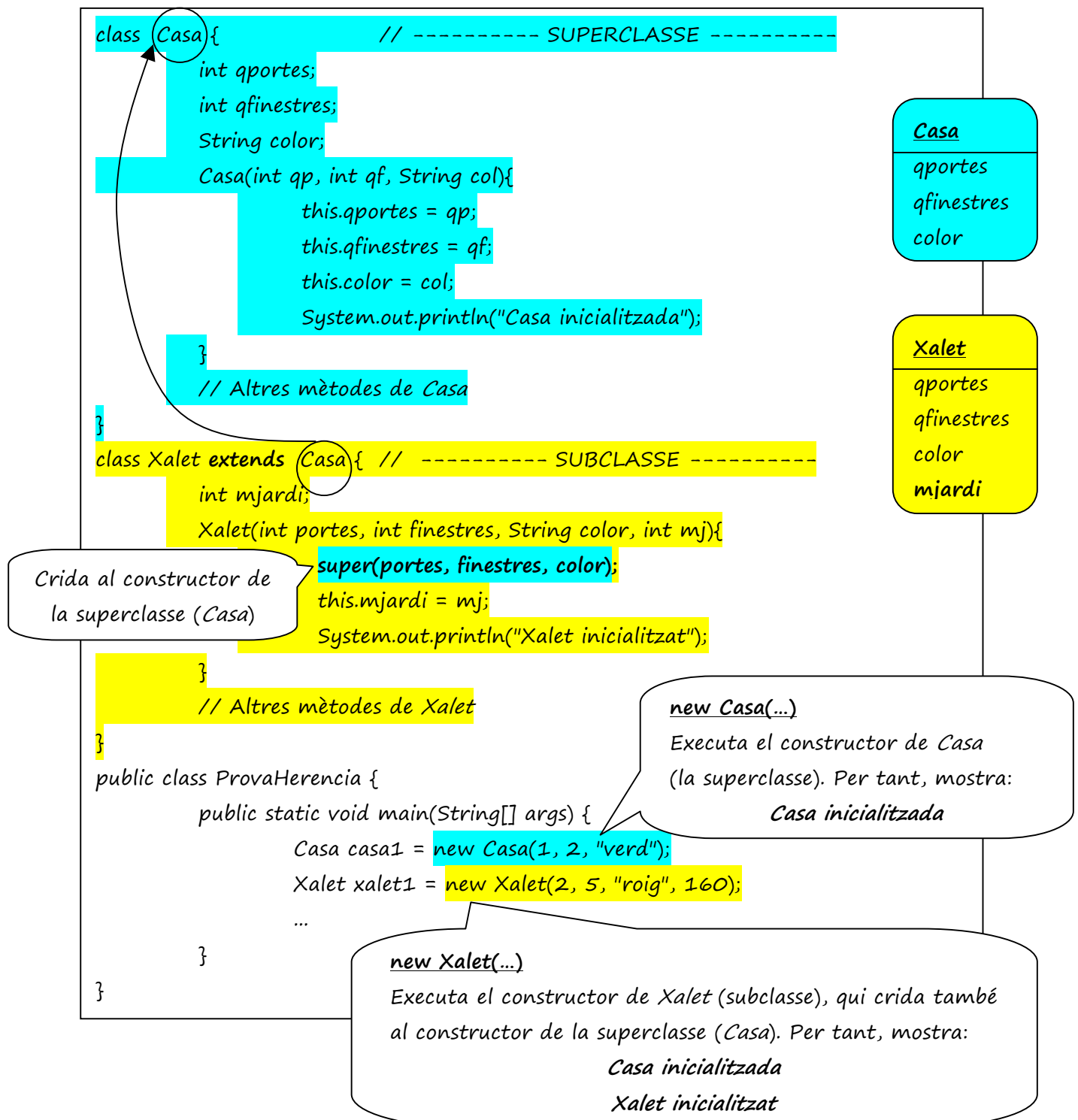
El constructor de la subclasse

Si la superclasse té algun constructor amb paràmetres però no té cap constructor sense paràmetres, és obligatori:

- Que la subclasse també tinga el seu constructor
- Que la **primera instrucció** del constructor invoque el constructor de la superclasse. Això es fa amb: `super(paràmetres_constructor_superclasse);`

Nota: la crida a `super(...)` només pot estar dins dels constructors

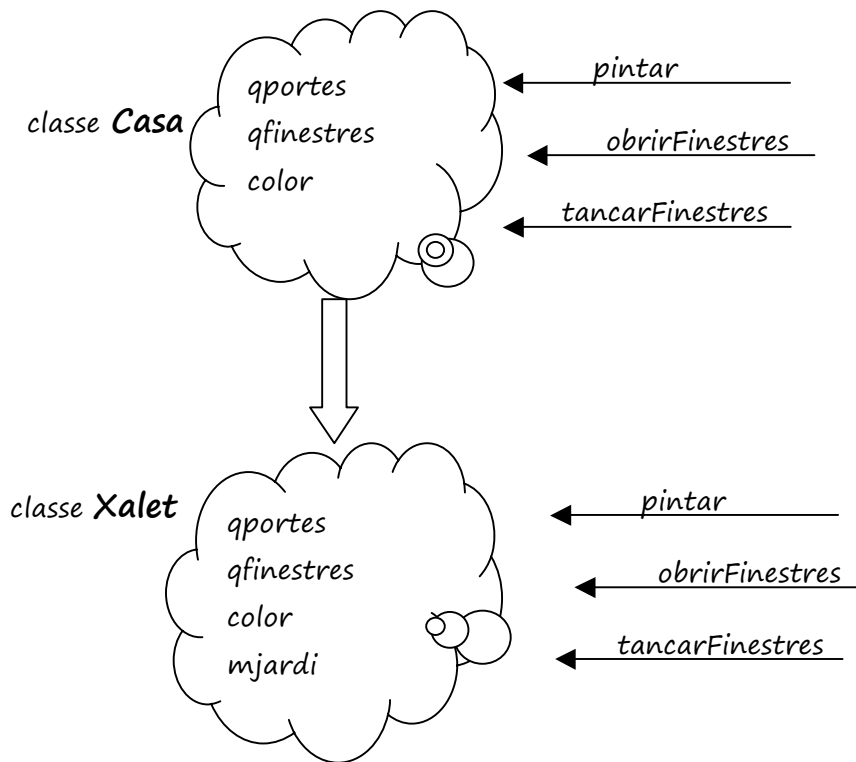
Exemple d'implementació d'una herència



Explicació de l'exemple de l'herència:

Suposem que tenim construïda la classe *Casa* i volem definir una nova classe que represente als xalets. Volem que la nova classe *Xalet* tinga els mateixos atributs i mètodes que *Casa* més un nou atribut que represente els metres quadrats de jardí.

En compte de tornar a definir una nova classe “des de zero”, pot aprofitar-se el codi escrit per a la classe *Casa* utilitzant l'herència, com ja hem vist.



Com pot vore's, únicament cal fer el següent:

- Indicar que la nova classe *Xalet* és descendent de *Casa* (*extends Casa*)
- Declarar els possibles nous atributs i/o mètodes (*int mJardi*);
- Declarar el constructor de *Xalet*.
 - La primera instrucció serà cridar al constructor de la superclasse, amb la paraula **super**. És a dir: *super(portes, finestres, color)*;
 - Inicialitzar atributs específics del xalet (*mJardi*).

Però els mètodes *pintar*, *obrirFinestres* i *tancarFinestres* no cal definir-los, són heretats de la classe *Casa* i poden utilitzar-se, per exemple de la manera següent:

```
xalet1.pintar("pistatxo");
```

Exercicis

19. Crea la classe *Cotxe2aMa* com a subclasse de la classe *Cotxe*. Tindrà un atribut nou: *kms* (enter), privat.
20. Fes un constructor de la classe *Cotxe2aMa* que agafe com a paràmetres el bastidor, marca, model, matricula i km; que cride al constructor adequat de la superclasse i que inicialitzi la resta d'atributs.
21. Afegix a la classe *Cotxe2aMa* els mètodes *getKms* i *setKms* per a lectura i escriptura del nou atribut (*kms*).
22. Crea l'objecte *cotxeAntic* de la classe *Cotxe2aMa* passant-li al constructor els paràmetres necessaris (inventa-te'ls).
23. Mostra per pantalla totes les dades del *cotxeAntic* usant els mètodes adequats.

Reimplementació dels mètodes de la subclasse

Si volem que un mètode d'una superclasse funcione de forma diferent per a una subclasse, el que hem de fer és tornar a implementar eixe mètode, amb el mateix nom, en la subclasse. Aleshores direm que existeix un polimorfisme (ho vorem al punt següent).

Exemple:

```
public class Persona {
    String nom;
    ...
    public void mostraNom(){
        System.out.println("La persona és: " + this.nom);
    }
}
public class Estudiant extends Persona{
    ...
    public void mostraNom(){
        System.out.println("L'estudiant és: " + this.nom);
    }
}
public class Prova {
    public static void main(String[] args) {
        Persona p = new Persona("Pep");
        Estudiant e = new Estudiant("Pepa");
        p.mostraNom();
        e.mostraNom();
    }
}
```

Invoca el mètode *mostraNom* de la classe *Persona*.
Per tant, mostrarà:
La persona és: Pep

Invoca el mètode *mostraNom* de la classe *Estudiant*.
Per tant, mostrarà:
L'estudiant és: Pepa

Exercicis

24. Reimplementa en la subclasse *cotxe2aMa* el mètode *mostrarDades* de la superclasse perquè mostre totes les dades.
25. Mostra per pantalla totes les dades del *cotxeAntic* usant només el nou mètode *mostrarDades*.

6.4 Polimorfisme

Acabem de veure que pot haver diferents mètodes amb el mateix nom però que fan coses diferents. D'això tracta el polimorfisme.

L'objectiu de la POO és resoldre els problemes com en el món real. En este sentit, el polimorfisme és la forma en que els llenguatges OO implementen la polisèmia:

“Un únic nom per a molts significats, segons el context”.

Tipus de polimorfisme:

- 1) **Sobrecàrrega basada en paràmetres d'entrada:** mètodes amb el mateix nom en una mateixa classe però amb diferents paràmetres d'entrada.

S'executarà un mètode o altre depenent dels paràmetres d'entrada.

- 2) **Sobrecàrrega basada en l'àmbit:** mètodes amb el mateix nom en diferents classes sense relacions d'herència entre ells.

S'executarà un mètode o altre depenent de la classe de l'objecte que fa la crida.

- 3) **Sobreescritura:** mètodes amb el mateix nom i mateixos paràmetres d'entrada en classes distintes però amb relacions d'herència entre ells.

S'executarà un mètode o altre depenent de la classe de l'objecte que fa la crida.

- 4) **Variables polimòrfiques:** variable que pot referenciar distints tipus d'objectes amb relacions d'herència.

Alguns autors solen diferenciar sobrecàrrega de polimorfisme però nosaltres entenem la sobrecàrrega com un tipus de polimorfisme. Atenent a estos quatre tipus, hi ha distintes definicions de polimorfisme:

- És la propietat per la qual es poden enviar missatges iguals a objectes de tipus distints (definició per als 3 primers tipus de polimorfisme).
- És una relaxació del sistema de tipus, de forma que una referència a una classe accepta adreces d'objectes d'eixa classe i de descendents d'ella (definició per al 4t tipus de polimorfisme, relacionat amb el càsting).

Vegem detalladament els 4 tipus de polimorfisme amb l'ajuda d'exemples.

1) Sobrecàrrega basada en paràmetres d'entrada

Exemple 1: Mètodes constructors per a inicialitzar objectes

```
class Casa {  
    ...  
    Casa(int qp, int qf, String col) {  
        this.qPortes=qp  
        this.qFinestres=qf;  
        this.color=col;  
    }  
  
    Casa() {  
        this.qPortes = 0;  
        this.qFinestres = 0;  
        this.color = "";  
    }  
}
```

Una mateixa classe (Casa) té dos mètodes amb el mateix nom però distinta quantitat de paràmetres d'entrada.

En el primer cas s'inicialitzaran els atributs de l'objecte amb els paràmetres del mètode i en el segon cas s'inicialitzaran a un valor inicial per defecte.

En temps de compilació ja se sap el mètode que s'invocarà en cada crida (segons els paràmetres).

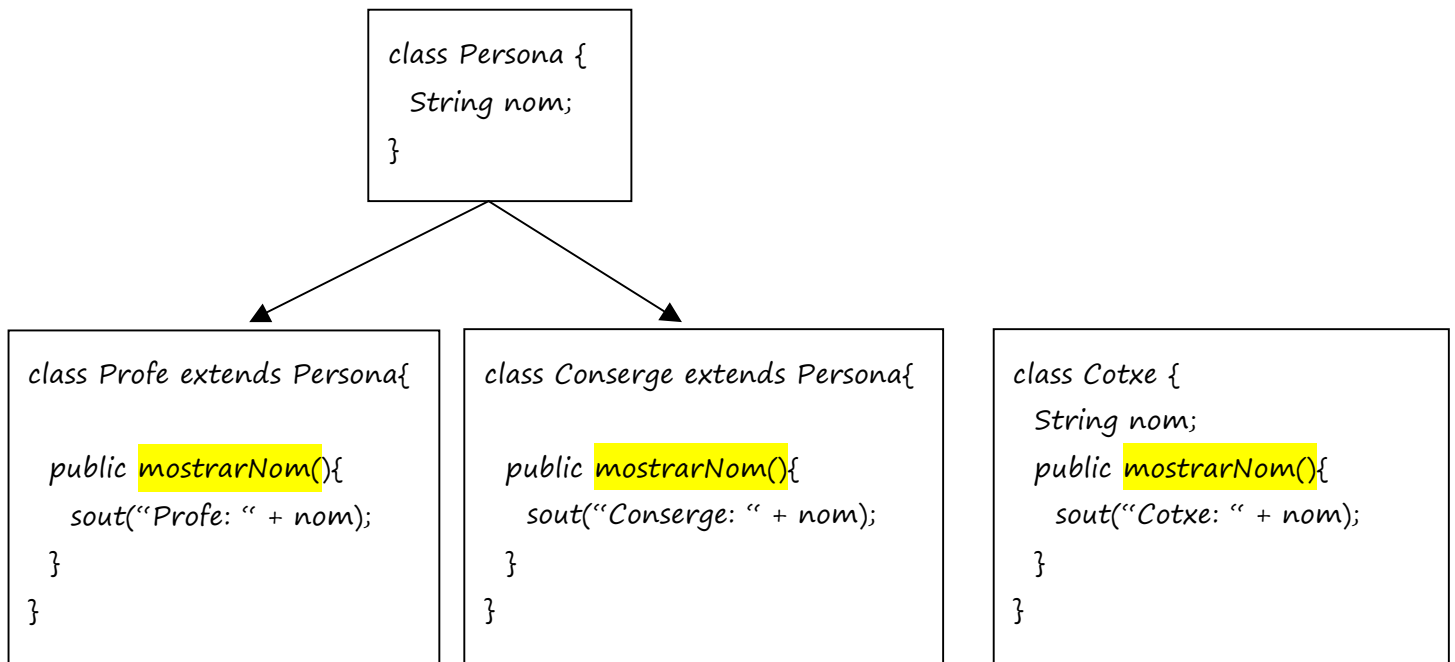
Exemple 2: Altres mètodes per a fer una feina diferent segons els paràmetres:

```
public static void imprimir(int i) {  
    System.out.println(i);  
}  
  
public static void imprimir(int v[]) {  
    for(int i=0;i<v.length;i++){  
        System.out.println(v[i]);  
    }  
}
```

Podem tindre distintes implementacions de la funció per a diferents tipus dels paràmetres d'entrada (no importa els paràmetres d'eixida).

El primer mètode s'executarà quan li passem un enter. El segon s'executarà quan li passem un vector d'enters.

2) Sobrecàrrega basada en l'àmbit



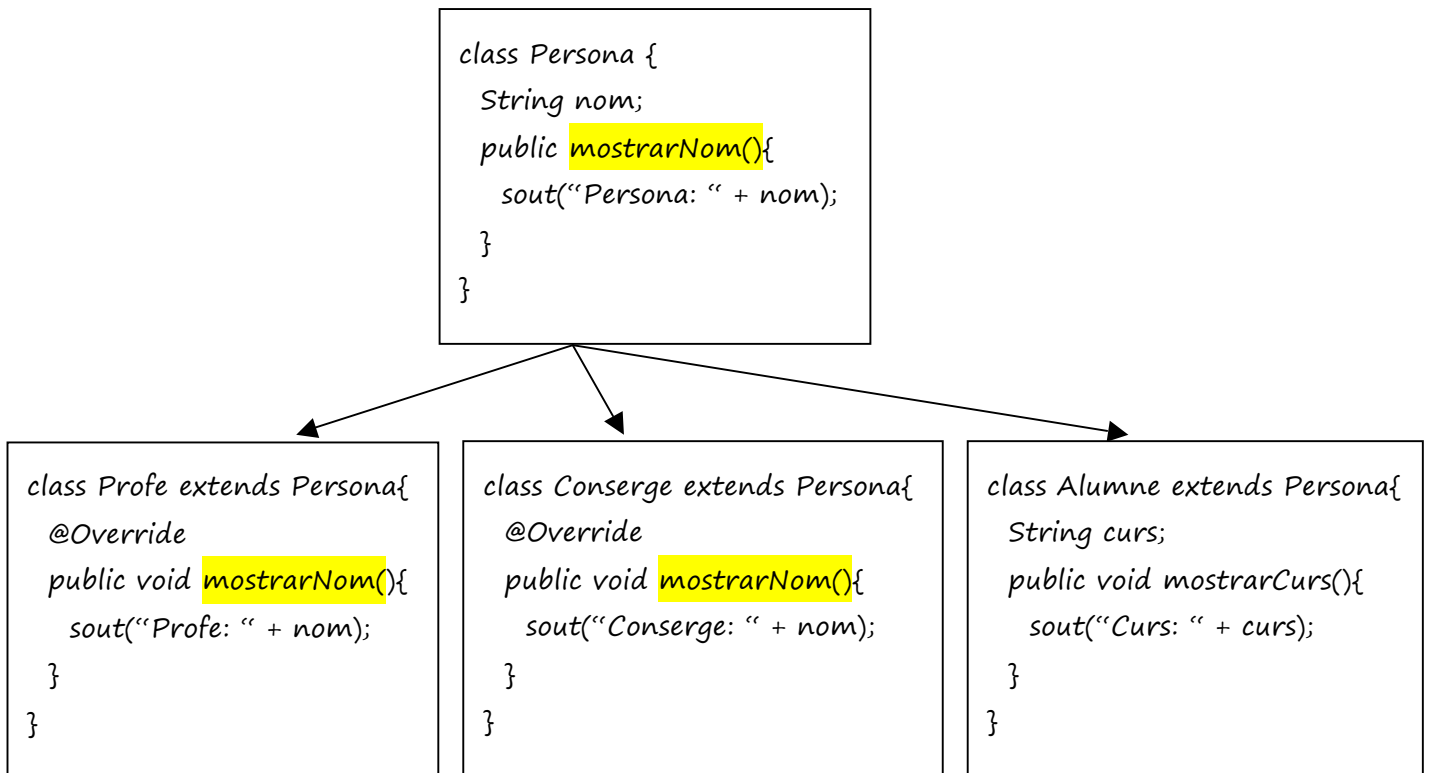
El mètode `mostrarNom()` està sobrecarregat en base a l'àmbit en les classes `Profe`, `Conserge` i `Cotxe`. El `mostrarNom()` de `Profe` i el de `Conserge` no tenen res a veure, ja que no estan sobreescrivint el mètode en la classe pare en comú.

```
Persona per = new Persona();    per.mostrarNom(); // Error compilació
Profe pro   = new Profe();      pro.mostrarNom(); // Profe: Pep
Conserge con = new Conserge();  con.mostrarNom(); // Conserge: Pep
Cotxe cot   = new Cotxe();      cot.mostrarNom(); // Cotxe: Ford
```

No hi ha conflicte. En cadascuna de les crides anteriors cridarà al `getNom()` corresponent de la classe de cada objecte. En temps de compilació ja se sap quin mètode s'invocarà.

3) Sobreescritura

La sobreescritura està lligada a l'herència entre classes.



Veiem que en distintes classes però relacionades per l'herència tenim diferents implementacions del mètode `mostrarNom()` amb els mateixos paràmetres (cap). Quan s'invoque eixe mètode s'executarà el de la classe de l'objecte que ha fet la crida:

```
Persona per = new Persona();    per.mostrarNom();    // Persona: Pep
Profe pro   = new Profe();      pro.mostrarNom();    // Profe: Pep
Conserge con = new Conserge();  con.mostrarNom();    // Conserge: Pep
Alumne alu  = new Alumne();     alu.mostrarNom();    // Persona: Pep
(new Profe()).mostrarNom();      // Profe: Pep
```

Se sol dir que la sobrescriptura és el polimorfisme pròpiament dit.

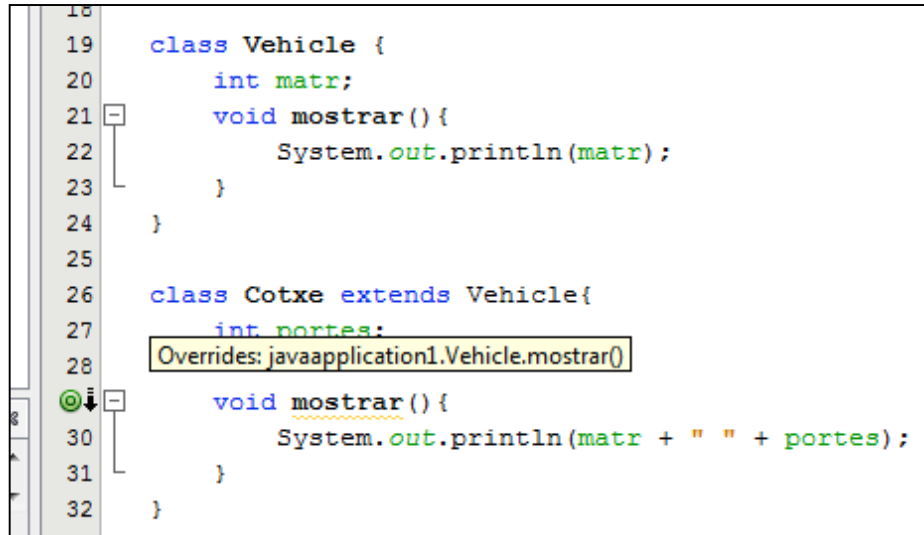
Veiem en l'exemple anterior que dalt dels mètodes sobreescrits apareix la paraula `@Override`. A continuació s'explicarà què és i per a què serveix.

Què és el @Override?

Serveix per a indicar que estem sobrecarregant un mètode. Si no ho posem no passa res, però Netbeans ens avisa:

Overrides: NomPrograma.ClassePare.metodeSobreescrit();

Per exemple:



```
18
19 class Vehicle {
20     int matr;
21     void mostrar() {
22         System.out.println(matr);
23     }
24 }
25
26 class Cotxe extends Vehicle{
27     int portes;
28     Overrides: javaapplication1.Vehicle.mostrar()
29     void mostrar() {
30         System.out.println(matr + " " + portes);
31     }
32 }
```

No és un error. Simplement ens avisa que el mètode mostrar de la classe Cotxe ja l'hem definit en la classe pare i, per tant, l'oculta.

Independentment de Netbeans, nosaltres podem indicar que està sobreescrit posant @Override just abans de la definició del mètode sobreescrit:

```
@Override
void mostrar(){
    System.out.println(matr + " " + portes);
}
```

El funcionament serà el mateix. Per tant, per a què serveix?

- ✓ Indica al programador que eixe mètode "substitueix" l'altre en el fill.
- ✓ Si volem sobreescriure un mètode i posem el @Override però ens hem equivocat amb el nom del mètode en alguna lletra, ens avisaria, ja que no estarem sobrecarregant res.
- ✓ Si més endavant eliminàrem el mètode mostrar() de la classe pare o li canviàrem el nom, el compilador avisaria, ja que el mètode corresponent en la classe filla deixaria d'estar sobreescrit.

4) Variables polimòrfiques

La potència del polimorfisme en Java està en que és possible **definir** un objecte d'una classe pare i **instanciar-lo** amb una classe filla (o descendent):

```
ClassePare obj = new ClasseFilla();
```

Algunes utilitats de les variables polimòrfiques:

a) Tindre una llista d'objectes de diferents classes (que hereten d'un mateix pare)

Suposem que volem tindre un array de persones (classe pare) on pugam guardar tant alumnes com professors o conserges (classes filles).

```
Persona [] llista = new Persona[N];  
llista[0] = new Profe();  
llista[1] = new Alumne();  
llista[2] = new Persona();  
llista[0].mostrarNom(); // Invoca el mostrarNom() de Profe  
llistas[1].mostrarNom(); // Invoca el mostrarNom() de Persona
```

```
llista[1].mostrarCurs();
```

Error de compilació ja que el mètode `mostrarCurs` no està en la classe `Persona` sinó en una classe filla (`Alumne`).

La solució és fer un **càsting** de l'objecte. És a dir: dir-li al compilador que agafe les característiques d'alumne:

```
( (Alumne)llista[1] ).mostrarCurs(); // Invoca el mostrarCurs() d'Alumne
```

Ara bé, si fem:

```
( (Alumne)llista[2] ).mostrarCurs();
```

Donarà **error d'execució** perquè no pot fer el càsting, ja que en la posició 2 hi ha una `Persona`, no un `Alumne`. I només podem fer càsting a una classe ascendent.

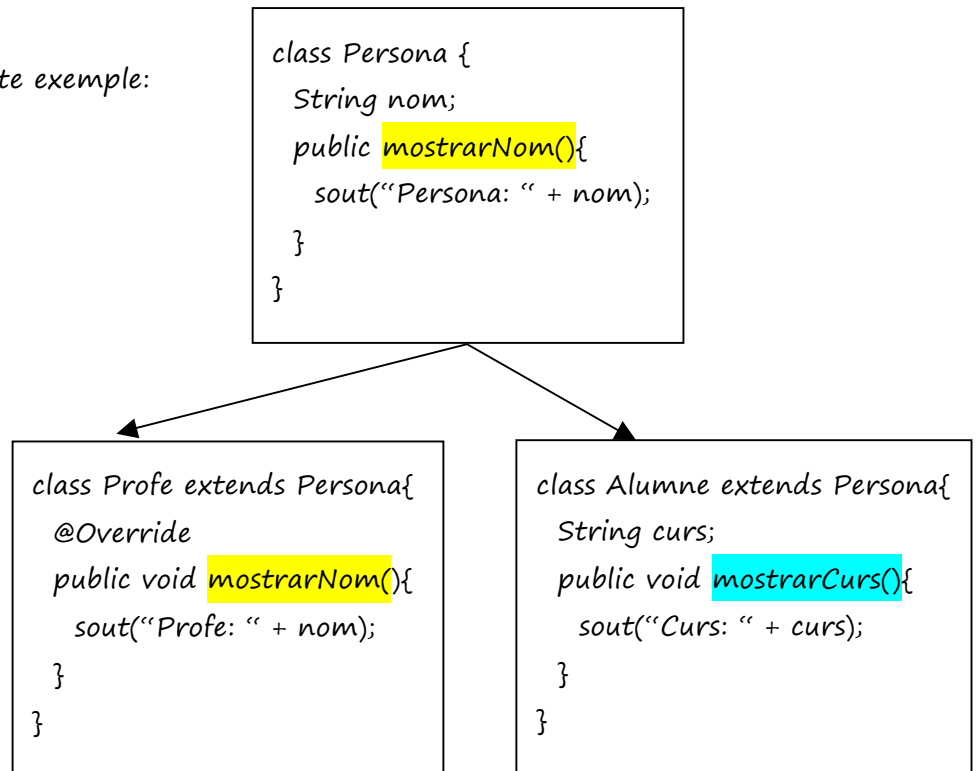
b) Definir una funció que reba com a paràmetre un objecte pare per a poder acceptar objectes de diferents fills.

```
static void mostrar( Persona p ){ // 'p' accepta profes, alumnes i conserges  
    System.out.println("-----");  
    p.mostrarNom();  
    System.out.println("-----");  
}
```

Accés als mètodes i atributs d'una variable polimòrfica:

En les variables polimòrfiques, en la compilació es miren els mètodes i atributs de la classe pare però en l'execució es miren els de la classe filla. Per tant, segons la classe on estiguen definits els mètodes i atributs, i segons càsting que fem, tindrem uns resultats o altres.

Mirem-ho amb este exemple:



```
Persona pro = new Profe();
pro.nom = "Pep";
pro.mostrarNom();
```

Si el mètode està en la classe pare i filla, invoca el de la classe filla.

Profe: Pep

```
Persona alu = new Alumne();
alu.nom = "Pep";
alu.mostrarNom();
```

Si el mètode només està en la classe pare, invoca el de la classe pare (ja que l'hereta).

Persona: Pep

```
alu.curs = "1DAM";
alu.mostrarCurs();
((Alumne) alu).curs = "1DAM";
((Alumne) alu).mostrarCurs();
```

Si l'atribut o el mètode només està en la classe filla, dóna **error de compilació**.

Solució: fer el càsting.

Curs: 1DAM

```
Cotxe cotxe1 = new Cotxe();
((Alumne) cotxe1).mostrarCurs();
```

Però este càsting donaria **error d'execució**, ja que un objecte només pot fer càsting a una classe ascendent (mare, iaia...)

Esquema de les característiques dels distints tipus de polimorfisme:

POLIMORFISME						
TIPUS	DE MÈTODE O DE VARIABLE	ON ESTÀ EL POLIMORFISME?	PODEN TINDRE ELS MATEIXOS PARÀMETRES D'ENTRADA?	USOS FREQUENTS	QUAN S'INVOCA UN MÈTODE, QUIN S'EXECUTA?	SE SAP A QUI INVOCA EN TEMPS DE...
Sobrecàrrega basada en els paràmetres d'entrada	Mètode	En la mateixa classe	NO	<ul style="list-style-type: none"> - Classe amb més d'un constructor - Mètode que accepta diferents tipus de paràmetres 	El que concorda amb els paràmetres d'entrada	Compilació
Sobrecàrrega basada en l'àmbit	Mètode	En classes distintes, sense relació d'herència	SÍ	<ul style="list-style-type: none"> - Mètodes amb un nom comú però que tenen poc a veure 	El de la classe de l'objecte que fa la crida	Compilació
Sobreescritura (Override)	Mètode	En classes distintes, amb relació d'herència	SÍ	<ul style="list-style-type: none"> - Subclasses que volen modificar el comportament d'un mètode de la superclasse 	El de la classe de l'objecte que fa la crida. Si no existeix, el de la superclasse.	Compilació
Variables polimòrfiques	Variable	En classes distintes, amb relació d'herència entre la classe de la referència (pare) i la classe de l'objecte instanciat (filla)	-	<ul style="list-style-type: none"> - Una llista que accepta objectes de distintes subclasses - Un mètode que accepti objectes de distintes subclasses 	<ul style="list-style-type: none"> - Si el mètode (o atribut) està en pare i fill, agafa el del fill. - Si només en pare, agafa el del pare. - Si només en fill, cal fer el càsting per a agafar-lo (si no, error de compilació). 	Execució

Exercicis

26. El mètode `mostrarDades` té polimorfisme, ja que està implementat en distintes classes (`Cotxe` i `Cotxe2aMa`). Modifica el mètode `mostrarDades` de la subclasse: fes ús de “`super`” per a accedir al `mostrarDades` de la superclasse.
27. Anem a sobrecarregar el mètode `revisar` en la classe `Cotxe`. Crea en la classe `Cotxe` altre mètode `revisar` però que reba un vector de 5 booleans. El mètode haurà de copiar les components del vector d'entrada en les components de l'atribut `revisions`.
28. En el programa principal:
- Crea un cotxe i fes diverses crides al mètode `revisar` de forma que s'executen les 2 implementacions d'eixe mètode.
 - Crea un array de cotxes i posa en ell cotxes nous i de 2a mà. Recorre tot l'array per a cridar al `mostrarDades` de cada objecte de la llista.
 - Dins la classe principal (la que té el `main()`) crea el mètode `mostrarKms` de forma que accepti com a paràmetre un cotxe de segona mà o normal (per tant, el paràmetre serà de la classe pare: `Cotxe`) i que mostri per pantalla la quantitat de kms que té.

Problema: Voràs que et dóna error al compilar ja que la classe `Cotxe` té l'atribut dels kms.

Solució: Fes un càsting de l'objecte `cotxe` per a accedir als kms.

- Crida al mètode anterior passant-li com a paràmetre un objecte de la classe `Cotxe2aMa` i una altra crida passant-li un objecte de la classe.

Problema: Voràs que dóna un error d'execució en el moment de passar-li com a paràmetre un objecte de la classe `Cotxe`. El motiu és que ja que no pot fer el càsting a `Cotxe2aMa`, ja que només podem fer càsting a una classe ascendent.

Solució: Modifica el mètode `mostrarKms` per a que si se el paràmetre és una instància de `Cotxe2aMa`, que mostri els kms; si no, que mostri el text “Cotxe nou, amb 0 kms”. Prova diverses solucions:

- ✓ Amb un `try-catch`
- ✓ Amb el mètode `getClass`:
 `if (cotxe.getClass().getSimpleName().equals("Cotxe2aMa")) { ... }`
- ✓ Amb l'operador `instanceof`:
 `if (cotxe instanceof Cotxe2aMa) { ... }`

7 Atributs i mètodes estàtics

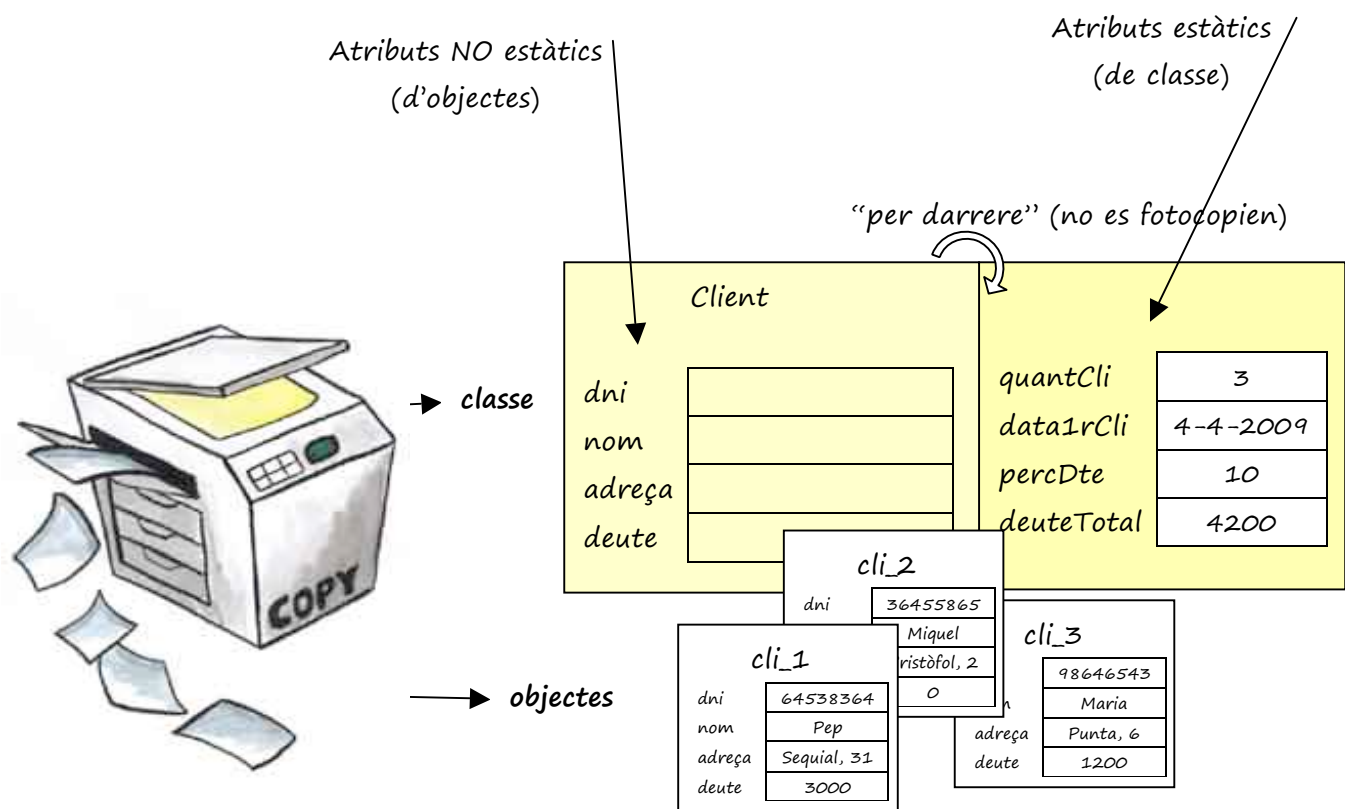
7.1 Atributs estàtics

En les classes, com ja sabem, podem definir atributs. Eixos atributs poden ser:

- Atributs d'objecte. Són els que hem vist fins ara. Per cada objecte que definim d'eixa classe, tindrà el corresponent valor per a eixe atribut. No podem assignar un valor a eixos atributs **en la classe**.
- Atributs de classe. Tindrem un únic valor de l'atribut per a tota la classe, no per a cada objecte. Estos atributs s'anomenen **estàtics**. Sí que podem assignar un valor a eixos atributs **en la classe**.

És a dir, els atributs estàtics són propis únicament de la classe i no dels objectes que puguem crear-se de la classe. Per tant, donarem valor a eixos atributs en la classe.

Si recordem l'analogia de classe i objectes amb el full original i les fotocòpies, ara podem dir que els atributs estàtics estarien a la part de darrere del full original i, per tant, no es fotocopien. És a dir, pertanyen només a la classe, no als objectes.



Exemple d'atribut estàtic: la quantitat de fotocòpies que es creen d'eixe full (quantitat d'objectes creats a partir d'eixa classe). En el nostre cas: quantitat de clients.

Declaració

Els atributs estàtics van precedits pel modificador *static*.

Sintaxi:

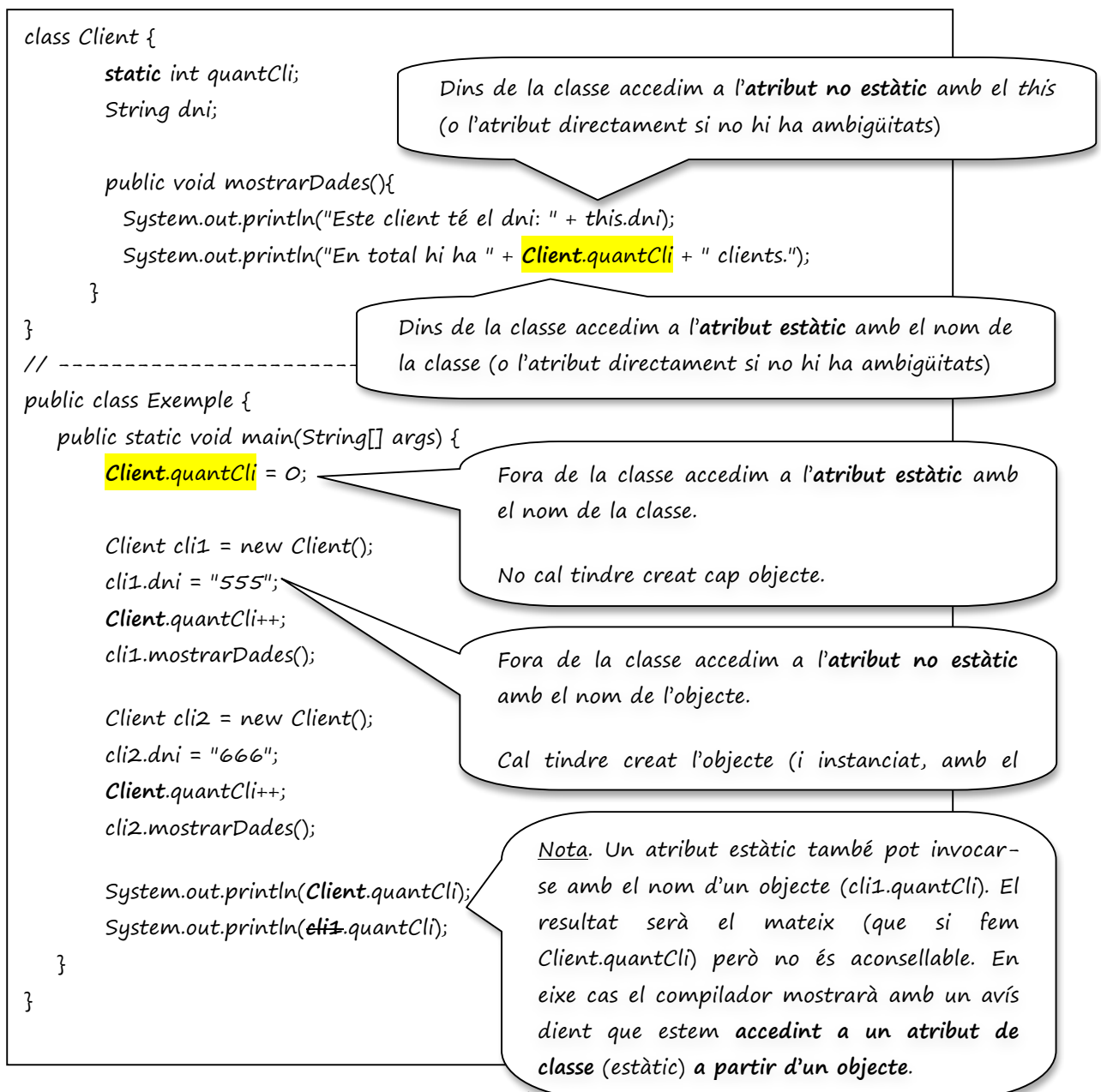
```
class Xxx {  
    static int atributEstatic;  
    int atributNoEstatic;  
    ...  
}
```

Exemple:

```
class Client {  
    static int quantCli;  
    String dni;  
    ...  
}
```

Invocació

Per a usar un atribut estàtic (de classe) posarem *nomClasse.nomAtribut* (en compte de ~~nomObjecte~~.nomAtribut). Vegem un exemple:



Utilitats dels atributs estàtics

Els atributs estàtics solen utilitzar-se per a:

- ✓ Definir constants
- ✓ Definir variables amb el mateix valor per a tots els objectes de la classe.
- ✓ Fer un comptador d'objectes de la classe
- ✓ Etc

Exercicis

29. Afegir a la classe *Cotxe* l'atribut estàtic *quantitat* (enter), on guardarem la quantitat de cotxes que tenim (quantitat d'objectes creats de la classe *Cotxe*).
30. Modificar la classe *Cotxe* per a que s'actualitzi el nou atribut *quantitat* quan calgui. Com que és un comptador, caldrà inicialitzar-lo i incrementar-lo.
31. Fes proves creant diversos cotxes, també de segona mà. Posteriorment mostra el valor de l'atribut *quantitat* per comprovar que funciona correctament.

7.2 Mètodes estàtics

Són mètodes que poden cridar-se sense invocar a cap objecte.

És a dir, els mètodes declarats com a *static* pertanyen a tota la classe, i no a cap objecte en particular d'eixa classe.

Declaració

Els mètodes estàtics (o de classe) es distingixen dels mètodes d'instància (o d'objecte) perquè porten el modificador *static* davant de la definició del mètode.

```
class Client {  
    // --- Atributs ---  
    private static int quantCli;  
    String dni;  
    ...  
    // --- Mètodes ---  
    ...  
    static int quantsClients(){  
        return this.quantCli;  
    }  
    String getDni(){  
        return this.dni;  
    }  
}
```

Este mètode s'aplica a tots els clients en general.
Per a accedir a l'atribut estàtic ho farem amb Client.quantCli, o bé, com estem dins de la mateixa classe, simplement, quantCli.

Este mètode s'aplica a un client en concret.
Per a accedir a l'atribut NO estàtic, ho farem amb this.dni, o bé, simplement amb dni (ja que no hi ha conflicto de noms).

Invocació

Per a cridar a un mètode *static* farem:

nomClasse.nomMètodeEstàtic(paràmetres);

Nom de la classe (no de l'objecte)

Mètode estàtic

O bé, si el mètode està dins la classe on fem la crida, podem no posar el nom de la classe. Per tant, simplement farem:

nomMètodeEstàtic(paràmetres);

Exemple:

```
public class Principal {  
    public static void main (String[] args) {  
        ...  
        Client c3 = new Client();  
        ...  
        int n = Client.quantsClients();  
        ...  
        String d = c3.getDni();  
        ...  
    }  
}
```

Cridem a un mètode estàtic amb el nom de la classe.

Cridem a un mètode NO estàtic amb el nom d'un objecte.

Utilitats dels mètodes estàtics

Sense saber-ho, fins ara hem fet ja crides a mètodes static. Per exemple:

```
float f = areaTriangle(10, 5);    // Càlcul àrea d'un triangle  
System.out.println("Hola, món!"); // Imprimir un text en pantalla  
imprimir(vectorNotes);           // Imprimir un vector d'enters  
int i = Integer.parseInt("10");   // Convertir una cadena a un enter  
int j = llegirEnter();            // Llegir un enter de teclat
```

És a dir, els mètodes estàtics no treballen amb objectes, sinó amb:

- ✓ Paràmetres d'entrada
- ✓ Atributs estàtics (de classe)

Exercicis

32. Fes que l'atribut estàtic *quantitat* que has creat en la classe *Cotxe* siga privat. Per tant, caldrà un mètode per a consultar el seu valor: crea el mètode *getQuantitat* que retorne la quantitat de cotxes creats. Tin en compte que ha de ser un mètode estàtic, ja que no s'aplica a cap objecte en concret. Finalment, mostra la quantitat de cotxes creats, usant el nou mètode *getQuantitat*.

8 Miscel·lània

8.1 Ús de this. i super.

Fins ara hem dit que, si en un mètode d'una classe volem accedir als atributs membre, cal usar el "this." davant de l'atribut membre. Realment no cal si no hi ha conflicte de noms amb els paràmetres del mètode (o amb variables locals seues).

Ús obligatori del "this.":

```
class Alumne {  
    String nom;  
    void setNom(String nom){  
        this.nom = nom;  
    }  
}
```

Ús opcional del "this.":

```
class Alumne {  
    String nom;  
    void setNom(String n){  
        nom = n;  
    }  
}
```

El this és una referència per a accedir a l'objecte des del qual es crida al mètode. Per això, si el mètode és estàtic donarà error. També si l'atribut membre és estàtic.

Des dels mètodes (no estàtics) d'una classe podem accedir a les variables membre i als mètodes de la classe amb el "this." davant. I, per a accedir a les variables membre i als mètodes de la classe pare usarem el "super." davant.

Igual que el this només és necessari si hi ha conflicte de noms, el super també. És a dir: si des d'una classe filla volem accedir a una mètode o variable membre de la classe pare que també està declarat en la classe filla, haurem de posar-li el "super." davant.

Exercicis

33. Lleva el `this` en els exercicis que has fet i comprova si va igual. Prova-ho també fent que els noms dels paràmetres es diguen igual que les variables membre.

34. Crea un projecte nou i crea estes classes:

```
class Vehicle {
    String matr;
    int any;
    public void mostrarDades(){
        System.out.println(this.matr);
        System.out.println(this.any);
    }
}

class Cotxe extends Vehicle {
    int portes;
    int any; // Este any és diferent de l'any del vehicle
    public void mostrarDades(){ // Este mostrarDades és diferent del del vehicle
        super.mostrarDades();
        System.out.println(this.portes);
        System.out.println(this.any);
    }
    public void proves(){
        System.out.println("matr = " + matr);
        System.out.println("this.matr = " + this.matr);
        System.out.println("super.matr = " + super.matr);
        System.out.println("any = " + any);
        System.out.println("this.any = " + this.any);
        System.out.println("super.any = " + super.any);
        System.out.println("mostrarDades():"); mostrarDades();
        System.out.println("this.mostrarDades():"); this.mostrarDades();
        System.out.println("super.mostrarDades():"); super.mostrarDades();    }
}
```

Què creus que mostraria el mètode `proves()`? Per a comprovar-ho, des del main, crea un objecte de la classe `Cotxe`, dóna-li valor a la matrícula, any i portes i fes una crida al mètode `proves()` d'eixe objecte. Diques si és V o F:

- Si no hi ha conflicte → Per a accedir al pare puc posar this, super o res.
- Si hi ha conflicte → this agafa el fill ; super el pare ; “res” el fill.
- Si una variable està en pare i fill, cadascuna pot guardar un valor distint.

8.2 Paràmetres per referència

Recordem que el pas de paràmetres a una funció pot ser:

- **Per valor:** qui crida a la funció li passa en el paràmetre una còpia del valor. Per tant, si la funció modifica eixe paràmetre, en tornar de la funció no s'haurà modificat el valor que tenia abans.
- **Per referència:** qui crida a la funció li passa la direcció del paràmetre, de forma que si la funció modifica eixe paràmetre, en tornar de la funció sí que s'haurà modificat el valor que tenia abans.

En Java només es poden passar per referència els vectors, matrius i objectes. És a dir, Java no permet passar per referència una variable simple (un enter, per exemple). Però feta la llei, feta la trampa. Podríem fer-ho posant eixe enter dins d'un objecte.

Per exemple, si volem passar a una funció la variable entera `edatMeua` per a que l'ompliga, podríem definir eixa variable com un objecte en compte de com un enter. És a dir:

```
class Edat {
    int anys;
}

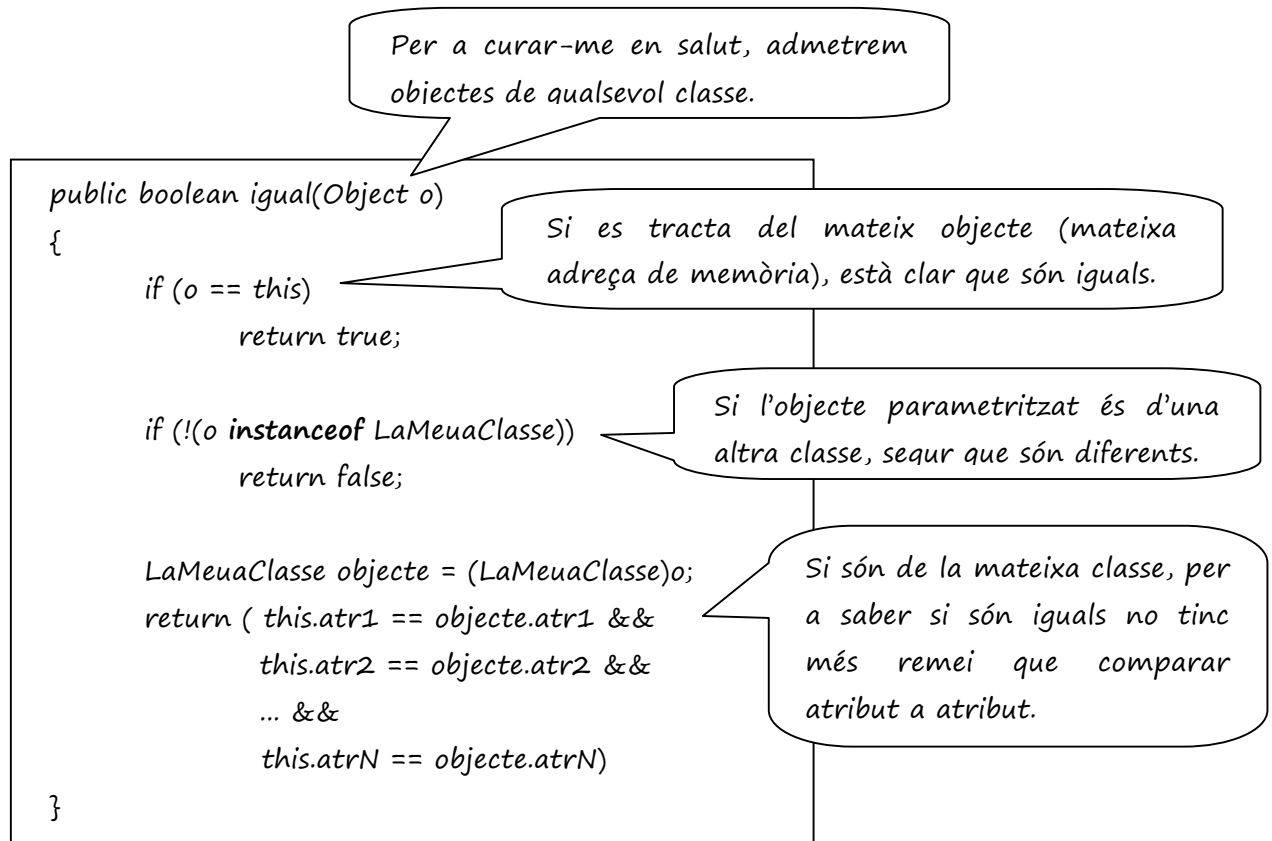
class exemple_variables_per_referencia{
    public static void main( String args[] ) {
        Edat edatMeua = new Edat();
        demanarEdat(edatMeua);
        System.out.println("La meua edat és: " + edatMeua.anys);
    }

    // Funció que rebrà l'objecte (pas de variables per referència)
    static void demanarEdat( Edat edat ) {
        System.out.println( "Quina edat tens?" );
        edat.anys = llegirEnter();
    }
}
```

8.3 Comprovar si dos objectes són iguals

Com ja diguérem al tema anterior, per a saber si dos objectes són iguals, no s'ha de fer comparant els objectes directament amb `==`, ja que això comprova que realment siga el mateix objecte, és a dir, que els dos apunten a la mateixa adreça de memòria.

Si el que volem és saber si dos objectes guarden la mateixa informació, dins la classe hauríem de fer-nos un mètode com este:



Si volguérem comparar si els objectes `o1` i `o2` són iguals, la crida seria:

```
if ( o1.igual(o2) ) { ... }
```

Ara bé, la classe `Object` ja té el mètode `"equals"`. Per tant, en compte de definir per a cada classe un mètode que es diga `"igual"`, caldrà redefinir el mètode `"equals"` per a cada classe que necessitem. Entre altres coses perquè altres mètodes ja predefinits en Java (com el `contains` o `indexOf` de la classe `ArrayList`) faran ús del mètode `equals` de cada objecte.

9. La classe ArrayList i la classe Vector

Estes classes permeten guardar dades en memòria de manera similar als arrays convencionals, però amb els avantatges següents:

- No s'indica la grandària sinó que s'assigna de forma dinàmica.
- Disposen d'un conjunt de mètodes que permeten consultar, eliminar, introduir elements, etc de forma automàtica.

Per a usar eixes classes cal importar els paquets:

```
java.util.ArrayList  
java.util.Vector
```

Les dos classes fan el mateix (tenen pràcticament els mateixos mètodes). Per tant, explicarem més en detall una d'elles: els ArrayList.

Només es diferencien en qüestions d'eficiència pel que fa al temps d'execució i la gestió de l'ocupació de la memòria. Explicarem millor estes diferències quan veiem la classe Vector.

9.1 La classe ArrayList

Declaració d'un objecte ArrayList

De dos formes possibles:

a) Indicant el tipus de dades dels elements que guardarà l'ArrayList:

ArrayList <nomClasse> nomLlista;

- ✓ Podrà guardar elements d'eixa classe o d'una classe "filla".
- ✓ Si se sap la classe dels elements que guardarà l'ArrayList, és recomanable esta forma de declarar-lo.
- ✓ En cas de guardar dades d'un tipus bàsic de Java com char, int, double, etc, s'ha d'especificar el nom de la classe associada: Character, Integer, Double, etc.
- ✓ Exemples:

```
ArrayList <String> llistaPaisos;  
ArrayList <Integer> edats;  
ArrayList <Alumne> alumnes;
```

b) Sense indicar el tipus de dades dels elements que guardarà l'ArrayList:

ArrayList nomLlista;

- ✓ Podrà guardar elements de qualsevol classe.
- ✓ Com vorem més avant, els mètodes d'ArrayList que retornen un element de la llista el retornaran com a instància de la classe Object i caldria fer el càsting si volguérem accedir a les seues propietats
- ✓ Exemple:

```
ArrayList llistaCoses;
```

Creació d'un ArrayList (que ja està declarat)

nomLlista = new ArrayList();

- ✓ Però, com sol ser habitual, es pot crear la llista al mateix temps que es declara:

ArrayList <nomClasse> nomLlista = new ArrayList();

- ✓ Exemples:

```
ArrayList <String> llistaPaisos = new ArrayList();  
ArrayList <Alumne> alumnes = new ArrayList();  
ArrayList llistaCoses = new ArrayList();
```

Afegir elements al final de la llista

```
boolean add (Object elementAInserir);
```

El mètode *add* de la classe *ArrayList* permet afegir elements (després de l'últim element que hi haguera a l'*ArrayList*). El primer element es posa en la posició 0.

Exemples:

```
ArrayList <String> llistaPaisos = new ArrayList ();  
llistaPaisos.add ("Alemanya");      // Ocupa la posició 0  
llistaPaisos.add ("França");      // Ocupa la posició 1  
llistaPaisos.add ("Portugal");     // Ocupa la posició 2
```

```
ArrayList <Integer> edats = new ArrayList ();  
edats.add (22);  
int edat = 18;  
edats.add (edat);
```

```
ArrayList <Cotxe> llistaCotxes = new ArrayList();  
Cotxe cotxe1 = new Cotxe("Ford", "Festa", "V-3316-FP");  
llistaCotxes.add( cotxe1 );  
llistaCotxes.add( new Cotxe("Seat", "Panda", "V-1023-AA") );
```

```
ArrayList llistaCoses = new ArrayList();  
llistaCoses.add("Alemanya");  
llistaCoses.add(22);  
llistaCoses.add( new Cotxe("Seat", "Panda", "V-1023-AA");
```

Inserir elements en una determinada posició

```
void add (int posició, Object elementAInserir);
```

Inserix un element en la posició indicada, desplaçant l'element que es trobava en aquesta posició, i tots els següents, una posició més.

Exemple:

```
ArrayList <String> llistaPaisos = new ArrayList ();  
llistaPaisos.add ("Alemanya");    // Posició 0  
llistaPaisos.add ("França");    // Posició 1  
llistaPaisos.add ("Portugal");   // Posició 2  
// L'ordre fins ara és: Alemanya, França, Portugal  
llistaPaisos.add (1, "Itàlia");  
// L'ordre ara és: Alemanya, Itàlia, França, Portugal
```

Si s'intenta inserir en una posició que no existeix, es produeix l'excepció `IndexOutOfBoundsException`.

Canviar un element de la llista

```
Object set (int posició, Object elementQueSubstituirà);
```

Servix per a canviar un element que està en la llista. El 1r paràmetre és la posició que ocupa l'element a modificar, i el 2n és el nou element que substituirà l'antic. Retorna l'objecte substituït (el podrem arrebregar, si volem).

Exemple:

Si en la llista de països volem substituir el país de la posició 1 per Alemanya, caldrà fer:

```
llistaPaisos.set (1, "Andorra");
```

Però si, a més, volíem conservar l'element substituït en alguna variable, farem:

```
String paisAnterior = llistaPaisos.set (1, "Andorra");
```

Suprimir elements de la llista

De 2 formes:

- Indicant la posició (retorna l'objecte suprimit):

`Object remove (int posició)`

- Indicant l'element a esborrar (retorna false si no estava):

`boolean remove (Object elementASuprimir)`

Exemple amb Strings:

```
// L'ordre ara és: Alemanya, Itàlia, França, Portugal
String paisEsberrat = llistaPaisos.remove(2);
// Eliminada França, queda: Alemaya, Itàlia, Portugal
llistaPaisos.remove("Portugal");
// Eliminada Portugal, queda: Alemanya, Itàlia
```

Exemple amb objectes:

```
ArrayList <Cotxe> llistaCotxes = new ArrayList();
...
Cotxe primerCotxe = llistaCotxes.remove(0);
```

Hem eliminat el cotxe de la primera posició de la llista, però no perdem la referència a ell, ja que l'hem guardada en l'objecte "primerCotxe". És a dir: el cotxe en sí no s'elimina, sinó que ara ja no està en la llista.

Nota: si en un ArrayList s'ha especificat la classe dels seus elements, els mètodes de la classe ArrayList que retornen un element de la llista no retornaran un element de la classe Object sinó de la classe especificada en la declaració de l'ArrayList. Això passa, per exemple, en els mètodes set, remove, get, clone, etc. En el mètode que anem a veure a continuació (get) s'explica detalladament.

Consulta d'un determinat element de la llista

Object get (int posició)

Retorna l'element guardat en una determinada posició de l'ArrayList.

Amb l'element obtingut es podrà realitzar qualsevol de les operacions possibles segons el tipus de dada de l'element (assignar l'element a una variable, incloure'l en una expressió, mostrar per pantalla, etc).

Exemple:

```
System.out.println( llistaPaisos.get(3) );  
// Seguint l'exemple anterior, mostraria: Portugal
```

Exemple per veure la diferència en la declaració de l'ArrayList:

<pre>ArrayList llistaCotxes = new ArrayList(); ... Cotxe c = (Cotxe) llistaCotxes.get(0);</pre>	<pre>ArrayList <Cotxe> llistaCotxes = new ArrayList(); ... Cotxe c = llistaCotxes.get(0);</pre>
<p>Si declarem un ArrayList sense indicar la classe, l'objecte retornat pel mètode serà de la classe Object i, per tant, caldrà fer el càsting.</p>	<p>Si declarem l'ArrayList indicant la classe, l'objecte retornat pel mètode serà de la classe especificada i, per tant, no caldrà fer el càsting.</p>

Buscar un element

```
int indexOf (Object elementBuscat)
```

Retorna la posició que ocupa l'element que s'indique per paràmetre.

Si no està, retorna -1.

I si està en més d'una posició, retorna la primera posició que trobe. Si volguérem l'última posició caldria usar el mètode:

```
int lastIndexOf (Object elementBuscat)
```

Exemple que comprova si França està a la llista, i mostra la seva posició:

```
...
String paisBuscat = "França";
int pos = llistaPaisos.indexOf(paisBuscat);
if (pos != -1)
    System.out.println (paisBuscat + " s'ha trobat a la posició: " + pos);
else
    System.out.println (paisBuscat + " no s'ha trobat");
```

Recórrer el contingut de la llista

És possible obtenir cada un dels elements de la llista utilitzant un bucle amb tantes iteracions com elements continga, de forma similar a l'empleada amb els arrays convencionals. Per obtenir la quantitat d'elements d'un *ArrayList* s'usa el mètode *size()*.

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

També podem usar l'altra sintaxi del *for*, en la qual es va assignant cada element de la llista a una variable del mateix tipus que els elements de l'*ArrayList*:

```
for (String cotxe: cotxes) {
    System.out.println( cotxe.toString() );
}
```

Altres mètodes d'interès

void clear(): Esborra tot el contingut de la llista.

Exemple: `llistaCotxes.clear();`

boolean contains(Object element): Retorna true si es troba l'element indicat a la llista, i false en cas contrari.

Exemple: `if (llistaCotxes.contains(cotxeMeu)) { sout("Sí que està"); }`

boolean isEmpty(): Retorna true si la llista és buida.

Exemple: `if (llistaCotxes.isEmpty()) { sout("Llista buida"); }`

Object clone(): Retorna una còpia exacta de la llista (com a ArrayList). Però els elements no són copiats (les respectives posicions apunten als mateixos objectes). Com el clone() retorna un Object, cal fer el càsting:

Exemple: `llistaCotxes2 = (ArrayList<Cotxe>)llistaCotxes1.clone();`

Object [] toArray(): Retorna una còpia de la llista com a array d'Objects. En este cas, cadascun dels elements sí que són copiats (no es copien les referències). El toArray retorna un vector d'Objects però no es pot fer el càsting directament a vector de la classe que volem, sinó que cal fer el càsting element a element del vector:

Exemple: `Object [] vectorCotxes = llistaCotxes.toArray();
for (Object obj: vectorCotxes){
 ((Cotxe)obj).mostrarNom();
}`

Exercicis

35. Exercici sobre ArrayList de Strings.

- a. Crea un ArrayList de Strings anomenat pobles.
- b. Afig a la llista 4 pobles: Tavernes, Sueca, Sollana Cullera.
- c. Afig en la primera posició: Gandia
- d. Canvia el poble de la posició número 2 per Cullera (un altre Cullera al que ja hi havia) i guarda el poble que s'ha canviat en la variable pobleCanviat.
- e. Esborra el poble de la posició 3 de la llista i guarda el poble que s'ha esborrat en la variable pobleEsborrat.
- f. Esborra el poble Sueca (no sabem en quina posició està).
- g. Mostra per pantalla el poble de la posició 2.
- h. Mostra per pantalla la primera posició de Cullera i l'última.
- i. Mostra tots els pobles de la llista (cadascun en una línia).
- j. Mostra per pantalla si la llista està buida o si no.

36. Exercici sobre ArrayList d'objectes.

Volem guardar les dades de cada alumne i les de cada grup, així com quins alumnes pertanyen a cada grup. Per a fer això, crearem la classe Alumne i la classe Grup. Esta última tindrà, com a un altre atribut, un vector d'alumnes on estaran tots els alumnes de cada grup. Crea l'aplicació AlumnesGrups i crea en ella:

a. La classe Alumne:

- Atributs (privats): dni, nom, cognoms, edat, poble
- Mètodes:
 - o constructor amb paràmetres: dni, nom, cognoms, edat, poble
 - o gets i sets
 - o toString. Retorna una cadena amb les dades. Per exemple:

12999999 Pep Garcia Garcia, 21 anys (Sueca)

b. La classe Grup:

- Atributs (privats): codi, curs, cicle, llistaAlumnes

L'atribut llistaAlumnes ha de ser un ArrayList d'Alumnes (no de noms d'alumnes, sinó d'objectes de la classe Alumne).

- Mètodes:
 - o constructor amb paràmetres: codi, curs i cicle.
 - o gets i sets de codi, curs i cicle (no de llistaAlumnes)
 - o afegir Alumne. Per a afegir un alumne al grup. Li passem com a paràmetre un alumne. En un grup no podran haver més de 20 alumnes. Si cap l'alumne, l'inserirà i retornarà la quantitat d'alumnes que encara caben. Si no cap, retornarà -1.

- *llevarAlumne*. Per a llevar un alumne del grup. Li passem com a paràmetre un alumne. Retornarà true si l'alumne estava en el grup. False en cas contrari.
- *llevarAlumne*: per a llevar un alumne del grup. Li passem com a paràmetre el dni de l'alumne. Retornarà true si l'alumne estava en el grup. False en cas contrari.
- *Quantitat*: retornarà la quantitat d'alumnes del grup.
- *getAlumne*: li passem com a paràmetre el dni i ha de retornar l'Alumne corresponent (no el nom). Si no està, retornar null.
- *toString*. Retornarà una cadena amb les dades del grup i dels alumnes:

GRUP: 1DAM Curs: 1 Cicle: Desenv.Aplic.Informàtiques
 12999999 Pep Garcia Garcia, 21 anys (Sueca)
 86444368 Pepa Garcia Garcia, 23 anys (Sueca)
 94577544 Pepet Manyes Garcia, 18 anys (Simat)

- Crea altres mètodes que cregues convenient

c. La classe principal. En el mètode main:

- Crea un ArrayList de grups (anomenat grupsInsti) on estaran tots els grups de l'institut.
- Crea un ArrayList d'alumnes (anomenat alumnesInsti) on estaran tots els alumnes de l'institut.
- Fes proves per a utilitzar eixos ArrayList i utilitzar els mètodes de les classes fetes anteriorment. O, millor, un bucle amb el següent menú:
 - Crear grups
 - Crear alumnes
 - Assignar alumnes a grups
 - Desassignar alumnes a grups
 - Etc

9.2 La classe Vector

Esta classe fa el mateix que *ArrayList*. També té mètodes per a inserir objectes, per poder saber si inclou cert objecte o no, per buidar-les, etc.

Diferències respecte la classe ArrayList:

1. Assignació dinàmica de memòria:

Si necessiten augmentar la seua capacitat, un *ArrayList* la incrementa un 50%, mentre el vector un 100%. A més, un dels constructors de la classe *Vector* pot indicar eixe %.

```
public Vector(int capacitatInicial, int increment)
```

L'operació d'augmentar la capacitat d'un *Vector* o d'un *ArrayList* consumeix temps, cosa que el programador ha de tindre en compte.

2. Seguretat front a fils d'execució

Els mètodes de la classe *Vector* són *synchronized* mentre que els de la classe *ArrayList* no.

Això vol dir que els mètodes de la classe *Vector* són segurs pel que fa a "atacs" d'altres fils d'execució (*threads*): cap altre fil pot actuar sobre el *Vector* mentre s'està executant algun d'estos mètodes! En canvi, els mètodes de la classe *ArrayList* són insegurs en este aspecte (com ho és la manipulació directa sobre arrays).

Però es paga un preu: els mètodes *synchronized* són un poc més lents.