Tema 9
Programació Orientada a Objectes. Classes amb mètodes

1. Introducció

Ja diguérem que la POO (Programació Orientada a Objectes) és una metodologia que basa l'estructura dels programes en objectes. I que els objectes es classifiquen en classes. I que de cada classe ens poden interessar determinades característiques i operacions per a descriure cadascun dels seus objectes.

Per tant, per a descriure una classe necessitem:

- ✓ **Atributs** o característiques — Vist al tema anterior.
- ✓ **Mètodes** o operacions — Ho vorem en este tema.

Exemple: la clase Casa

- **Atributs:** 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.
- **Mètodes**: Construir-la, destruir-la, pintar la façana, obrir una nova finestra, pintar habitacions, etc.

Evidentment, a partir dels requeriments de l'aplicació, el programador determinarà quins atributs i quins mètodes cal definir en cada classe.

No ve mal recordar els conceptes de la POO

CONCEPTES POO	TERMINOLOGIA INFORMAL	ANALOGIA EN LA VIDA REAL	EXEMPLES
Classe	Tipus	Plantilla plastificada	*Alumne*
Objectes	Variables	Fotocòpies a emplenar	*alu1, alu2, alu3*
Atributs	Variables membre	Apartats de la fitxa	*num, nom, dom, curs*
Mètodes	Funcions i proced.	Accions sobre la fotocòpia	*matricularAlumne(), canviDeCurs()*...
Estat de l'objecte	Valors dels atributs	El que s'escriu a la fotocòpia	*num = 3, nom = "Pep"*

2. Estructura d'un programa OO

Exemple de l'estructura d'un programa OO. Després ho vorem detalladament.

```
// ------------------------------ CLASSE Alumne ----------------
class Alumne {
        // ATRIBUTS
        String nom;
        int edat;

        // MÈTODES
        void setNom(String n)  { this.nom = n;}
        void setEdat(int e)  { this.edat = e;}
        String getNom()  { return this.nom; }
        int getEdat() { return this.edat; }
        String dades() { return this.nom + " " + this.edat; }
        void augmentaEdat(int anys)  { this.edat += anys; }

        // CONSTRUCTOR d'objectes de la classe Alumne
        Alumne(String nom, int edat){
                this.nom = n;
                this.edat = e;
   }
}
// ---------------------------- CLASSE PRINCIPAL ----------------
public class Main {
   public static void main(String[] args) {
        // DEFINICIÓ I ÚS DELS OBJECTES
        Alumne a1 = new Alumne("Pep", 18);
         if (a1.getEdat() >= 18)
                System.out.println(a1.getNom()+ " pot eixir");
        a1.setEdat(20);
        a1.augmentaEdat();
        System.out.println(a1.dades());
   }
}
```

Exercicis introductoris

1. Per vore si t'han quedat clars els conceptes d'objecte, classe, atributs, mètodes i estat, pensa en els distints objectes que solen aparéixer en un programa de marcianets. Quines classes d'objectes apareixen? Apareixen molts objectes de la mateixa classe? Pensa en alguns atributs i alguns mètodes per a cada classe, així com l'estat que podria tindre algun objecte.

2. Al llarg del tema farem exercicis sobre un concessionari per a tindre un control dels vehicles que ven. De moment crea una aplicació anomenada *Concessionari*. Allí crea la classe *Cotxe*, en un fitxer diferent al de la classe principal on està el *main*. De moment crea la classe sense atributs ni mètodes (ja anirem ampliant-la conforme anem veient més temari).

3. Definició d'una classe

3.1. Sintaxi

```
<modeD'Accés> class nomClasse {
        // ---------- DEFINICIÓ D'ATRIBUTS ----------
        <modeD'Accés> tipus atribut_1;
        …
        <modeD'Accés> tipus atribut_N;
        // ---------- DEFINICIÓ DE MÈTODES ----------
        <modeD'Accés> tipus mètode_1 (llista_de_paràmetres) {
                // Cos del mètode
        }
        …
        <modeD'Accés> tipus mètode_M (llista_de_paràmetres) {
                // Cos del mètode
        }
        // ---------- DEFINICIÓ DE CONSTRUCTORS -------
        <modeD'Accés> nomClasse (llista_de_paràmetres) {
                // Inicialització dels atributs, etc.
        }
}
```

3.2. Modes d'accés de la classe

Els modes d'accés de la classe poden ser una combinació d'entre els següents:

- ***public*** / (res): s'utilitza per a indicar des d'on es pot utilitzar
  - Si no es posa cap mode d'accés, eixa classe només podrà ser accedida des de fitxers del mateix paquet on està la classe.
  - Si es posa *public* la classe podrà ser usada des de qualsevol lloc de l'aplicació. En un fitxer .java només pot haver una classe pública i ha de tindre el mateix nom que el fitxer (sense l'extensió, clar).
- ***abstract*** / ***final*** / *(res): s'utiliza per a l'herència de classes (ja ho vorem)*
  - Si una classe és abstract no podrem crear objectes d'eixa classe, sinó que s'hauran de definir de classes filles d'eixa classe.
  - Si una classe és final no es podran crear subclasses d'ella.

3.3. Atributs

Com ja hem vist al tema anterior, són les variables membre que caracteritzen l'objecte. El seu valor en un moment donat indica l'**estat** de l'objecte.

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

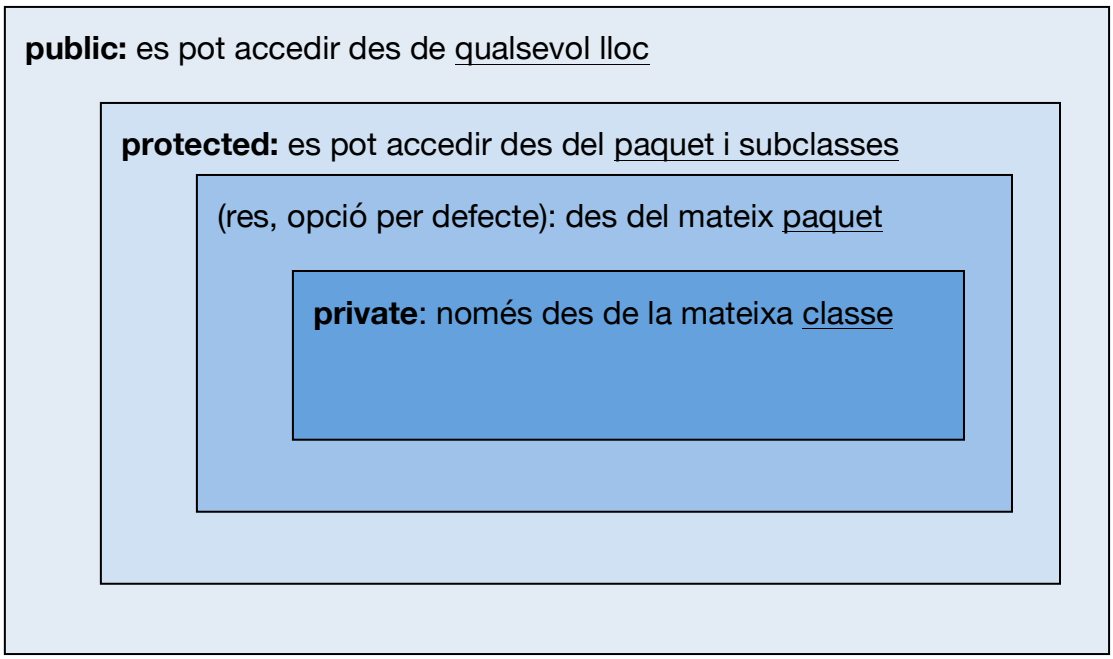
```
<nivellD'Accés> <TipusDeDades> nomAtribut ;
```

Tipus de dades dels atributs

Recordem que el tipus de dades dels atributs pot ser qualsevol: enter, real, lògic, caràcter, cadena, vector, matriu... o una altra classe (classes niuades).

Nivell d'accés dels atributs (i dels mètodes)

Els atributs (i mètodes) sempre són accessibles des de la classe on estan definits però podem que també puguen 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:



Consell: **els atributs solen ser privats**, i la classe que els inclou, si cal, donarà accés a ells mitjançant els mètodes (generalment públics) de lectura i modificació d'eixos atributs. És el que es coneix com a **encapsulament** (que vorem en detall més endavant).

Per si no ha quedat clar, mireu en la següent taula els diferents nivells d'accés dels atributs i mètodes:

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**	**Qualsev. lloc**
public	Des de qualsevol lloc	S	S	S	S
protected	Des del mateix paquet de la classe i en les subclasses	S	S	S	N
	Des del mateix paquet de la classe	S	S	N	N
private	Des de cap altre lloc. Només des de la pròpia classe	S	N	N	N

Exercici

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

3.4 Mètodes

Són les operacions (procediments o funcions) que s'apliquen sobre els objectes. Entre altres coses, permeten canviar el seu estat (valor dels atributs), fer operacions sobre el seu estat, consultes... però tenen un funcionament un poc diferent ja que actuen sobre un objecte. Veiem com invocaríem un mètode i com es defineix:

Ús dels mètodes

Després de crear un objecte (en el *main* o on siga), si volem fer alguna cosa sobre ell, "li enviarem un missatge": indicarem el nom de l'objecte, un punt, i el missatge (nom del mètode amb els paràmetres). Per exemple, si tenim un objecte de la classe Cercle (on tindrà el radi, etc), podríem fer:

```
public class Main {
   public static void main(String[] args) {
      Cercle cercle1 = new Cercle();
       cercle1.radi = 10;
       System.out.println( "L'àrea és: " + cercle1.area() );
       cercle1.incrementaRadi(3);
       ...
}
```

Missatge a *cercle1*:
"*cercle1*, dona'm la teua àrea"

Missatge a *cercle1*:
"*cercle1*, incrementa el teu radi en 3"

Definició dels mètodes

```
public 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;
        }
}
```

El *this* és una referència a *cercle1* (l'objecte de la crida).

Per tant, usarem el *this* per a accedir als atributs, encara que és opcional.

Modificador *static* dels mètodes

Recordem que sempre posàvem la paraula *static* en la declaració de funcions i procediments. Però observem que ara no l'estem posant.

Quan posarem el modificador *static* en un mètode?

Mètode sense *static*:

Quan volem que s'invoque amb el nom de l'objecte:

Posem el nom de l'objecte davant del mètode

```
cercle1.diametre()
```

Dins del mètode podrem accedir als atributs de l'objecte (amb this):

Si li posàrem *static* ens donaria error de compilació (ja que usa el *this*):

non-static variable cannot be referenced from a static context

```
class Cercle {
        // ---------- ATRIBUTS ------------------
        int radi;

        // ---------- MÈTODES ------------------
        static int diametre() {
                return (2 * this.radi);
        }
}
```

Mètodes amb *static*:

Quan volem que s'invoquen sense cap objecte:

No posem res davant del mètode

```
diametre(10);

Trigonometria.diametre(10);
```

Si el mètode està definit en altra classe, posarem el nom d'eixa classe.

En estos casos no tindrà sentit que el mètode accedisca als atributs de la classe ja que no estem treballant en cap objecte.

Si en la funció usàrem *this* ens donaria error de compilació ja que he posat *static*.

```
static int diametre(int r) {
        return (2 * r);
}
```

Exercicis sobre mètodes

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

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

5. En el main, crea 2 cotxes (cotxe1 i cotxe2) i fes les següents operacions sobre ells usant els mètodes corresponents:
   a) Demana una matrícula per teclat i posa-li-la a *cotxe1*. Idem al *cotxe2*.
   b) Pinta el *cotxe1* de groc i el *cotxe2* de lila.
   c) Passa la revisió del 1r any de *cotxe1*, i la del 2n any dels 2 cotxes. Comprova que no podries fer-ho sense el mètode, ja que l'atribut *revisions* és privat.
   d) Evita que done error si intentem passar un número de revisió inexistent (fora de les dimensions del vector) però no ho faces en el *main*, sinó en el mètode corresponent.
   e) Mostra les dades del cotxe que ha passat més revisions.

3.5. Mètodes constructors

Definició

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

Utilitat

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

Exemple

```
// -------------------------------- CLASSE Alumne ----------

class Alumne {
    // ATRIBUTS
    String nom;
    int edat;
    String poble;

    // MÈTODES
    ...

    // CONSTRUCTORS

    Alumne(String nom, int edat, String poble){
        this.nom = nom;
        this.edat = edat;
        this.poble = poble;
    }

    Alumne(){
        this.poble = "Tavernes";
    }
}


// -------------------------------- CLASSE PRINCIPAL ------
public class Main {
    public static void main(String[] args) {

            // DEFINICIÓ D'OBJECTES
            Alumne a1 = new Alumne("Pep", 18, "Sueca");
            Alumne a2 = new Alumne();
            Alumne a3 = new Alumne();
            ...
}
```

Havíem dit que l'ús de *this* és opcional, però no en este cas, ja que hi hauria conflicte de noms amb els paràmetres.

Exemple de constructor amb paràmetres.

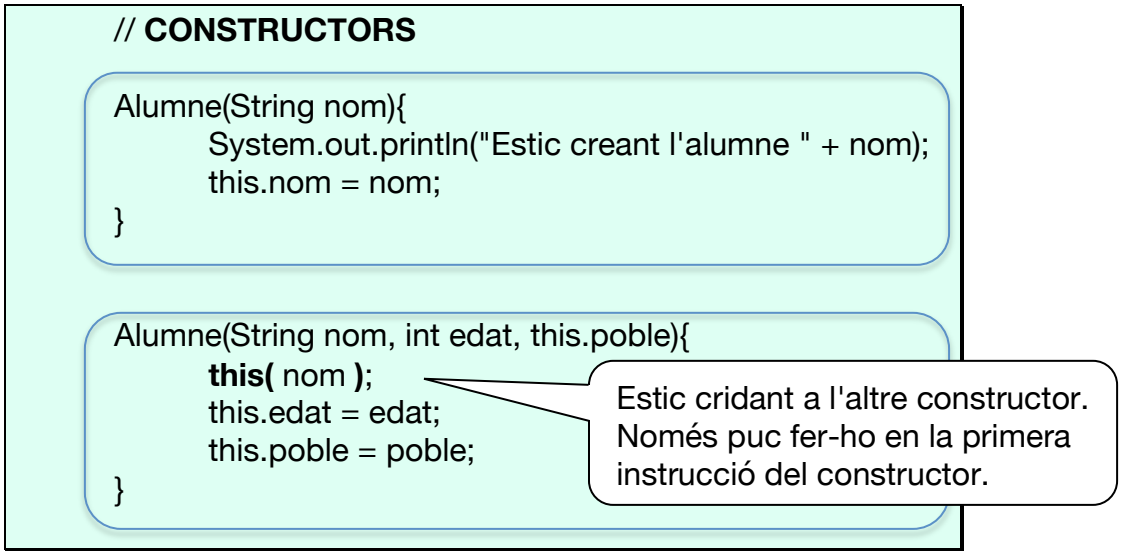
Exemple de constructor sense paràmetres.

Invoquem el constructor de 3 paràmetres.

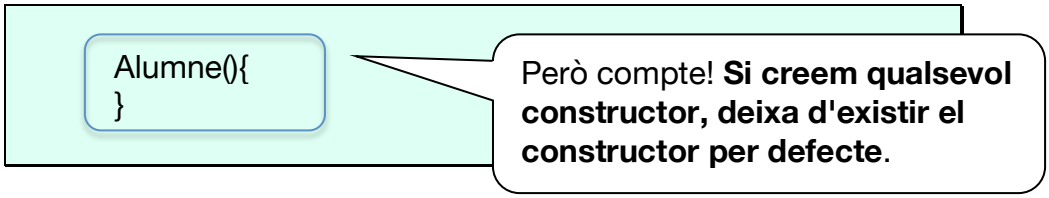
Invoquem el constructor sense paràmetres.

Propietats dels constructors

- El **nom** del constructor sempre és el nom de la classe.
- 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*).
- No podem **invocar els constructors** directament. Només es fa amb l'operador *new*, mentre instanciem la classe.
- El constructor pot estar **sobrecarregat**: una classe pot tindre més d'un constructor (sempre amb el mateix nom) però amb distinta quantitat de paràmetres o de tipus distints En eixe cas, des d'un constructor de la classe es pot cridar a altre constructor de la mateixa classe (sempre en la 1a línia del constructor), amb: *this(paràmetres)*;



- Constructor per defecte. Si no hem especificat cap constructor, es crea automàticament un constructor sense paràmetres. No es veu, però està:



Exercicis sobre creació dels constructors

6. Crea un constructor de la classe *Cotxe* que reba com a paràmetres el número de bastidor, la matrícula, marca, model, color i preu. El constructor deu inicialitzar l'estat de l'objecte amb estos. El vector de revisions es posarà tot a fals. Voràs que ara dona error la creació dels cotxes anteriors que usaven el constructor sense paràmetres. Què ha passat? Passa a l'exercici següent per a arreglar-ho.

7. Totes les classes tenen un constructor per defecte (sense paràmetres) però "desapareix" si creem altre constructor. Per tant, crea un constructor sense paràmetres que no faça res. Comprova que ara ja no et dona error la creació dels cotxes que usaven el constructor sense paràmetres.

8. Crea un altre constructor de la classe *Cotxe* que reba com a paràmetres el número de bastidor, la marca i el model. No inicialitzarà cap atribut, sinó que cridarà a l'altre constructor per a que ho faça ell, amb l'ús de *this*(paràmetres)).

9. Crea altre constructor a la classe *Cotxe*, que reba només la matrícula. 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? Perquè la crida a un altre constructor ha de ser la primera instrucció del constructor. Per tant, no hi haurà més remei que inicialitzar els atributs directament.

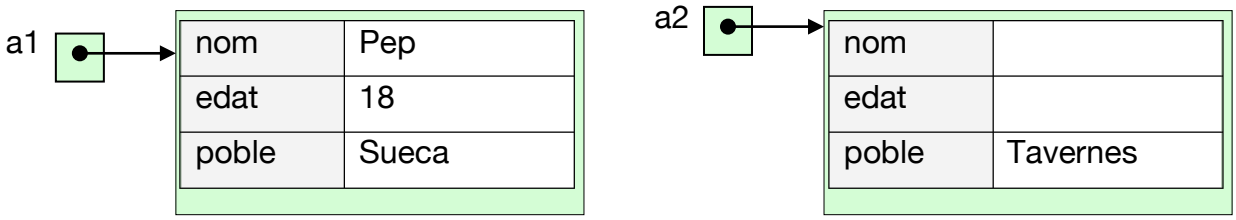
Crides als constructors (instanciació d'objectes)

Sempre havíem creat els objectes amb el *new*. Estàvem cridant al constructor per defecte (el que no té paràmetres).

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

Com ara ens hem creat els nostres constructors, podem usar-los posant els corresponents paràmetres al *new*:

```
Alumne a1 = new Alumne("Pep",  18, "Sueca);
Alumne a2 = new Alumne();
...
```



És a dir, una instrucció d'**instanciació** d'un objecte (crear-lo, amb el new) fa açò:

- Reserva memòria per a l'objecte.
- Crida al constructor corresponent, qui sol omplir dades en eixa memòria.
- Assigna a l'objecte una referència (adreça) a la memòria reservada.

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, com sabem, hi ha una forma abreujada de fer-ho per als objectes de la classe String:

String nom = "Pep";

Exercicis de crides als constructors

10. Crea els objectes cotxe3, cotxe4 i cotxe5, amb valors qualssevol, usant els 3 constructors de cotxes que has definit.

11. Demana per teclat les dades d'un cotxe (bastidor, marca, model, color i preu. Crea l'objecte *cotxe6* passant-li com a paràmetres al constructor les dades que s'han arreplegat per teclat. Nota: passa una cadena buida per a la matrícula.

12. Intenta crear l'objecte *cotxe7* passant-li al constructor només el preu. Per què no et deixa?

13. Crea el vector cotxesAparador, de 5 cotxes. Inicialitza'ls tots usant el constructor al qual només se li passa la matrícula. Les matrícules aniran de l'1 al 5.

3.6 Destructors

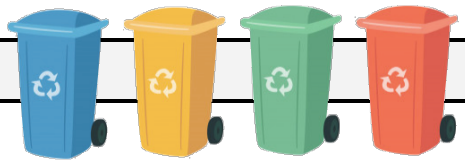
Igual que es creen els objectes (amb *new*), com que ocupen espai en memòria, és convenient alliberar eixa memòria quan ja no vagen a utilitzar-se els objectes durant l'execució del programa.

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*** (recol·lector de fems), que és qui s'encarrega de fer este alliberament de recursos. Per tant, el programador no s'ha de preocupar de res.

Exemples d'alliberament de memòria:

- Si fem un 2n *new* sobre un objecte, la memòria reservada en el 1r new sel·limina.
- En eixir d'una funció, sel·limina la memòria dels objectes locals a ella.

4 Abstracció



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

Propietats POO			
Abstracció	Encapsulament	Herència	Polimorfisme
			

En este apartat parlarem de l'abstracció.

Definició

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

Avantatges

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

Mecanismes d'abstracció

L'**abstracció en la POO** és la capacitat per a aïllar en una classe els atributs i mètodes comuns a un determinat conjunt d'objectes. Però l'abstracció no la trobem únicament en les classes:

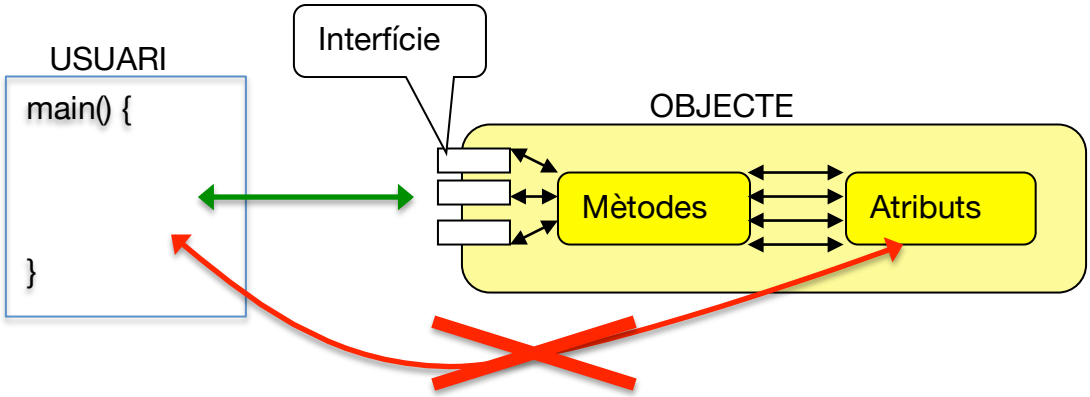
MECANISMES D'ABSTRACCIÓ	CARACTERÍSTIQUES
Funcions	Conjunt d'instruccions que es pot parametritzar i ser invocat en qualsevol moment.
Mòduls	Conjunt de funcions que guarden alguna relació. Són fitxers, paquets, llibreries...
Classes	Conjunt de dades i funcions que descriuen una série d'objectes. Incorpora tècniques com l'herència, polimorfisme, etc.

5. Encapsulament

5.1. Introducció

Els usuaris dels objectes (*main*, per exemple) poden accedir als atributs directament, però no és aconsellable.

En compte d'això, els usuaris dels objectes haurien d'accedir als atributs mitjançant una interfície: el conjunt de noms dels mètodes públics de la classe. Estos mètodes faran d'intermediaris entre l'usuari i l'objecte.



Així, si el programador vol canviar la forma d'accedir a un atribut, no ha de canviar tot el programa sinó només el mètode que dona accés a eixe atribut.

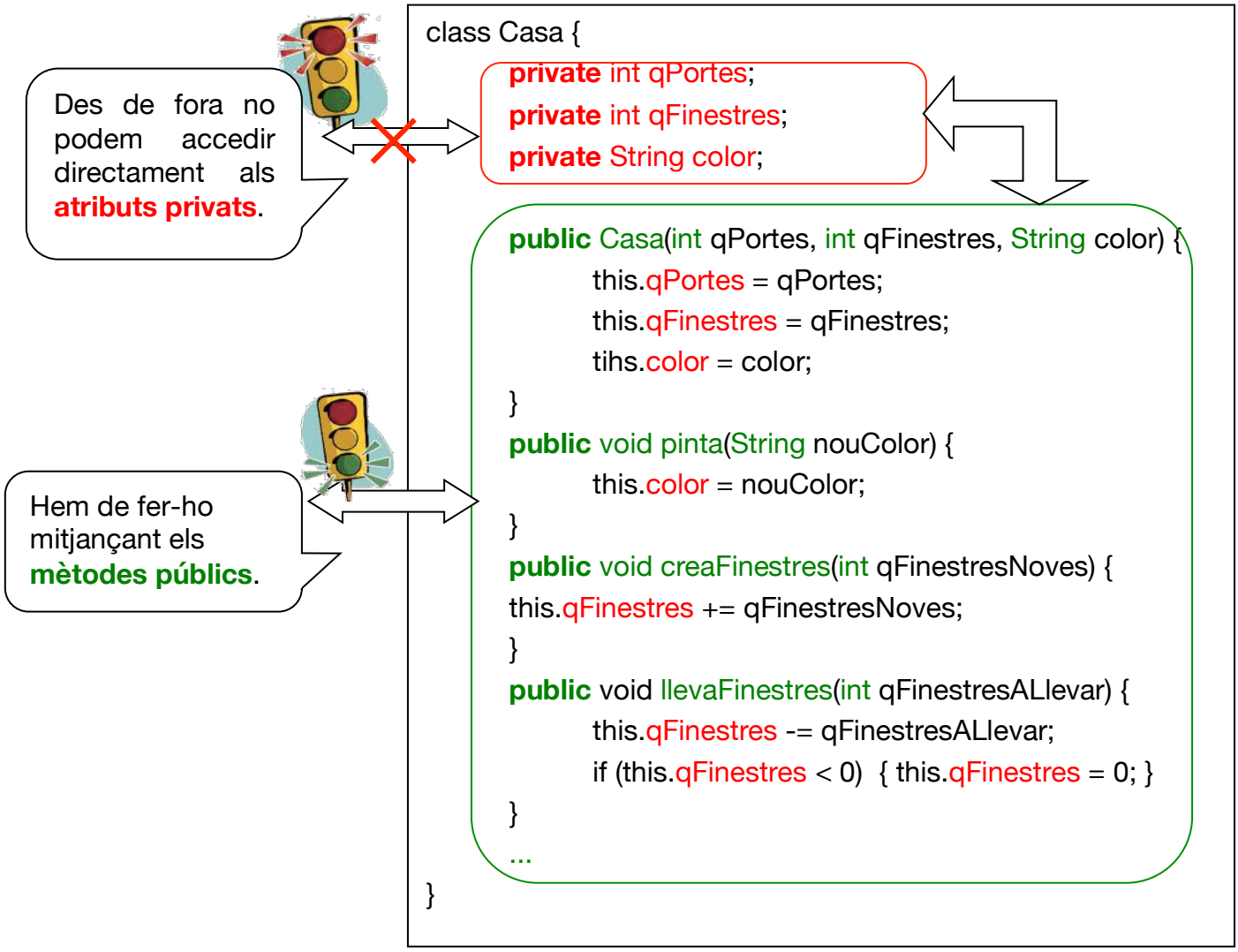
L'***encapsulament*** és la propietat que tenen els objectes d'ocultar els seus atributs a altres parts del programa.

Com ho farem?

- Posarem els **atributs privats**
- Crearem **mètodes públics** per a accedir a eixos atributs.

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. El conjunt de mètodes públics és la ***interfície***.

Vegem-ho amb un exemple.




```
public class Main {
    public static void main(String[] args) {
        ...
        Casa macasa = new Casa(1, 3, “blanc”);
        ...
        macasa.qfinestres += 2;

        macasa.creaFinestres(2);
}
```

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 més finestres de les que té la casa.

Per tant, són els mètodes dels objectes qui controlen els errors. Podríem dir que els mètodes són "professionals" en crear finestres, llevar-les... Així, els usuaris de l'objecte no s'han de preocupar de comprovar si hi ha suficients finestres o no abans de tancar-les.

5.2. Getters i setters

Com hem vist, convé que els atributs siguen privats. Per tant, per cada atribut necessitarem 2 mètodes: un per a modificar el seu valor i un altre per a consultar-lo.

Els mètodes destinats a donar un valor a un atribut s'anomenen ***setters***, mentre que els destinats a consultar el valor de l'atribut s'anomenen ***getters***.

Per simplificar la tasca dels programadors, hi ha una convenció pel que fa als noms dels *setters* i *getters*: sempre els anomenarem amb les lletres get o set més el nom de l'atribut (amb la primera lletra en majúscula). Per exemple, per a l'atribut *edat* de la classe *Alumne* tindrem els mètodes *setEdat* i *getEdat*:

```
public class Alumne {
        private int edat;

        public int getEdat() {
            return edat;
        }

        public void setEdat(int edat) {
            this.edat = edat;
        }
}
```

Al *getter* de l'atribut *edat* no se li passa res i retorna l'edat de l'objecte.

Al *setter* de l'atribut *edat* se li passa una edat i l'assigna a l'atribut de l'objecte.

```
public class Proves {
    public static void main(String[] args) {
        Alumne a1 = new Alumne();

        System.out.println("Edat:");
        a1.setEdat(teclat.nextInt());
        ...
        System.out.println("En complirà " + (a1.getEdat()+1))
    }
}
```

Invoquem al *setter* d'*edat*.

Invoquem al *getter* d'*edat*.

Exercicis sobre encapsulament

14. Encapsula els atributs *color* i *preu* de la classe *Cotxe*: posa'ls com a privats i crea els *getters* i *setters* corresponents. Fes ús d'eixos mètodes des del *main*.

Construcció automàtica de *getters* i *setters*

Per facilitar la tasca del programador, els IDEs solen incorporar una inserció automàtica de codi per a construir els *getters* i *setters* (entre altres coses).

Per exemple, en *Netbeans* es faria així:

- Cal fer clic dret en algun lloc dins la classe que volem encapsular.
- Opció *Insert code*
- Opció *Getter and setter*
- Seleccionar els atributs dels quals volem crear *getter* i *setter*.
- Marcarem la casella *Encapsulate Fields* perquè, a més, pose com a privats els atributs corresponents.

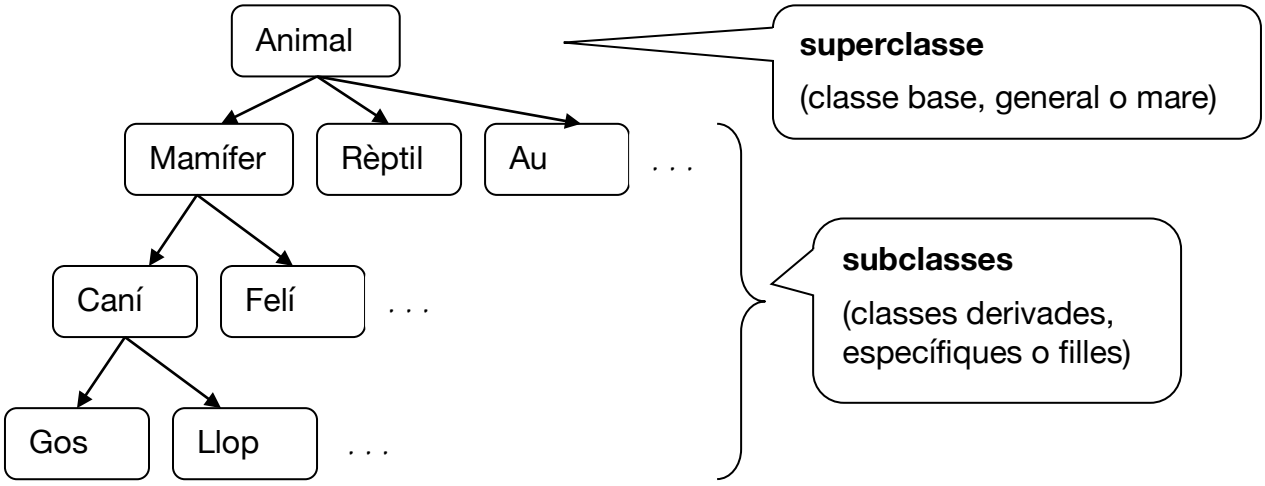
Exercicis sobre encapsulament automàtic

15. Encapsula els atributs *numBastidor*, *matricula*, *marca* i *model* de la classe *Cotxe*. Fes-ho amb la inserció automàtica que ofereix *Netbeans*, *Eclipse* o l'IDE que utilitzes.

6 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

- La subclasse hereta els atributs i mètodes de la classe.
- La subclasse pot tindre nous atributs i mètodes.
- La subclasse pot redefinir mètodes de la superclasse (per exemple, un mateix mètode pot tindre una implementació en la classe pare i una altra distinta en la classe filla).

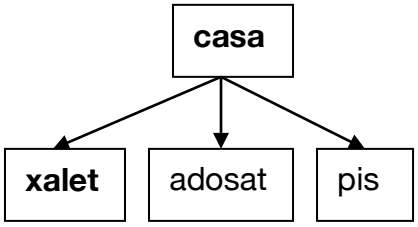
Avantatges

- Reusabilitat o reutilització del codi: una subclasse pot aprofitar el codi d'una superclasse ja existent, modificant-lo mínimament per a adaptar-lo a les seues necessitats.
- Els canvis en una classe general (nous atributs o mètodes, etc) es voran reflectits també en les classes filles.

Exemples

L'herència és la forma natural de definir objectes en la vida real. Per exemple, podríem definir un **xalet** com una **casa** però amb jardí. Per tant, un xalet és un tipus de casa: xalet serà una classe filla de casa

- Què és un **xalet**?
- És una **casa** amb jardí.



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

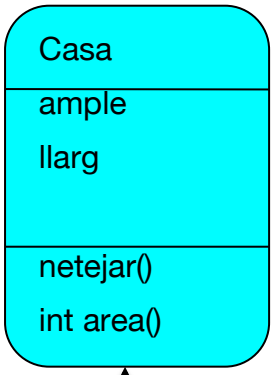
- El xalet (com a casa que és) també tindrà el atributs de la casa (portes, finestres, color...) i els mètodes de la casa (construir, pintar...)
- El xalet tindrà nous atributs: color de la tanca, m^2 de jardí, quantitat d'arbres, etc, i també nous mètodes: pintar la tanca, plantar un arbre, arrancar 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 pinte també la tanca.

Implementació

Veiem-ho amb l'exemple del xalet.

Superclasse

```
public class Casa {
        int ample;
        int llarg;
        void netejar(){
            System.out.println("Casa neta");
        }
        int area(){
            return this.ample * this.llarg;
        }
}
```



Subclasse

```
public class Xalet extends Casa{
        int m2Jardi;

        void tallarGespa(){
            System.out.println("Gespa tallada");
        }

        int area(){
            int areaCasa = super.area();
            return areaCasa + m2Jardi;
        }
}
```

L'herència s'indica amb l'***extends***.

Com l'àrea dels xalets és diferent, sobreescrivim el mètode *area()*.

Amb ***super.*** podem accedir a mètodes i atributs de la mare.


```
public class Main {
    public static void main(String[] args) {
        Casa casa1 = new Casa();
        Xalet xalet1 = new Xalet();

        xalet1.ample = 10;
        xalet1.llarg = 15;
        xalet1.m2Jardi = 50;
        ...
        xalet1.netejar();
        xalet1.tallarGespa();

        System.out.println(xalet1.area());
        System.out.println(casa1.area());
    }
}
```

Un xalet pot usar tant els seus atributs com els heretats de la casa.

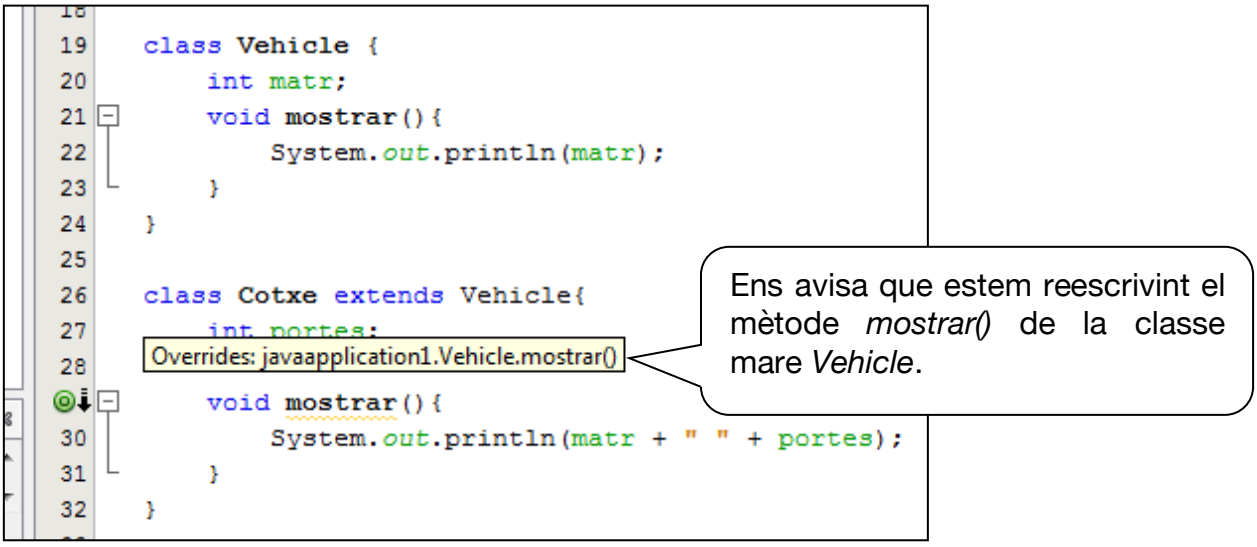
Un xalet pot usar tant els seus mètodes com els heretats de la casa.

El *xalet1* invocarà el mètode *area()* de *Xalet*.

La *casa1* invocarà el mètode *area()* de *Casa*.

Què és el *@Override*?

Quan estem sobreescrivint un mètode, Netbeans ens avisa:



No és un error. Simplement ens avisa que el mètode mostrar de la classe *Cotxe* ja l'hem definit en la classe mare *Vehicle* i, per tant, l'oculta. És a dir: abans de sobreescriure el mètode, si un objecte de la classe *Cotxe* cridava a *mostrar()* s'executava el de la classe pare *Vehicle*, però ara ja no podrem invocar-lo en objectes de la classe *Cotxe* ja que estarem invocant al *mostrar()* de la classe *Cotxe*.

Si fem clic a l'avís ens aconsella posar *@Override* dalt del mètode:

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

El funcionament serà el mateix però té distintes utilitats:

- Indica al programador que eixe mètode "substitueix" l'altre en el fill (que "oculta" el mètode del pare).
- 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 sobreescrivint 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.

Els constructors de la subclasse

Observem el funcionament dels constructors de la subclasse amb el mateix exemple del Xalet:

Superclasse

```
public class  Casa {
        ...
        Casa(int ample, int llarg){
            this.ample = ample;
            this.llarg = llarg;
            System.out.println("Casa creada");
        }
}
```

Subclasse

```
public class Xalet extends  Casa {
        ...
        Xalet(int ample, int llarg, int m2Jardi){
            super(ample, llarg);
            this.m2Jardi = m2Jardi;
            System.out.println("Xalet creat");
        }
}
```

Crida al constructor de la superclasse (sempre en la 1a instrucció)

Si no estiguera eixa crida, estaria esta per defecte:

```
super();
```

Per tant, en este cas hauria donat error, ja que no existeix el constructor de la classe mare sense paràmetres.

Si no haguérem posat cap constructor de *Xalet*, estaria este constructor per defecte:

```
Xalet(){
    super();
}
```

Per tant, en este cas també hauria donat error, ja que no existeix el constructor de la classe mare sense paràmetres.

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

        Casa casa1 = new Casa(5, 15);

        Xalet xalet1 = new Xalet(10, 15, 50);
        ...
    }
}
```

Crida al constructor de *Casa*.

Mostrarà: *Casa creada*

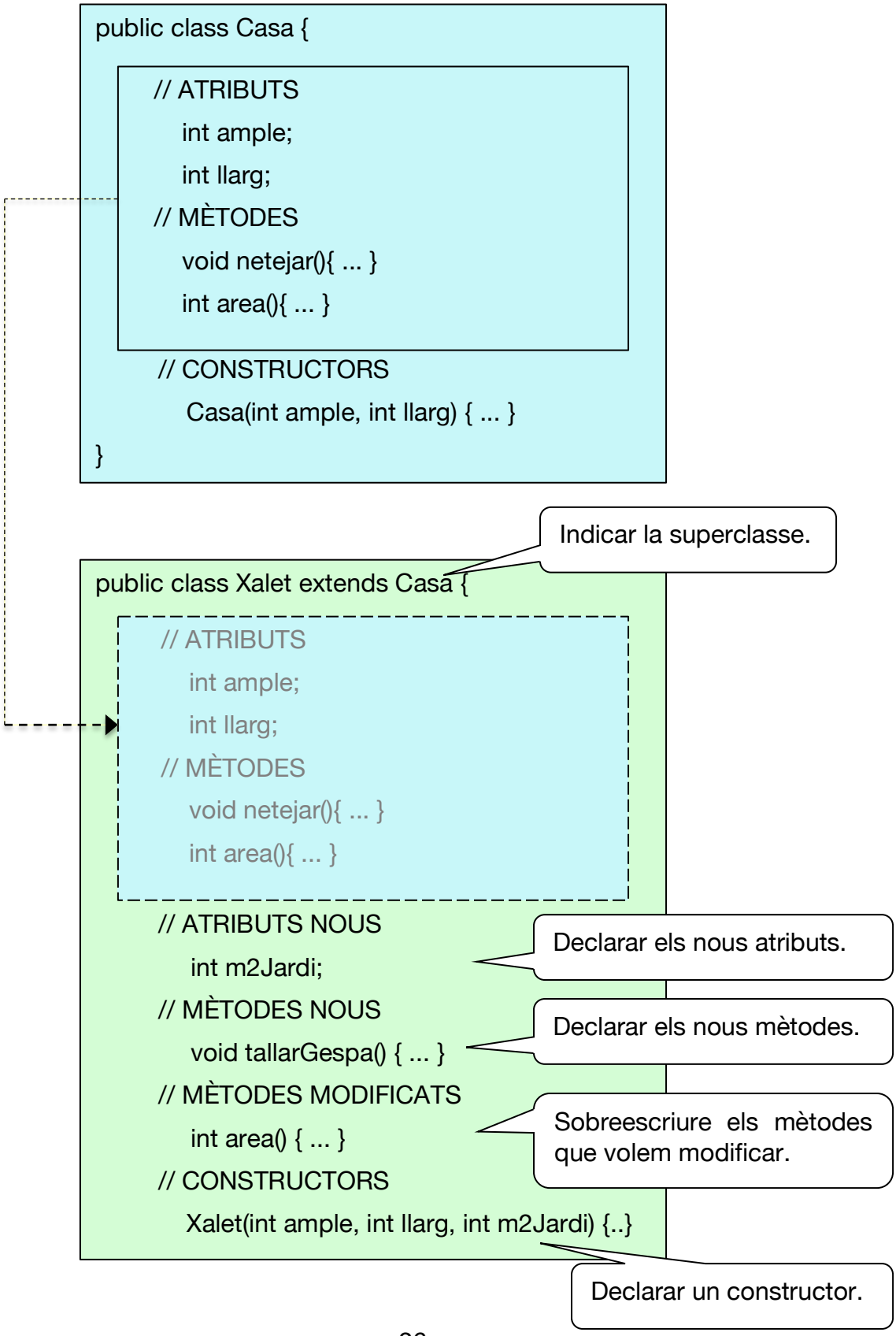
Crida al constructor de *Xalet*, i este crida al constructor de la superclasse (*Casa*).

Mostrarà: *Casa creada*

Xalet creat

Passos a fer per a crear una subclasse

Quan definim una subclasse no hem de copiar atributs i mètodes ja que podrem usar-los encara que no estiguen físicament. Només farem el següent:



Exercicis sobre l'herència

16. Crea, en un altre fitxer, la classe *Cotxe2aMa.*

    a) *Fes que siga* subclasse de la classe *Cotxe*.

    b) Posa en *Cotxe2aMa* un atribut nou, *kms* (enter), i encapsula'l (fes-lo privat i crea els seus mètodes *getter* i *setter*).

    c) Sobreescriu *mostrarDades()* perquè també mostre els kms. Però no faces un copiar-i-apegar sinó que has d'invocar al *mostrarDades()* de la mare.

    d) Comprova que si en la classe *Cotxe* eliminares el constructor sense paràmetres, tindries un error de compilació en la classe *Cotxe2aMa*. Pensa per què.

    e) Fes un constructor de la classe *Cotxe2aMa* que agafe com a paràmetres el *bastidor*, *marca*, *model*, *matricula* i *km*. Este ha d'invocar al constructor adequat de la superclasse i inicialitzar la resta d'atributs.

17. En el main fes el següent:

    a) Crea l'objecte *cotxeAntic* de la classe *Cotxe2aMa* passant-li al constructor els paràmetres necessaris (inventa-te'ls).

    b) Crea l'objecte *cotxeNou* de la classe Cotxe. Mostra per pantalla les dades dels dos cotxes amb crides a *mosrarDades()*. Comprova que no trauen la mateixa informació. Podríem invocar el *mostrarDades()* de la classe *Cotxe* amb l'objecte *cotxeAntic*?

Exemple resum de l'ús de *this. this(...) super. i super(...)*

```
public class Persona {
    String nom;
    public void hola() {
        System.out.println("Hola, " + this.nom);
    }
    public void soc() {
        System.out.println("Sóc persona");
    }
    public Persona(String nom) {
        this.nom = nom;
    }
}
```

```
public class Estudiant extends Persona {
    String curs;

    @Override
    public void soc() {
        System.out.println("Sóc estudiant");
    }
    public Estudiant(String nom) {
        this(nom, "");
    }
    public Estudiant(String nom, String curs) {
        super(nom);
        this.curs = curs;
    }

    public void proves() {
        System.out.println(curs);
        System.out.println(this.curs);

        System.out.println(nom);
        System.out.println(this.nom);
        System.out.println(super.nom);

        soc();
        this.soc();

        hola();
        this.hola();
        super.soc();

    }
}
```

Accés a altre constructor de la classe.

Accés a altre constructor de la mare.

Accés a atributs de la classe.

Accés a atributs de la mare.

Accés a mètodes de la classe.

Accés a mètodes de la mare.

Consell: usa sempre *this*. Quan no siga possible, usa *super*.

Exercici sobre l'ús de *this.* i *super.*

18. Crea un projecte nou i esriu en ell estes classes:

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

```
public class Cotxe extends Vehicle {
    int portes;
    int any;      // Compte! Atribut sobreescrit
    @Override
    public void mostrarDades(){    // Mètode sobreescrit
          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*, dona-li valor a la *matrícula*, *any* i *portes*, i fes una crida al mètode *proves()* d'eixe objecte. Digues si és vertader o fals:

a) Si no hi ha conflicte → Per a accedir al pare puc posar *this*, *super* o res.

b) Si hi ha conflicte → *this* agafa el fill ; *super* el pare ; "res" el fill.

c) Si una variable està en pare i fill, cadascuna pot guardar un valor distint.

Mètodes abstractes i classes abstractes

En una jerarquia de classes podem fer que una classe tinga algun mètode declarat però no implementat. Per exemple:

```
public abstract class Figura {

    private String color;

    public String getColor() {  return color;  }

    public void setColor(String color) { this.color = color; }

    pubic  abstract float area();
}
```

Mètode abstracte: definit però no implementat. Per tant, no podrem crear objectes de la classe Figura (per això és també abstracta), sinó que només podrem crear objectes d'alguna classe filla de Figura que sí que implemente eixe mètode abstracte.

```
class Cercle extends Figura {
    int radi;

    @Override
    public float area() {
        return (float)Math.PI * radi * radi;
    }
}
```

Com la classe *Cercle* és filla de *Figura*, estem obligats a implementar el mètode *area()* per a així poder crear objectes de la classe *Cercle*.

Ara bé, si en la classe *Cercle* no volguérem implementar el mètode *area()*, hauríem de declarar també la classe *Cercle* com abstracta. En eixe cas, no podríem crear objectes de la classe Cercle i no tindríem més remei que crear alguna altra classe filla de *Cercle* que implementara eixe mètode.

Per tant, tenim que:

- **Mètode abstracte**: mètode definit però no implementat.
- **Classe abstracta**: classe que té algun mètode abstracte i que, per tant, no podem instanciar objectes d'eixa classe (sinó que s'hauran d'instanciar a partir d'alguna classe filla que sí que implemente eixos mètodes abstractes).

7. Polimorfisme

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

Una paraula polisèmica és aquella que té distints significats, segons el context. Igual passa amb el polimorfisme en la POO un nom de mètode pot tindre distintes implementacions, i s'invocarà a un o altre segons el context.

Tipus de polimorfisme:

- **Sobrecàrrega basada en els paràmetres**: mètodes amb el mateix nom en una mateixa classe però amb diferents paràmetres.

  S'executarà un mètode o altre depenent dels paràmetres.

- **Sobreescriptura**: mètodes amb el mateix i nom i mateixos paràmetres 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.

- **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.

- **Variables polimòrfiques.** És el polimorfisme pròpiament dit. Ens permet que un objecte definit sobre una superclasse puga ser instanciat en qualsevol de les seues subclasses.

Nota: alguns autors solen diferenciar *sobrecàrrega* de *polimorfisme* però nosaltres vorem la *sobrecàrrega* com un tipus de *polimorfisme*.

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

7.1. Sobrecàrrega basada en els paràmetres

En una classe poden haver distints mètodes amb el mateix nom, sempre que tinguen distints paràmetres d'entrada (no importa el paràmetres d'eixida).

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

Exemple 1: Podem tindre en una classe distints mètodes per a mostrar dades per pantalla, una per a cada tipus de paràmetre.

```
public class Eixida {
   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]);
   }
}
```

Este mètode s'executarà quan li passem un enter.

Eixida.imprimir(nota)

Este mètode s'executarà quan li passem un vector d'enters:

Eixida.imprimir(notes);

Exemple 2: Una classe pot tindre distints constructors (amb el mateix nom, clar), sempre que tinguen distints paràmetres.

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

   public Casa() {
         this.qPortes = 0;
         this.qFinestres = 0;
         this.color = "";
   }
}
```

S'executarà quan creem una casa amb 2 enters i una cadena:

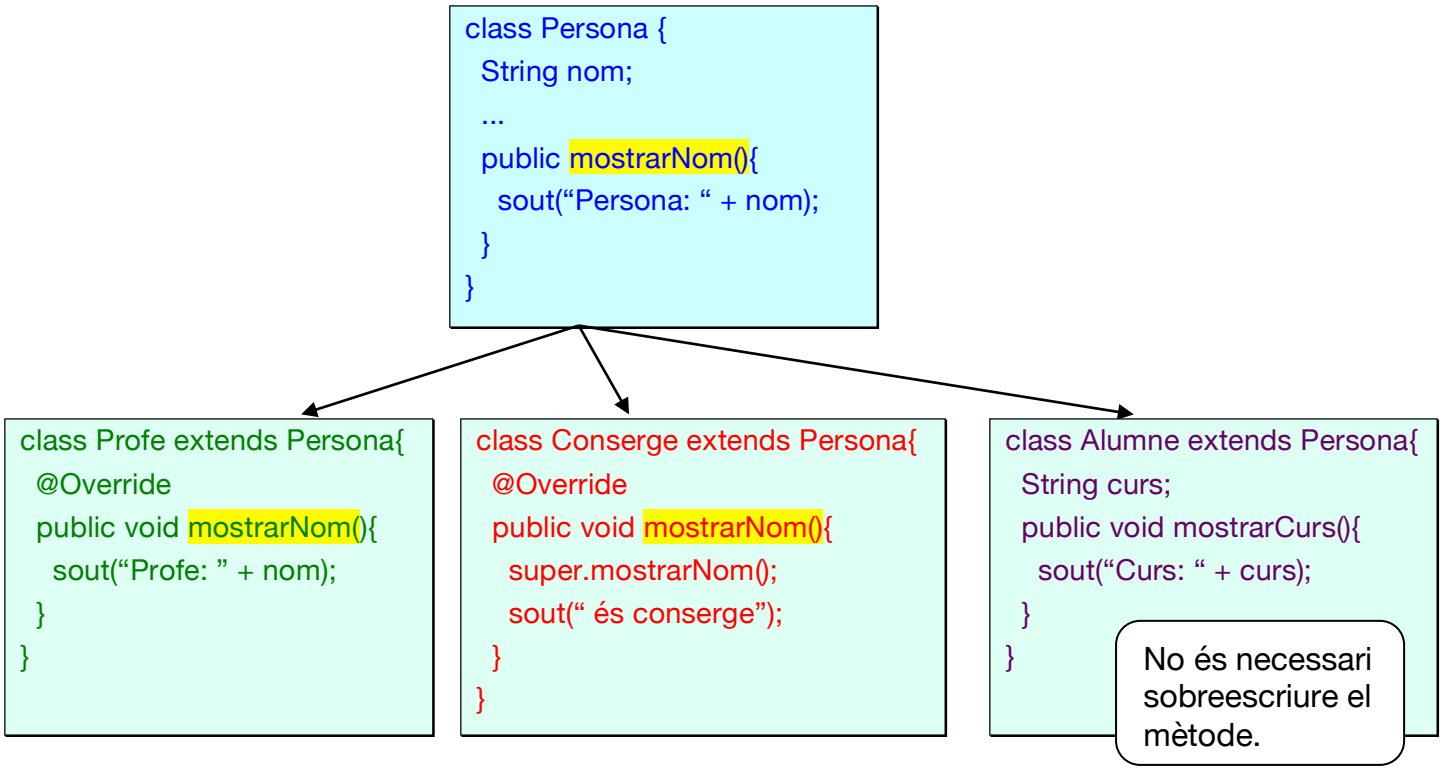
Casa casa1 = new Casa(2, 4, "blanc");

S'executarà quan creem una casa sense paràmetres:

Casa casa2 = new Casa();

7.2. Sobreescriptura

És el tipus de polimorfisme que hem vist en l'herència: mètodes amb el mateix nom i mateixos paràmetres, en classes distintes però amb relacions d'herència entre ells. Se sol dir que la sobrescriptura és el polimorfisme pròpiament dit.



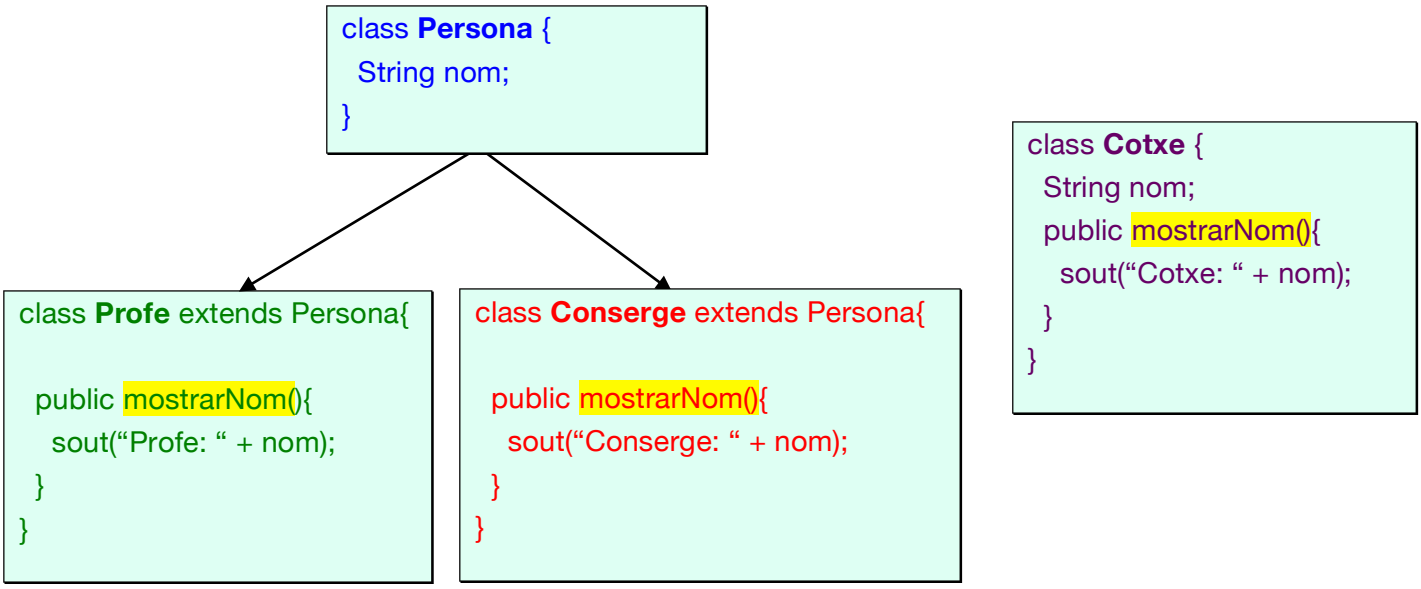
Quan s'invoque un mètode sobrescrit s'executarà el de la classe de l'objecte que ha fet la crida:

```
public class Principal {
  public static void main(String args[]){
    Persona per  = new Persona("Pep");  per.mostrarNom();     // Persona: Pep
    Profe pro    = new Profe("Pep");    pro.mostrarNom();     // Profe: Pep
    Conserge con = new Conserge("Pep"); con.mostrarNom();     // Persona: Pep és conserge
    Alumne alu   = new Alumne("Pep");   alu.mostrarNom();     // Persona: Pep
                                        (new Profe("Pep")).mostrarNom();  // Profe: Pep
  }
}
```

Recordem que, si una subclasse sobreescriu un mètode però en eixa subclasse volem invocar la implementació d'eixe mètode que està en el pare (o en altre ascendent superior), ho hem de fer amb: **super**.mostrarNom();

7.3. Sobrecàrrega basada en l'àmbit

Mètodes amb el mateix nom en diferents classes sense relacions d'herència entre ells.



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 vore, ja que no estan sobreescrivint el mètode, ja que no existeix en la classe pare.

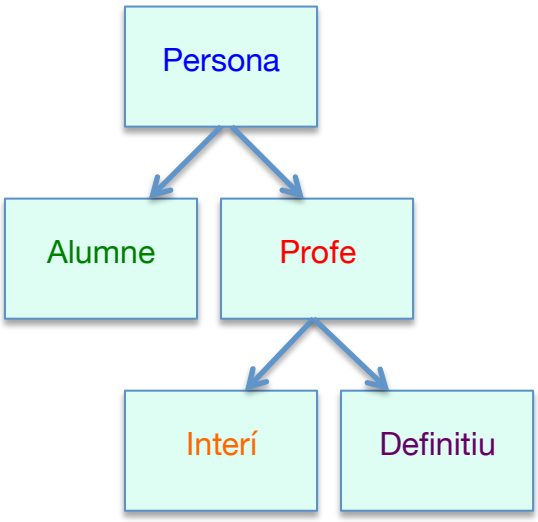
```
public class Principal {
    public static void main(String args[]){
        Persona per    = new Persona();   per.mostrarNom();   // Error de compilació!
        Profe pro      = new Profe();     pro.mostrarNom();   // Profe: Pep
        Conserge con   = new Conserge();  con.mostrarNom();   // Conserge: 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. Per això donarà error de compilació la instrucció *per.mostrarNom()*, ja que la classe *Persona* no té eixe mètode.

7.4. Variables polimòrfiques

Donada esta jerarquia de classes, si tinc un objecte definit com a *Persona*, puc instanciar-lo a un objecte d'eixa classe... o a qualsevol descendent d'ella.



Per exemple:

```
Persona p;
p = new Persona();
p = new Alumne();
p = new Profe();
p = new Definitiu();
```

En una persona puc posar un alumne, un profe... (qualsevol descendent).

En eixe cas diguem que *p* és una **variable polimòrfica**, ja que admet distints tipus d'objectes (*Persona, Alumne, Professor, Definitiu*...), no només 1.

Cal tindre en compte que no té sentit fer coses com:

```
Alumne a;

a = new Profe();

a = new Persona();
```

En un alumne no té sentit posar un professor.

En un alumne no podem posar una persona (ja que podria ser un professor).

És a dir: una variable polimòrfica és un objecte que està **definit** sobre una classe però **instanciat** a una classe descendent:

```
ClasseMare obj = new ClasseDescendent();
```

Classe de la definició

Classe de la instanciació

Algunes utilitats:

a) Tindre una llista d'objectes descendents d'una mateixa classe.

Per exemple, puc voler guardar en una mateixa llista tant alumnes com professors (persones en general):

```
Persona [] llista = new Persona[N];

llista[0] = new Profe();
llista[1] = new Alumne();
llista[2] = new Persona();
```

Vector d'objectes de de la classe *Persona*.

En cada component del vector està definit un objecte de *Persona* però es pot instanciar a *Profe* o a *Alumne*.

Per tant, cada component del vector és una variable polimòrfica.

b) Tindre una funció que puga rebre objectes d'una classe o de filles seues.

Per exemple, vull un procediment que accepte qualsevol tipus de persones: tant alumnes com professors:

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

        static void mostrar( Persona p ){
            System.out.println("--------");
            p.mostrarNom();
            System.out.println("--------");
        }

        public static void main(String args[]){
            Alumne alu = new Alumne();
            mostrar( alu );
        }
    }
}
```

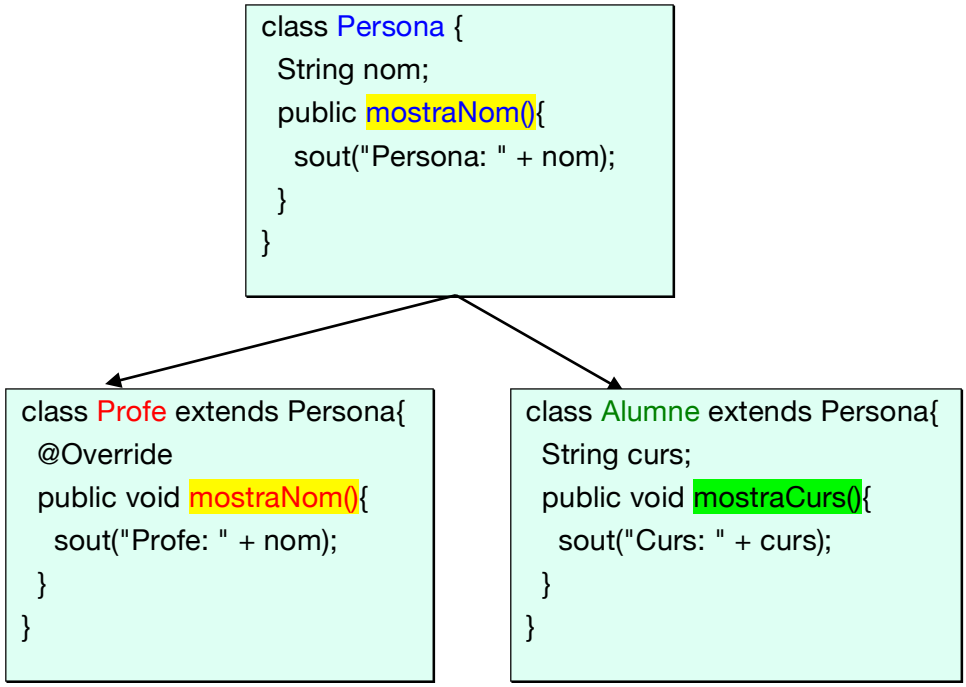
Ojbecte de la classe *Persona* però accepta tant persones com alumnes i profes.

Per tant, *p* és una variable polimòrfica.

Quan es faça eixa crida, el paràmetre *p* (de la funció *mostrar()*), que estava definit com a *Persona*, estarà apuntant a un objecte de la classe *Alumne*. Per tant, *p* serà una variable polimòrfica.

Accés als mètodes i atributs d'una variable polimòrfica:

Suposem que tenim esta jerarquia de classes:




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

        Persona per_pro = new Profe();
        per_pro.nom = "Pep";
        per_pro.mostraNom();

        Persona per_alu = new Alumne();
        per_alu.nom = "Pep";
        per_alu.mostraNom();
```

per_pro està definit com a *Persona* però està instanciat a *Profe*.

Per això, el compilador comprova que *mostraNom()* estiga en *Persona* però quan s'executa el programa, invoca el *mostraNom()* de *Profe*.

per_alu està definit com a *Persona* però està instanciat a *Alumne*.

Per això, el compilador comprova que *mostraNom()* estiga en *Persona*, però quan executa el programa, s'intenta invocar el *mostraNom()* d'Alumne. Ara bé, com no el té, invoca el *mostraNom()* del pare, *Persona*, ja que l'hereta.

Per tant, veiem que:

- En **tems de compilació**: es comprova en la classe de definició de l'objecte.
- En **temps d'execució**: s'executa en la classe que ha fet el *new*.

Però veiem què passa en estos casos:

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

        Persona per_alu = new Alumne();



        per_alu.curs = “1DAM”;



        per_alu.mostraCurs();



        ((Alumne) per_alu).curs = “1DAM”;



        ((Alumne) per_alu).mostraCurs();

    }
}
```

Encara que *per_alu* estiga instanciat a *Alumne*, com està definit com a *Persona*, el compilador NO ens deixa accedir a les particularitats d'*Alumne*. Per tant, això provocaria un error de compilació.

Sabem que en temps d'execució sí que funcionaria però no podem compilar.

Tenim el mateix problema amb el mètode *mostraCurs()*. Donaria error de compilació ja que eixe mètode no està en *Persona*.

La solució és aplicar càsting a l'objecte

Així, el programador està dient-li al compilador: "Fia't de mi: en *per_alu* hi ha un *Alumne* i, per tant, pots accedir al curs".

Igualment podrem accedir a *mostraCurs()* de la classe *Alumne*.

Però cal anar en compte, ja que el càsting pot donar error d'execució:

```
public class Principal {
    public static void main(String args[]){
        Alumne alu = new Alumne();
        mostra( alu );

        Profe pro = new Profe();
        mostra( pro );
    }

    static void mostra( Persona per ){
        ...
        ((Alumne) per).mostraCurs();
        ...
    }
}
```

Quan passem un profe al procediment, este intentarà fer el càsting a *Alumne* i donarà error, ja que en un profe no hi ha un alumne.

Ací és on donarà error d'execució si el que arriba al procediment en *per* no està instanciat a *Alumne*.

Solució a l'error del càsting:

Només farem el càsting si es pot fer. És a dir, només voldrem fer el càsting a *Alumne*, si l'objecte en qüestió està instanciat a *Alumne*. Distintes formes de fer-ho:

a) Amb l'operador *instanceof*

```
static void mostra( Persona per ){
    ...
    if (per instanceof Alumne) {
        ((Alumne) per).mostraCurs();
    }
    ...
}
```

b) Amb el mètode *getClass()*

```
static void mostra( Persona per ){
    ...
    if (per.getClass().getSimpleName().equals("Alumne")){
        ((Alumne) per).mostraCurs();
    }
    ...
}
```

c) Amb un *try-catch*

```
static void mostra( Persona per ){
    ...
    try {
        ((Alumne) per).mostraCurs();
    }
    catch(ClassCastException e) { }
    ...
}
```

Esquema resum dels distints tipus de polimorfisme:

	POLIMORFISME			
Tipus	Sobrecàrrega basada en paràmetres	Sobreescriptura (Override)	Sobrecàrrega basada en l'àmbit	Variables polimòrfiques
De mètode o de variable	Mètode	Mètode	Mètode	Variable
On està el polimorfisme?	En la mateixa classe	En classes distintes amb herència	En classes distintes sense herència	En objectes definits en certa classe però instanciats a una filla
Poden tindre els mateixos paràmetres?	No	Sí	Sí	-
Usos freqüents	- Classe amb més d'un constructor - Mètode que accepta diferents tipus de paràmetres	- Subclasses que volen modificar el comportament d'un mètode de la superclasse	- Mètodes amb un nom comú però que tenen poc a vore	- Llista que accepta objectes de distintes subclasses - Mètode que accepte objectes de distintes subclasses
Quan s'invoca un mètode, quin s'executa?	El que concorda amb els paràmetres d'entrada	El de la classe de l'objecte que fa la crida. Si no existeix, el de la superclasse.	El de la classe de l'objecte que fa la crida	- 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).
Se sap a qui invoca en temps de...	Compilació	Compilació	Compilació	Execució

Exercicis sobre polimorfisme

19. El mètode *mostrarDades()* té polimorfisme de sobreescriptura, ja que està implementat en distintes classes amb relació d'herència (*Cotxe* i *Cotxe2aMa*). Modifica el mètode *mostrarDades()* de la subclasse: fes ús de "*super.*" per a accedir al *mostrarDades()* de la superclasse.

20. 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.

21. En el programa principal:

    a) Crea un cotxe i fes diverses crides al mètode *revisar()* de forma que s'executen les 2 implementacions d'eixe mètode.

    b) 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.

    c) Dins la classe principal (la quet é el *main()*) crea el mètode *mostrarKms()* de forma que accepte com a paràmetre un cotxe de segona mà o normal (per tant, el paràmetre serà de la classe pare: *Cotxe*) i que mostre 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.

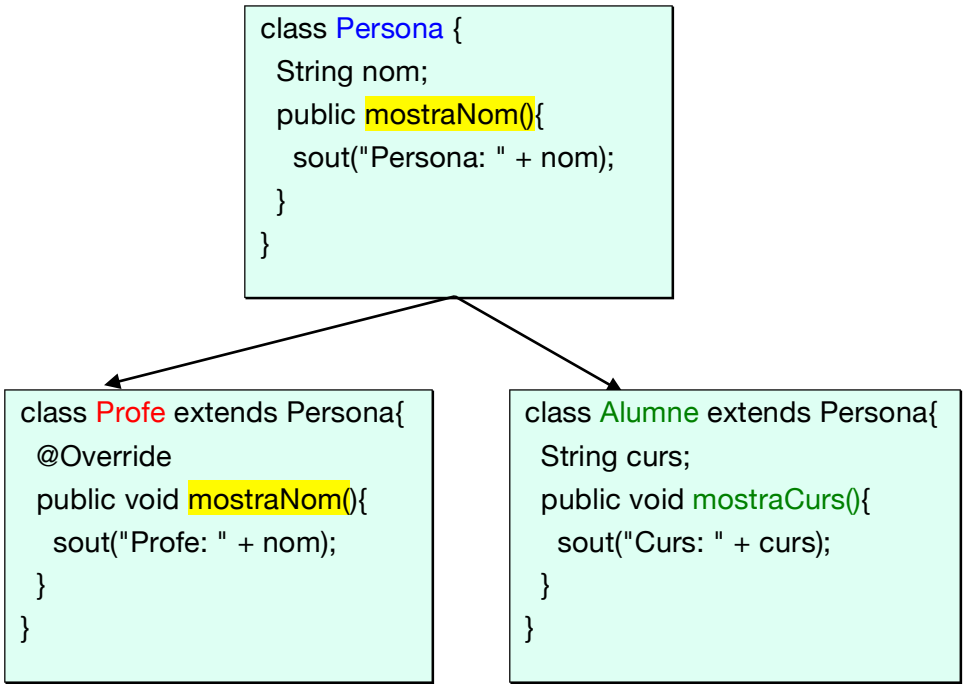
    d) Crida al mètode anterior passant-li com a paràmetre un objecte de la classe *Cotxe2aMa* i una altra crida passant-li un objecte de la classe *Cotxe*.

        - 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 mostre els kms; si no, que mostre el text “Cotxe nou, amb 0 kms”. Prova diverses solucions (*instanceof*, *getClass* o *try-catch*).

22. Llista de persones de l'institut.

a) Crea l'aplicació *PersonesInstitut*, amb les següents classes:



b) En el main crea una llista (array o ArrayList) de 100 persones, on guardarem tot el personal de l'institut (profes, alumnes o qualsevol altre tipus de persones).

c) Assigna un professor en el 1r element de la llista; un alumne en el 2n; i una persona en el 3r. Posa dades qualsevol a les 3 persones.

d) Utilitzant el mètode *mostraNom(),* mostra els 3 objectes que tens (usa un for). Executa-ho per comprovar a quina implementació de *mostrarNom()* s'ha invocat en cada cas.

e) Amb un altre for, mostrar els cursos dels 3 objectes de la llista (invocant el mètode *mostraCurs()*. Et donarà error de compilació ja que la llista és de persones però la classe Persona no té eixe mètode, sinó que és només de la classe *Alumne*. Fes ús del càsting per a solucionar-ho.

f) Executa el programa. T'eixirà un error quan intente fer el càsting del professor, ja que un *Professor* no pot "convertir-se" en *Alumne*. Arregla-ho per a que no done eixe error: fes que només es cride a *mostraCurs()* si l'objecte de la llista que s'està tractant és realment un *Alumne* (usa *instanceof*, *getClass* o *try-catch*).

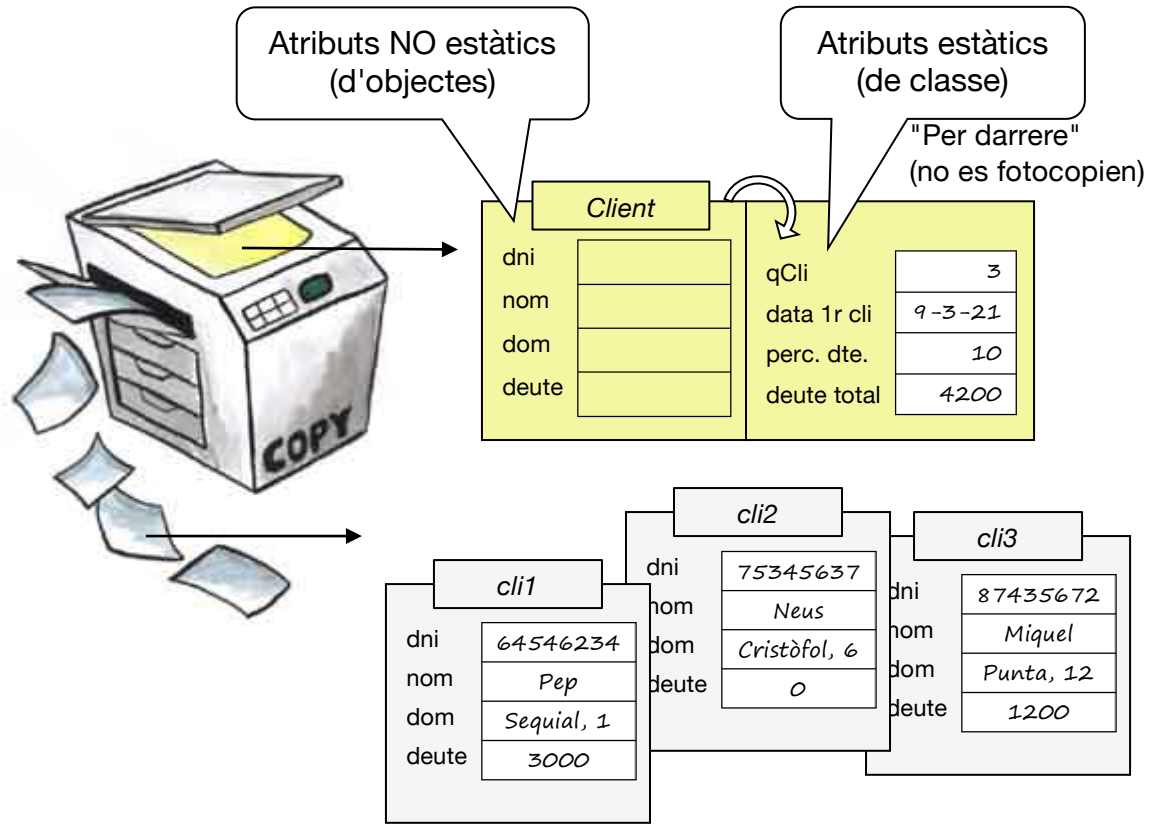
8 Atributs i mètodes estàtics

8.1. Atributs estàtics

Els atributs que hem vist fins ara en una classe es diuen "atributs d'objecte" però també hi ha els "atributs de classe" o "estàtics".

- Atributs d'objecte. Són els que hem vist fins ara. Un atribut d'objecte pot tindre **un valor diferent per cada objecte** de la classe.
- Atributs de classe (estàtics). Un atribut de classe o estàtic té només **un únic valor**, independentment dels objectes que es creen de la classe.

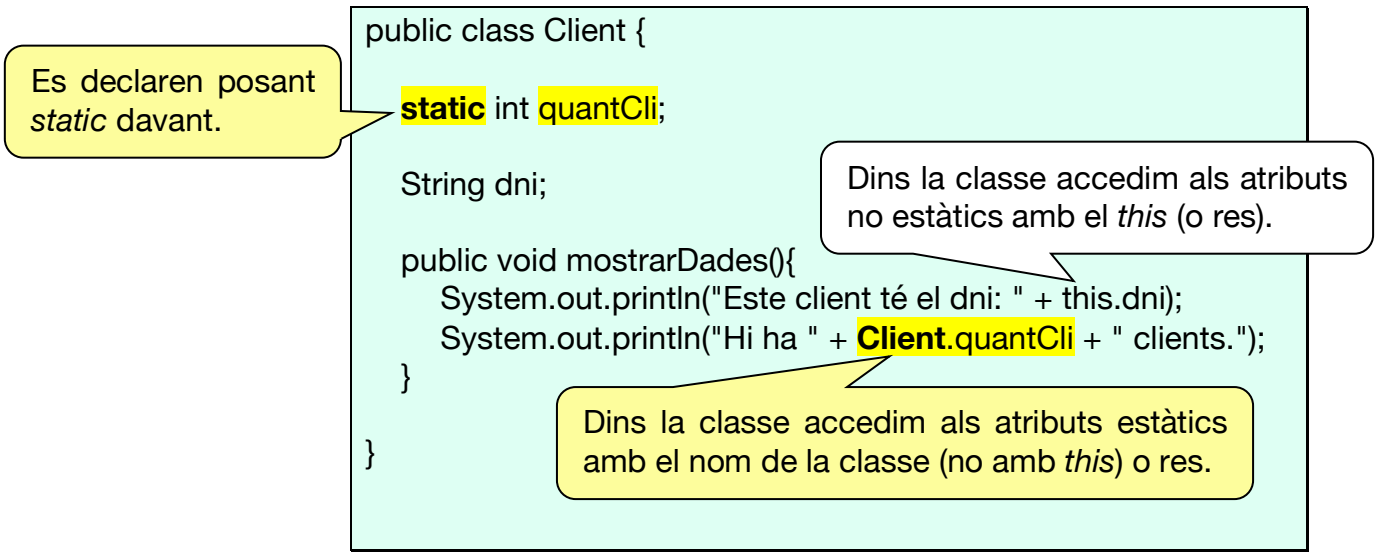
Si recordem l'analogia de classe i objectes amb el full original i les fotocòpies, ara podem dir que els atributs estàtics estarien a la part de darrere del full original i, per tant, no es fotocopien. És a dir, pertanyen només a la classe, no als objectes.

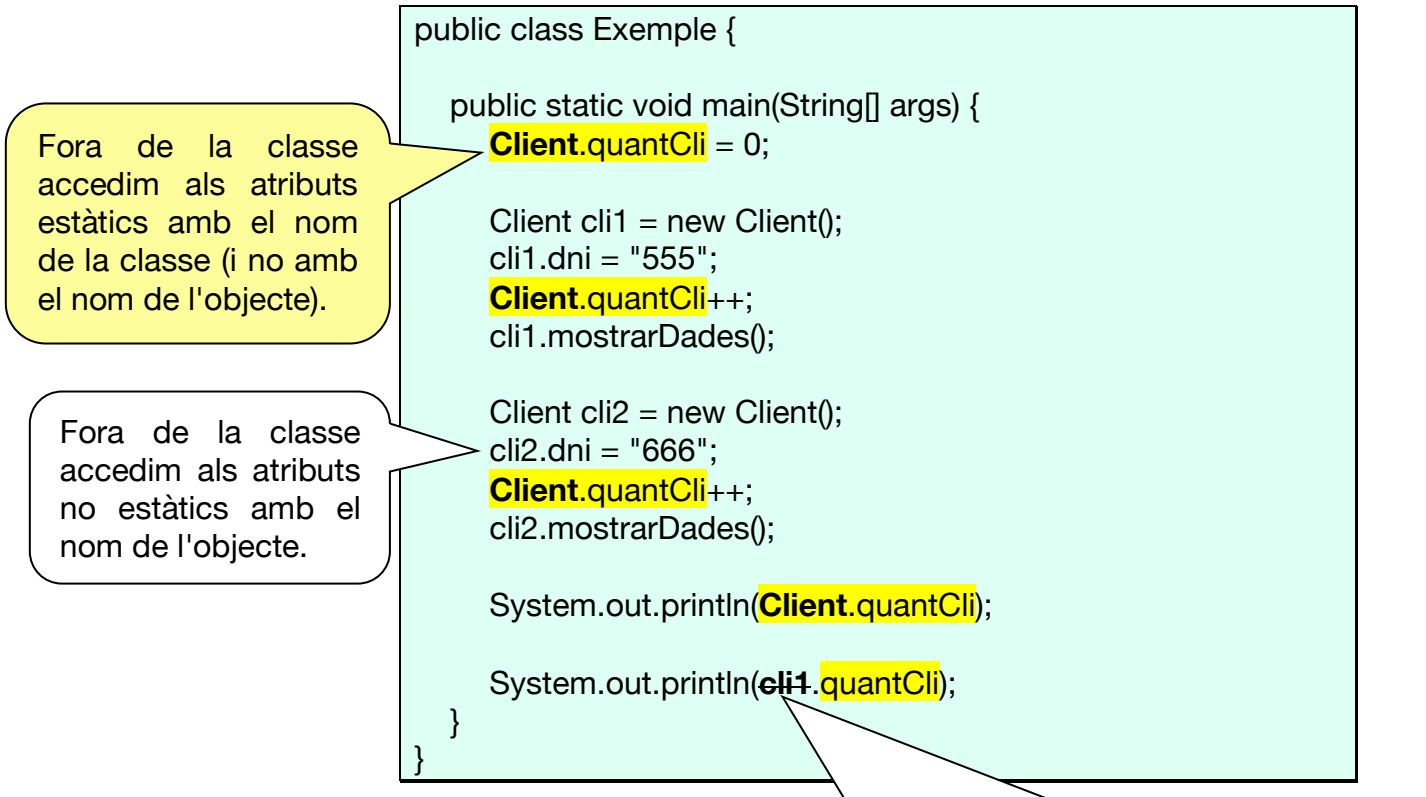


Exemples d'atributs estàtics: la quantitat de clients creats (quantes fotocòpies), la data del 1r client, el percentatge de descompte (igual per a tots els clients), el deute total dels clients...

Declaració i ús dels atributs estàtics

Veiem-ho amb un exemple:





Un atribut estàtic també pot invocar-se amb el nom d'un objecte (*cli1.quantCli*), encara que no té massa sentit i, per tant, no és aconsellable.

El resultat serà el mateix però el compilador mostrarà amb un avís dient que estem accedint a un atribut estàtic a partir d'un objecte.

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 sobre atributs estàtics

23. Afig 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*).

24. Modifica la classe *Cotxe* per a que s'actualitze el nou atribut *quantitat* quan calga. Com que és un comptador, caldrà inicialitzar-lo i incrementar-lo.

25. Fes proves creant diversos cotxes, també de segona mà. Posteriorment mostra el valor de l'atribut *quantitat* per comprovar que funciona correctament.

8.2. Mètodes estàtics

Són les funcions i procediments que usàvem abans de vore la POO (recorda que els posàvem *static* davant).

Els mètodes estàtics pertanyen a tota la classe, no a cap objecte en particular. Per això s'invoquen sense posar el nom de cap objecte davant.

Declaració i invocació de mètodes estàtics

```
public class Client {
    // --- Atributs ---

    private static int quantCli;
    String dni;


    // --- Mètodes ---

    String getDni(){
            return this.dni;
    }

    static int quantsClients(){
            return this.quantCli;
    }

}
```

Mètode no estàtic: s'aplica a un client en concret, i accedix als seus atributs amb el *this*.

Es declaren posant *static* davant.

Mètode estàtic: s'aplica a tots els clients en general. No pot fer ús del *this*.

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

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

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

Utilitats dels mètodes estàtics

Alguns exemples (que ja havíem usat abans de vore la POO) podrien ser:

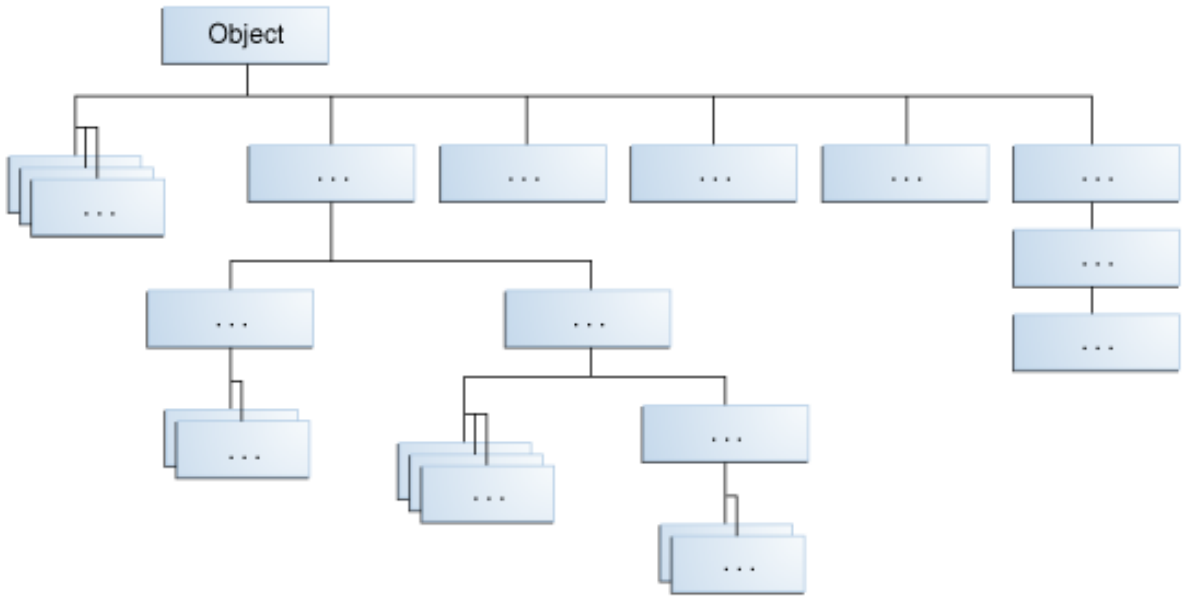
```
public class Principal {
   public static void main(String [] args) {

        // Mètodes nostres:

        float f = areaTriangle(10, 5);          // Càlcul àrea d'un triangle

        imprimir(vectorNotes);                  // Imprimir un vector d'enters

        int j = llegirEnter();                  // Llegir un enter de teclat


        // Mètodes existents en Java:

        System.out.println("Hola, món!");       // Imprimir un text en pantalla

        int i = Integer.parseInt("10");         // Convertir una cadena a un enter
   }
}
```

Exercicis sobre mètodes estàtics

26. 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*.

9. Mètodes de la classe Object

En Java hi ha moltes classes existents. De fet, hi ha una jerarquia de classes on, la classe mare de totes les classes és ***Object***.



Algunes classes que hem vist també estan en eixa jerarquia: *String, Integer, Float, ArrayList, Exception, ArrayIndexOutOfBoundsException, NullPointerException...*

I les classes que defineix el programador també són filles d'*Object* (encara que no posem l'*extends*.

La classe Object té diversos mètodes. En vorem dos que són molt importants ja que ens serviran de molta ajuda en les nostres classes:

- El mètode **toString()** : retornarà l'estat de l'objecte en forma de String
- El mètode **equals(...)** : retornarà un booleà per vore si 2 objectes són iguals.

Però no ens servirà de molt en les nostres classes si no els sobreescrivim posant el codi al nostre gust.

Anem a vore els 2 mètodes.

9.1. El mètode *toString()*

Serveix per a retornar en una cadena les dades d'un objecte. Com hem dit abans, és un mètode de la classe *Object*, per tant, sense implementar-lo el podem usar, però no ens servirà de molt, ja que ens retornarà l'adreça de memòria (en hexadecimal) d'on està l'objecte.

```
System.out.println( alu1.toString() );
```

Ens mostrarà alguna cosa com:
Alumne@7852e922

Per tant, hem de sobreescriure eixe mètode en la nostra classe i posar els atributs que volem i en el format que volem. Per exemple:

```
class Alumne {
    String nom;
    int edat;

    @Override
    public String toString() {
        return "(" + "Nom: " + nom + ", Edat: " + edat + ")";
    }
}
```

Netbeans ens pot generar el *toString()* automàticament:
- Clic dret dins la classe
- Insert Code --> **toString()**

```
System.out.println( alu1.toString() );
```

Ara ens mostrarà les dades d'*alu1*:
(Nom: Pep, Edat: 50)

De fet, en la crida a *println*, no cal ni posar el *toString()* ja que si el *println* rep un objecte, crida automàticament al *toString* d'eixe objecte.

```
System.out.println( alu1 );
```

(Nom: Pep, Edat: 50)

La classe *ArrayList* també té la seua implementació del *toString*, de forma que si fem un *println* d'un *ArrayList* d'alumnes, el *println* retornarà una cadena amb la concatenació dels resultats dels *toString* de cada alumne de l'*ArrayList*:

```
ArrayList <Alumne> alumnes = new ArrayList();
...
System.out.println( alumnes );
```

Mostrarà: [(Nom: Pep, Edat: 50), (Nom: Pepa, Edat: 20), (Nom: Pepet, Edat: 30)]

Exercicis sobre el mètode *toString()*

27. El *toString*() de la classe *Cotxe*

a) Crea el *toString()* de la classe *Cotxe* per a que retorne una cadena amb este format:

```
Ford Festa      Matrícula: 3316-FP
---------------------------------------
   Bastidor: 9726433453
   Color: roig obscur
   Preu: 12000
   Revisions: 1:sí  2:sí  3:sí  4:no  5:no
```

b) Fes crides des del *main* per a mostrar amb este format les dades de 2 cotxes. En un d'ells enviant a *println* la cadena que retorna el *toString()*; en l'altre envia a *println* el cotxe directament.

c) Mostra les dades de l'array de *cotxesAparador*, amb este format. Caldrà recórrer el vector.

d) Crea un ArrayList de cotxes. Posa'n alguns i mostra'ls tots, sense cap bucle per a recórrer l'ArrayList.

28. El *toString()* de la classe *Cotxe2aMa*

a) Crea el *toString(...)* de la classe *Cotxe2aMa* per tal que retorne una cadena amb totes les seues dades (també els kms). Caldrà invocar el *toString(...)* de *Cotxe* (no faces copiar-i-apegar).

b) Mostra les dades del *cotxeAntic*, amb este format.

c) Afig a l'ArrayList de cotxes el *cotxeAntic* i torna a mostrar tots els cotxes de la llista. Fixa't que per a cada cotxe de la llista l'intèrpret de Java ha sabut invocar al *toString(...)* de *Cotxe* o al de *Cotxe2aMa*.

9.2. Els mètodes *equals(...)* i *hashCode()*

El mètode equals()

Com ja sabem, per saber si 2 objectes són iguals no podem usar == ja que això només ens diu si apunten a la mateixa adreça de memòria.

Per tant, usarem el mètode ***equals(...)*** de la classe *Object*.

Exemple d'ús:

```
Alumne alu1 = new Alumne();
Alumne alu2 = new Alumne();
...

if (alu1.equals(alu2)) {...}
```

Comprova si *alu1* és **igual** a *alu2*

Ara bé: el mètode equals de la classe Object no pot saber quan 2 alumnes són iguals, i per això només diu que 2 objectes són iguals 2 si apunten a la mateixa zona de memòria.

Per tant, en la nostra classe Alumne haurem de sobreescriure el mètode equals amb les condicions que han de complir 2 alumnes per a ser considerats "iguals" (que tinguen el mateix nom, o que tinguen el mateix dni, o que tinguen el mateix dni, nom, edat...)

I no podríem dir-li d'una altra forma (en compte d'equals)? La resposta és no. És important que reimplemente el mètode *equals(...)* de la classe *Object*, ja que eixe mètode és invocat per certs mètodes d'algunes classes com l'ArrayList:

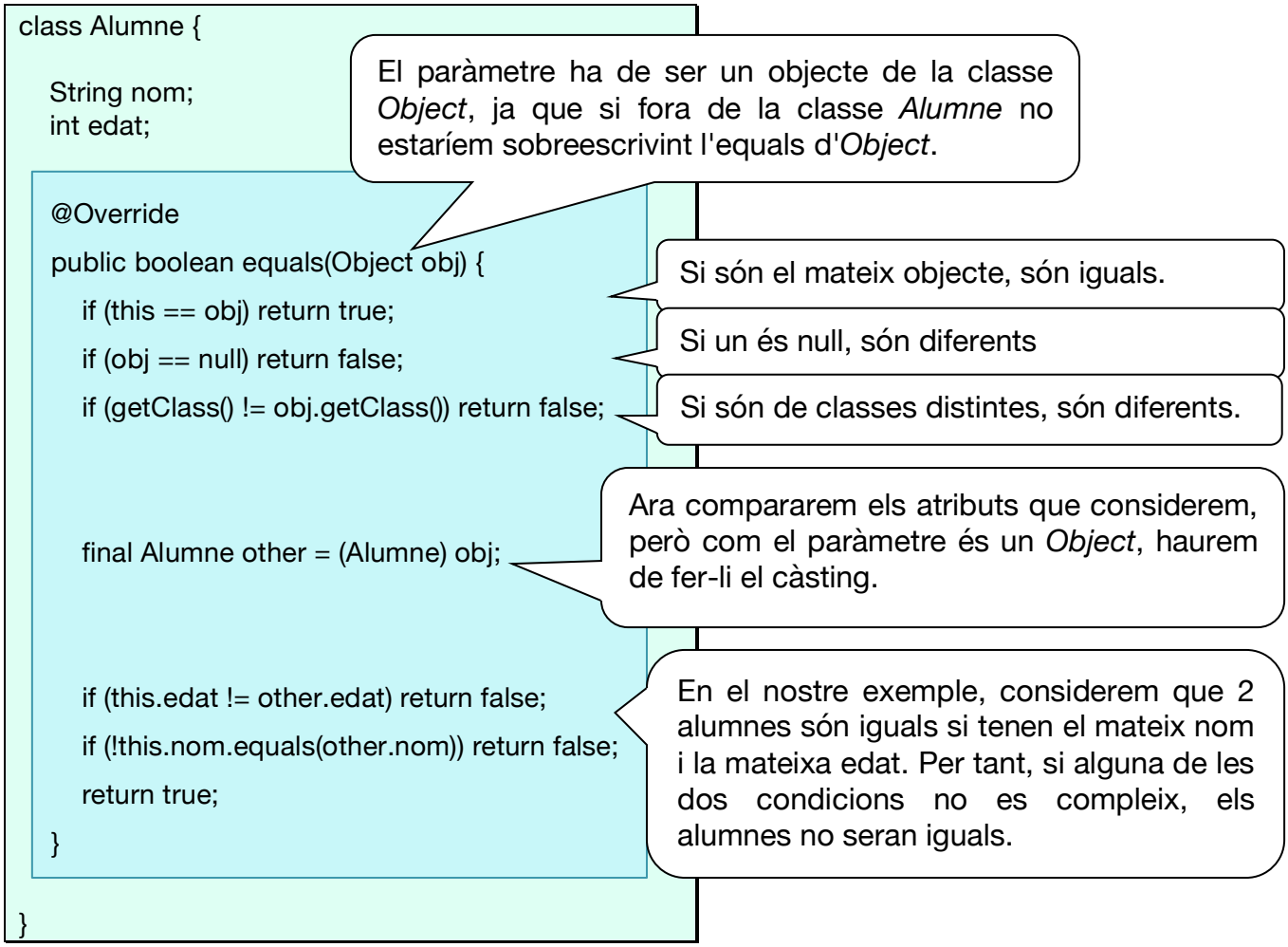
```
ArrayList <Alumne> alumnes = new ArrayList();
...

if (alumnes.contains(alu1)) {...}

int posicio = alumnes.indexOf(alu1);

alumnes.remove(alumne1);
```

Comprova si en la llista *alumnes* hi ha un alumne **igual** a *alu1*.

Retorna la posició de la llista *alumnes* on està el primer alumne **igual** a *alu1*.

Esborra de la llista d'alumnes el primer alumne que trobe que siga **igual** a alu1.

L'estructura d'un mètode *equals* sol ser la següent (Netbeans la fa automàticament si dins la classe fem clic dret i seleccionem *Insert Code*: *equals() and hashCode()*).



Exercicis sobre *equals(...)*

29. L'*equals()* de la classe *Cotxe.*

a) Fes el mètode equals() de la classe Cotxe tenint en compte que 2 cotxes són iguals si tenen la mateixa matrícula o si tenen el mateix número de bastidor.

b) Crea 2 cotxes amb les mateixes dades i comprova si són iguals.

c) Posa un d'eixos cotxes en l'ArrayList i comprova (amb el *contains*) si eixe cotxe està o no en la llista.

30. Fes l'*equals()* de *Cotxe2aMa* de forma que 2 cotxes de 2a mà són iguals si, a més de tindre el mateix bastidor o matrícula, tenen els mateixos kms. L'equals de *Cotxe2aMa* haurà d'invocar l'equals de *Cotxe*. Comprova des del *main* que 2 cotxes de 2a mà són iguals.

El mètode *hashCode()*

Este mètode complementa l'*equals(...)*, i serveix per a buscar elements iguals en estructures tipus *HashSet* o *HashMap* de forma més eficient. Per tant, només caldrà sobreescriure'l si volem usar eixes estructures (i estaríem obligats a fer-ho).

Com funciona el *hashCode*()? Per exemple, cada vegada que intentem inserir un element en un *HashSet*, el mètode *add* necessita saber si ja hi ha un objecte igual en la llista (ja que no poden estar repetits). Com ha de recórrer tots els elements, amb l'*equals()* aniria molt lent. La idea és associar a cada element un número i així tindre el *HashSet* ordenat per eixe número per fer la recerca més ràpida.

Exemple de *hashCode()* que genera Netbeans per a la nostra classe *Alumne*:

```
@Override
public int hashCode() {
    int hash = 3;
    hash = 53 * hash + this.nom.hashCode();
    hash = 53 * hash + this.edat;
    return hash;
}
```

No intentes entendre el per què de les fórmules. Es tracta de fer correspondre un número a les dades de l'objecte, de la millor forma possible per tal de minimitzar les col·lisions.

No passa res si hi ha col·lisions (que objectes diferents tinguen el mateix número de *hashCode()*) ja que, per a saber si un objecte és igual a un altre, primer es mira si els dos tenen el mateix número de *hashCode()*. Si és diferent, els objectes són diferents. Si és igual, podria ser de casualitat. En eixe cas, es cridaria a l'*equals(...)*.

Si no sobreescrivim el *hashCode()* d'*Object*, no funcionarien els mètodes *add*, *remove* i *contains* del *HashSet*, ja que el *hashCode()* de la classe *Object* retorna la posició de memòria (en sistema decimal) de l'objecte. Per tant, encara que els objectes tinguen les mateixes dades, sempre consideraria que els objectes són diferents (si no tenen la mateixa adreça de memòria).

10. La classe *ArrayList*

Al llarg del curs ja hem usat els ArrayList però els hem vist un poc per damunt. Ara els vorem més detalladament ja que, a més, ara tindrem ArrayList d'objectes, no només de tipus bàsics.

Com ja sabem, la classe ArrayList permet guardar dades en memòria de manera similar als arrays convencionals, però amb avantatges com:

- 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 eixa classe cal importar el paquet: java.util.ArrayList

10.1. Declaració d'un objecte ArrayList

```
ArrayList <nomClasse> nomLlista = new ArrayList();
```

- Podrà guardar elements d'eixa classe o d'una classe o de subclasses. Si no es posa <nomClasse> admetrà objectes de qualsevol classe però, com vorem, després cadrà fer càstings per a accedir als atributs.
- 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> paisos = new ArrayList();
ArrayList <Integer> edats = new ArrayList();
ArrayList <Alumne> alumnes = new ArrayList();
ArrayList coses = new ArrayList();
```

10.2. Mètodes d'ArrayList

Mètode	boolean add (Object elementAInserir);
Descripció	Afig un element al final de la llista. La primera posició és la 0.

Exemples	**ArrayList <String> paisos = new ArrayList ();**
	paisos.add ("Alemanya"); // Ocupa la posició 0
	aisos.add ("França"); // Ocupa la posició 1
	aisos.add ("Portugal"); // Ocupa la posició 2
	
	ArrayList <Integer> edats = new ArrayList ();
	edats.add (22);
	int edat = 18;
	edats.add (edat);
	
	ArrayList <Cotxe> cotxes = new ArrayList();
	Cotxe cotxe1 = new Cotxe(“Ford”, “Festa”, “V-3316-FP”);
	cotxes.add(cotxe1);
	cotxes.add(new Cotxe(“Seat”, “Panda”, “V-1023-AA”));
	
	ArrayList coses = new ArrayList();
	llistaCoses.add(“Alemanya”);
	llistaCoses.add(22);
	llistaCoses.add(new Cotxe(“Seat”, “Panda”, “V-1023-AA”));

Mètode	void add (int posició, Object elementAInserir);
Descripció	Inserix un element en la posició indicada, desplaçant l'element que es trobava en aquesta posició, i tots els següents, una posició més. Si s'intenta inserir en una posició que no existeix, es produeix l’excepció IndexOutOfBoundsException.

Exemples	// Alemanya, França, Portugal
	paisos.add (1, "Itàlia");
	// Alemanya, **Itàlia**, França, Portugal

Mètode	Object set (int posició, Object elementQueSubstituirà);
Descripció	Canvia l'element de la posició del 1r paràmetre per l'objecte del 2n paràmetre. Retorna l'objecte que s'ha llevat.

Exemples	// Alemanya, Itàlia, França, Portugal
	String paisAnterior = paisos.set (1, "Andorra");
	// Alemanya, Andorra, França, Portugal
	// paisAnterior --> Itàlia

Mètode	Object remove (int posició)
Descripció	

Exemples	// Alemanya, Itàlia, França, Portugal
	String paisEsborrat = paisos.remove(2);
	// Alemaya, Itàlia, Portugal
	
	ArrayList <Cotxe> cotxes = new ArrayList();
	...
	Cotxe primerCotxe = cotxes.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. S'explica més detallat en el mètode *get*.

Mètode	boolean remove (Object elementASuprimir)
Descripció	Retorna false si no estava.

Exemples	// Alemaya, Itàlia, Portugal
	paisos.remove("Portugal");
	// Alemanya, Itàlia

Mètode	Object get (int posició)
Descripció	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).

Exemples	// Alemanya, Itàlia
	System.out.println(paisos.get(1));
	// Mostrarà: Itàlia
	
	// Veiem la diferència segons hem declarat l'ArrayList:
	---------- 	
	ArrayList cotxes = new ArrayList();
	...
	Cotxe c = **(Cotxe)** cotxes.get(0);
	// 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.
	---------- 	
	ArrayList **<Cotxe>** cotxes = new ArrayList();
	...
	Cotxe c = cotxes.get(0);
	//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.

Mètode	int indexOf (Object elementBuscat)
Descripció	Retorna la primera posició que ocupa l'element que s'indique per paràmetre. Si no està, retorna -1. Realment busca un element igual (equals) a eixe. El mètode *lastIndexOf* fa el mateix però retorna l'últim element trobat igual.

Exemples	...
	String paisBuscat = "França";
	int pos = paisos.indexOf(paisBuscat);
	if (pos != -1)
	System.out.println (paisBuscat + " s'ha trobat en: " + pos);
	else
	System.out.println (paisBuscat + " no s'ha trobat");

Mètode	boolean contains(Object element):
Descripció	Retorna true si es troba l'element indicat a la llista, i false en cas contrari.
Exemples	if (cotxes.contains(cotxeMeu)) { sout("Sí que està"); }

Mètode	void clear()
Descripció	Esborra la llista
Exemples	cotxes.clear();

Mètode	boolean isEmpty()
Descripció	Retorna true si la llista és buida.
Exemples	if (cotxes.isEmpty()) { sout("Llista buida"); }

Mètode	Object clone()
Descripció	Retorna una còpia exacta de la llista (com a ArrayList). Però els elements no són copiats (les respectives posicions apunten als mateixos objectes). I com el *clone()* retorna un *Object*, cal fer el càsting:
Exemples	Object clone(): llistaCotxes2 = (ArrayList<Cotxe>)llistaCotxes1.clone();

Mètode	Object [] toArray()
Descripció	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:

Exemples	Object [] vectorCotxes = cotxes.toArray();
	for (Object obj: vectorCotxes){
	((Cotxe)obj).mostrarNom();
	}

Mètode	int size()
Descripció	Retorna la quantitat d'elements que té l'ArrayList.

Exemples	// Podem aprofitar el size per a recórrer l'ArrayList:
	for (int i = 0; i < cotxes.**size()** ; i + +) {
	System.out.println(cotxes.get(i).toString());
	}
	
	// També podem usar un foreach per a recórrer-lo:
	for (String cotxe: cotxes) {
	System.out.println(cotxe.toString());
	}

Exercicis

31. 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.

32. 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:
    - constructor amb paràmetres: dni, nom, cognoms, edat, poble
    - gets i sets
    - 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:
    - constructor amb paràmetres: codi, curs i cicle.
    - gets i sets de codi, curs i cicle (no de llistaAlumnes)
    - 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 convenients

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