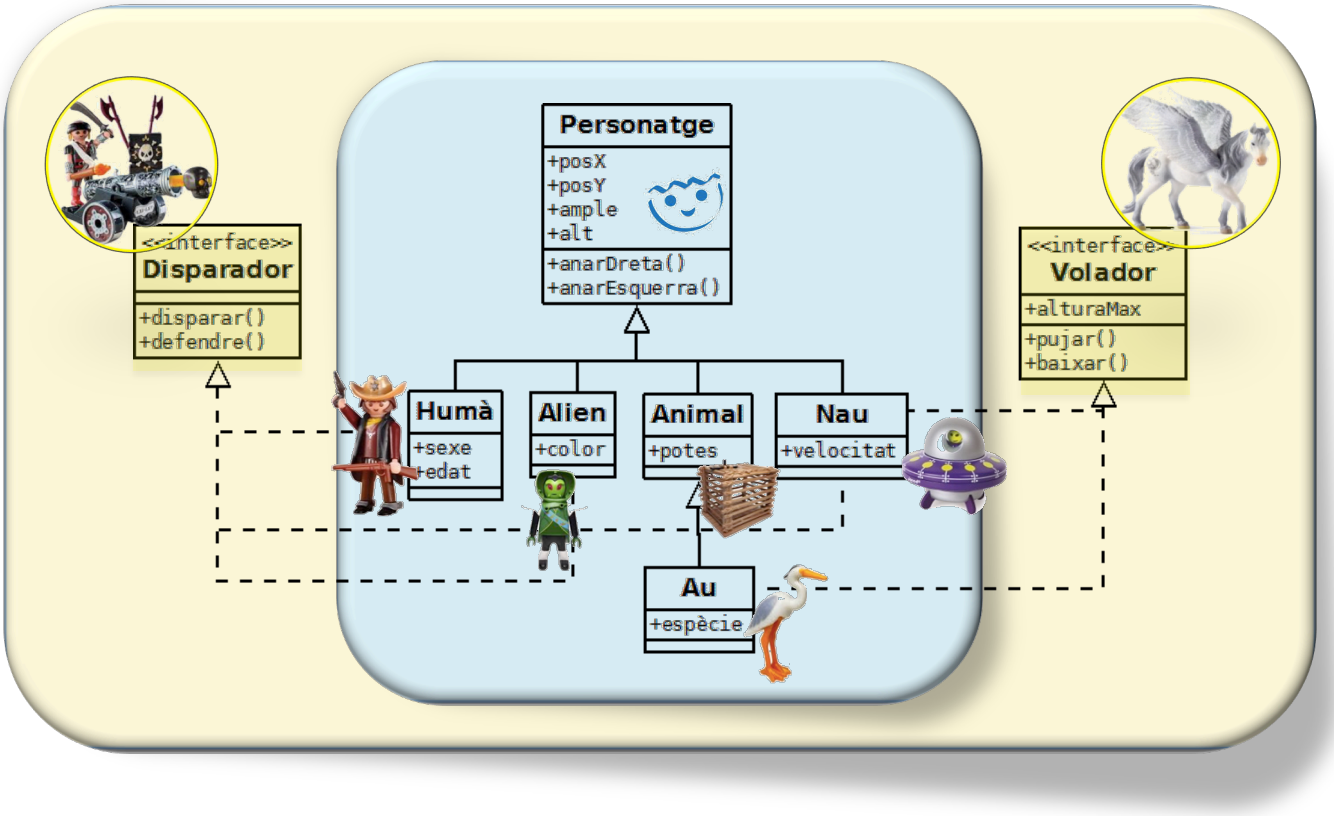
Tema 10

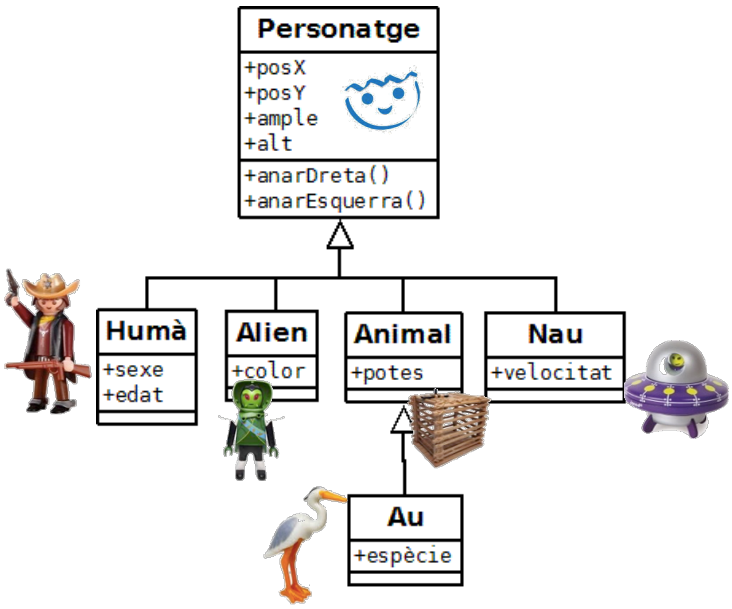
Interfícies



1 Introducció a les interfícies

En un videojoc de marcianets hi ha molts personatges (objectes) per la pantalla.

Tots seran de la classe *Personatge*, però són de tipus (subclasses) diferents.



Però ara volem completar-ho fent que algunes de les classes tinguen un **comportament en comú**.

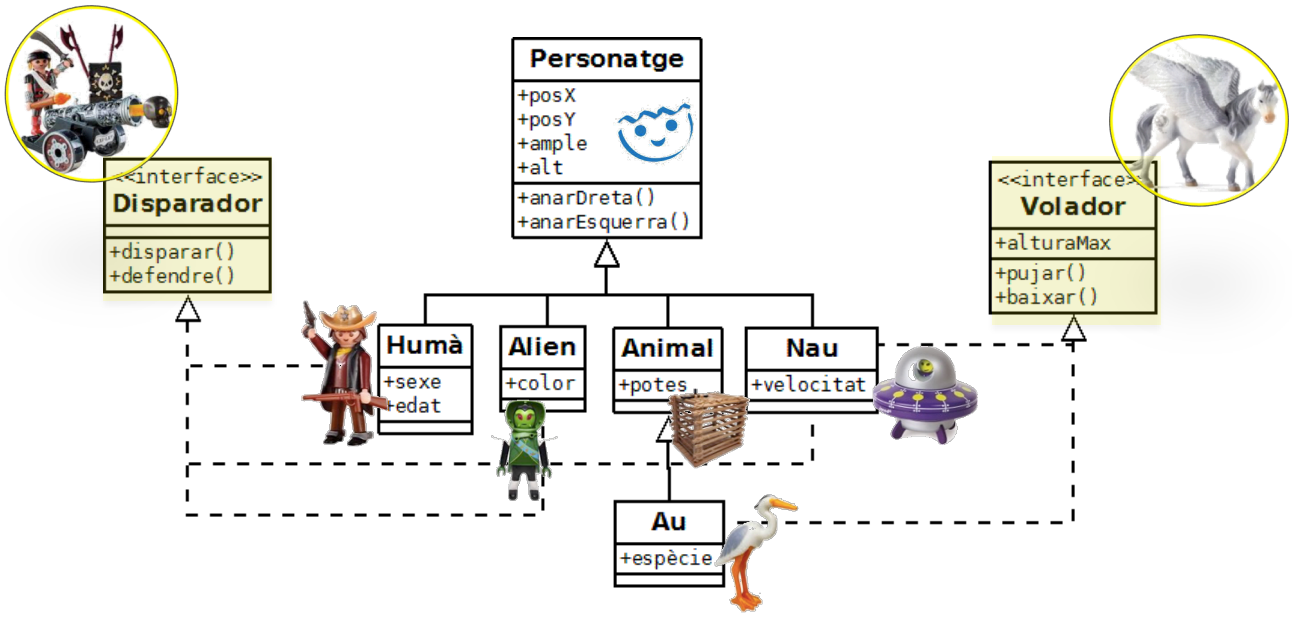
- ✓ Per exemple, si volem que la classe *Nau* i la classe *Au* puguen volar, voldrem que les dos implementen tots els mètodes que ha de tindre un personatge "volador": pujar i abaixar.

- ✓ O bé, si volem que les classes *Nau*, *Humà* i *Alien* puguen usar armes, voldrem que les tres implementen tots els mètodes que ha de tindre un personatge "disparador": *disparar* i *defendre*.

Podríem pensar en fer una classe *Volador* amb eixos mètodes, i altra *Disparador* amb els altres mètodes. Però en Java no podem fer que una classe siga filla de 2 classes. Per a això estan les interfícies.

Podríem dir que una **interfície** és un "conjunt d'operacions que sabem què han de fer però no com". Eixe "com" estarà definit en cada classe que vullga tindre eixe comportament. És a dir, en cada classe que implemente eixa interfície.

Veiem este exemple gràficament:



Veiem que caldrà definir una **interfície *Volador*** (i altra ***Disparador***), on posarem els mètodes que hauran d'implementar les classes "voladores" (i "disparadores", respectivament).

Els mètodes de les interfícies estaran "buits": només tindran la capçalera dels mètodes, però no estaran implementats. És a dir, seran mètodes abstractes, encara que no es posa la paraula *abstract*.

Veiem com es creen les interfícies:

```
public interface Disparador {

    void disparar();

    void defendre();
}
```

Els mètodes no s'implementen en la interfície, sinó que els haurà d'implementar cada classe que vullga implementar la interfície *Volador*.

Encara que no es pose, els mètodes són *public abstract*.

El nom de les interfícies convé que tinguen majúscula inicial (igua que les classes).

```
public interface Volador {

    int alturaMax = 100;

    void pujar();

    void baixar();
}
```

En una interfície també es poden definir **constants** (sempre seran *public static final*, encara que no es posa).

I veiem ara com una classe implementa eixes interfícies:

```
public class Nau extends Personatge implements Volador, Disparador{
   int velocitat;

   // ...

   // Implementació dels mètodes de Volador:
   @Override
   public void pujar() {
      this.posY += 3;
      if (this.posY > Volador.alturaMax) this.posY = Volador.alturaMax;
      this.velocitat++;
   }

   @Override
   public void baixar() {
      this.posY -= 3;
      if (this.posY < 0) this.posY = 0;
      this.velocitat--;
      if (this.velocitat < 0) this.velocitat = 0;
   }

   // implementació dels mètodes de Disparador:
   @Override
   public void disparar() {
      System.out.println("Pinyou, pinyou!");
   }

   @Override
   public void defendre() {
      System.out.println("Augh!");
   }
}
```

La classe *Nau* és filla de *Personatge*.

La classe *Nau* haurà d'implementar tots els mètodes de les interfícies *Volador* i *Disparador*.

Si no, donarà error de compilació.

En el main també podrem definir objectes "voladors" o "disparadors" però hauran de ser instanciats a una classe, mai a una interfície.

```
public class main {
  public static void main(String[] args){
     Nau n1 = new Nau();
     Volador v1 = new Nau();
     Volador v2 = new Au();
     ArrayList <Volador> llistaVoladors = new ArrayList();
     llistaVoladors.add(n1);
     llistaVoladors.add(v2);
     …
}
```

Puc definir un objecte a partir d'una interfície, però l'he d'instanciar a partir d'una classe.

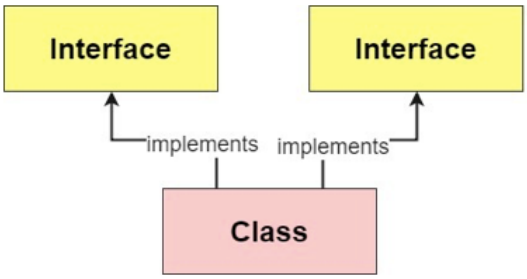
Igual que en les classes abstractes.

D'igual forma puc fer una llista de "voladors", on podré posar tant objectes *Nau* com *Au*.

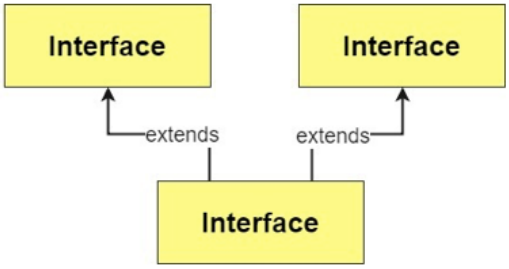
2 Concepte d'interfície

Una interficie és una col·lecció de mètodes abstractes i atributs constants. Les classes que implementen una interfície hauran de donar cos als mètodes de la interfície.

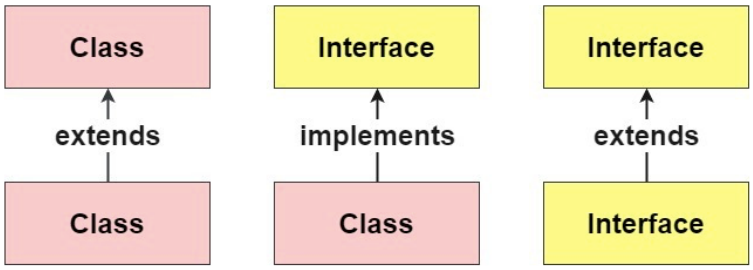
Una classe pot implementar moltes interfícies:



Una interfície tmbé pot ser filla (exteneds) d'altres interfícies:



Relacions entre classes i interfícies:



Utilitats de les interfícies

- Simular herència múltiple (ja que una classe pot tindre només una superclasse però pot implementar moltes interfícies).
- Obligar a que certes classes utilitzen els mateixos mètodes (noms i paràmetres) sense estar obligades a tindre una relació d'herència.
- Sabent que una classe implementa una determinada interfície, podrem usar els seus mètodes perquè ja sabrem què fan (ens dóna igual com estiguen implementats).
- Definir un conjunt de constants disponibles per a totes les classes que vullgam (implementant eixa interfície).

Mètodes per defecte en interfícies

A partir de Java 8 podem posar cos als mètodes de les interfícies.

En eixos casos caldrà indicar, amb *default*, que eixa és la implementació per defecte per a aquelles classes que implementen la interfície però no donen cos a eixos mètodes. Vegem-ho amb un exemple:

Les classes que implementen la interfície *Disparador* no estaran obligades a implementar el mètode *disparar()*, perquè ja n'hi ha un per defecte.

```
public interface Disparador {

    default void disparar(){
        System.out.println("Bang, bang!");
    }

    void defendre();
}
```

```
public class Nau implements Disparador{
    // ...

    @Override
    public void disparar() {
        System.out.println("Pinyou, pinyou!");
    }

    @Override
    public void defendre() {
        System.out.println("Augh!");
    }
}
```

La classe Nau implementa els 2 mètodes de la interfície.

```
public class Alien implements Disparador{
    // ...

    @Override
    public void disparar() {
        ...
    }

    @Override
    public void defendre() {
        System.out.println("Augh!");
    }
}
```

La classe *Alien* no implementa el mètode *disparar()*.

Per tant, tindrà la implementació per defecte que figura en la interficie.

3 Diferències entre interfícies i classes

3.1 Interfícies vs classes

	CLASSES	INTERFÍCIES

Quantitat de "pares" d'una classe.	Una classe només pot extendre (ser filla) d'**1 sola** classe:
	
	class Au extends **Animal**{
	...
	}
		Una classe pot implementar **moltes** interfícies:
		
		class Nau implements **Volador, Disparador** {
		...
		}

Es poden definir objectes d'eixe "tipus"?	**Sí, clar.**
	
	Nau enterprise;
	enterprise = new Nau();
		Sí, però no es pot instanciar.
		(igual que les classes abstractes):
		
		Volador ovni;
		ovni = new ~~Volador()~~;
		
		// Sí que podem fer:
		ovni = new Nau(); // new Au();
		
		ArrayList <Volador> voladors =
		new ArrayList<>();
		voladors.add(new Nau());
		voladors.add(new Au());

3.2 Interfícies vs classes abstractes

	CLASSES ABSTRACTES	INTERFÍCIES
Pot tindre mètodes amb cos?	Sí	Sí, des de de la versió 8 de Java, si poses davant "default".
Es poden definir atributs?	Sí	Només constants

Exercicis

1. Crea el projecte *ProvesInterficies*.
   a) Crea esta interficie:

```
public interface Estadistiques {
        double minim();
        double maxim();
        double suma();
}
```

   b) Crea la clase *ArrayListEstad* que tinga un ArrayList de doubles i que implemente la interfície *Estadístiques*.

> Nota: Quan poses l'implements, voràs que dona error. És degut a que encara no has impementant els mètodes de la interfície. Implementa'ls.

   c) Crea la clase *ArrayEstad*, amb un array (no ArrayList) de doubles, que implemente la interfície *Estadístiques*. Nota:

> Nota: Quan poses l'*implements*, clica en la icona de l'error i després en "*Implement all abstract methods*". Després posa el cos a eixos mètodes.

   d) En el *main* defineix un objecte de cadascuna de les classes anteriors, afegeix valors a les seues llistes i usa els mètodes de la interfície per a mostrar els respectius valors mínims, màxims i la suma dels seus elements.

2. Si volem que els objectes d'una classe es puguen comparar, eixa classe hauria de definir els mètodes *esMajor*, *esMenor* i *esIgual*. En compte d'implementar-los en eixa classe, l'objectiu és definir-los en una interfície i fer que cada classe que necessite poder comparar els seus objectes implemente eixa interfície.
   a) En el projecte Concessionari construeix la interfície *Comparar* amb els mètodes *esMajor*, *esMenor* i *esIgual*, als quals se'ls passa com a paràmetre un objecte i retornaran un booleà (true si *this* és major/menor/igual que l'objecte del paràmetre). Com sabràs, en la interfície no s'implementen, sinó que només es posa la capçalera dels mètodes. Posa també un comentari de què fa cada mètode.
   b) Fes que la classe *Cotxe* implemente la interfície *Comparar*. Un cotxe serà menor que un altre si el preu és menor.
   c) En el *main* mostra el més gran de 2 cotxes qualssevol.

4 Les interfícies *Comparable* i *Comparator*

4.1 Introducció

Per a ordenar un array d'enters, podem usar el mètode *sort* de la classe *Arrays* (caldrà ipmortar *java.util.Arrays*):

```
int [] edats = {4,7,3,6,9};

Arrays.sort(edats);

for (int e: edats){
        System.out.print(e + " ");
}
```

Si el que volem és ordenar un *ArrayList* d'enters podem usar el mètode *sort* de la classe *Collections* (caldrà importar *java.util.Collections*):

```
ArrayList<Integer> edats2 = new ArrayList();
edats2.add(4);
edats2.add(7);
...
Collections.sort(edats2);

System.out.println(edats2);
```

D'igual forma podríem ordenar una llista (*array*, *ArrayList*...) de *String*, de *float*, etc. Però què passa si volem ordenar una llista d'elements que no són directament ordenables (comparables), com pot ser una llista de cotxes, d'alumnes, etc? Si li aplicàrem el mètode *sort*, ens donaria error, ja que la màquina virtual "no sap" com comparar eixos objectes.

Per a fer que els objectes d'una classe puguen ser comparats, hem d'indicar un criteri de comparació. És a dir, cal definir quan un objecte de la classe que volem és menor que un altre, quan és major i quan és igual. La interfície *Comparar* que hem fet en l'exercici de l'apartat anterior no calia perquè Java ja té unes interfícies semblants, que són les que cal usar, ja que el *sort* (entre altres) usa els mètodes d'eixes interfícies.

Si volem establir un únic criteri d'ordenació, usarem la **interfície *Comparable*** però si volem establir diferents criteris d'ordenació usarem la **interfície *Comparator***.

4.2 La interfície *Comparable*

En Java ja existeixen interfícies fetes, com *Comparable*, per exemple.

L'han d'implementar les classes que vullguen establir un criteri de comparació dels seus objectes (i només un). L'únic mètode que cal implementar és *compareTo*:

```
interface Comparable <T>{
        int compareTo(T obj);
}
```

No l'hem de crear ja que és una interfície inclosa en Java.

Este mètode retorna un número negatiu, un zero o un número positiu, depenent de si *this* és menor, igual o major a *obj*. Ens servirà per a comparar dos objectes pel criteri que volem.

Exemple: suposem que volem comparar (o ordenar) alumnes. Si volem que l'ordre natural dels alumnes és pel seu codi, farem:

```
public class Alumne implements Comparable{
    int codi;
    String nom;
    int edat;
    String curs;

    @Override
    public int compareTo(Object obj) {
        return this.codi - ((Alumne)obj).codi;
    }
}
```

Amb la resta aconseguim que retorne:
- Un número negatiu si *this* < *obj*
- Un número positiu si *this* > *obj*
- O bé 0 si this == obj

O bé, per a no fer el càsting en el *compareTo*, podem implementar la interfície Comparable passant-li el tipus genèric. Així:

```
public class Alumne implements Comparable <Alumne> {
    int codi;
    String nom;
    int edat;
    String curs;

    @Override
    public int compareTo(Alumne alu) {
        return this.codi - alu.codi;
    }
}
```

Ara podrem comparar dos objectes de la classe *Alumne*:

```
if ( alume1.compareTo(alumne2) < 0 ) { ... }
```

O bé, ordenar una llista d'alumnes (*array*, *ArrayList*...) amb el *sort*, com abans:

```
ArrayList <Alumne> llistaAlumnes = new ArrayList();
llistaAlumnes.add( new Alumne(...) );
llistaAlumnes.add( new Alumne(...) );
...
Collections.sort(llistaAlumnes);
System.out.println(llistaAlumnes);
```

Ara el sort sí que sap com ordenar alumnes, ja que la classe *Alumne* implementa la interfície *Comparable*.

És un altre motiu de l'ús d'interfícies: mitjançant la implementació d'interfícies tots els programadors fan servir el mateix nom de mètode (*compareTo*, *sort*...) per a fer una mateixa acció (com comparar objectes).

Imagina't que estàs treballant en un equip de programadors i has d'utilitzar una classe que ha codificat un altre programador. Si vols comparar dos objetes d'eixa classe, només veient que implementa la interfície *Comparable*, ja saps quins mètodes pots usar sense saber com està implementat.

Això facilita el desenvolupament de programes i ajuda a comprendre'ls, sobretot quan intervenen centenars de classes diferents.

Exercicis

3. En l'exercici anterior has creat la interfície *Comparar* amb 3 mètodes però, com acabem de vore, ja existeix una interfície pareguda a l'API de Java: *Comparable*.
   a) Fes que la classe *Cotxe* implemente la interfície *Comparable*. Implementa el mètode *compareTo* fent que un *Cotxe* siga més xicotet que altre si la matrícula és menor alfabèticament (caldrà cridar al *compareTo* de la classe *String*). En cas d'igualtat, serà més xicotet el de menor preu.
   b) Modifica els mètodes de *Cotxe* que implementen la interfície *Comparar* per a que ara es basen en la definició del mètode *compareTo*.

4. En el *main* mostra una llista de cotxes (com tenim definit el *toString()* de *Cotxe* serà suficient amb *System.out.println(llistaCotxes)*. Ordena la llista i torna-la a mostrar.

4.3 La interfície *Comparator*

Amb la interfície *Comparable* podíem comparar (ordenar) alumnes per un criteri establert: el codi de l'alumne. Però a vegades, en un mateix programa, voldrem establir distints criteris d'ordenació: ara pel nom, ara pel curs, etc.

Per a fer que els objectes d'una classe puguen ser comparats per diversos criteris, per cada criteri caldrà crear un classe especial que implemente una interfície anomenada *Comparator*, on definirem el mètode *compare* (no *compareTo*), al qual se li passen com a paràmetre els dos objectes a comparar i retornarà un valor negatiu, zero o positiu (com en el mètode *compare* de *Comparator*).

```
interface Comparator <T>{
        int compare(T obj1, T obj2);
}
```

No l'hem de crear ja que és una interfície inclosa en Java.

Exemple. Volem establir 3 criteris d'ordenació. Per tant, farem 3 comparadors. És a dir, 3 classes que implementen *Comparator* per a implementar 3 voltes el *compare*:

```
import java.util.Comparator;
public class ComparadorPersonaPerEdat implements Comparator<Persona> {
    @Override
    public int compare(Persona p1, Persona p2){
        return p1.getedat() - p2.getEdat();
    }
}
```

```
import java.util.Comparator;
public class ComparadorPersonaPerNom implements Comparator<Persona> {
    @Override
    public int compare(Persona p1, Persona p2){
        return p1.getNom().compareTo(p2.getNom());
    }
}
```

```
import java.util.Comparator;
public class ComparadorPersonaPerCursEdat implements Comparator<Persona>{
    @Override
    public int compare(Persona p1, Persona p2){
        int cmpCurs = p1.getCurs() - p2.getCurs();
        return (cmpCurs != 0 ? cmpCurs : p1.getEdat() - p2.getEdat());
    }
}
```

Ara podrem comparar dos objectes de la classe *Alumne* pel criteri que vullgam. Ho farem creant un objecte de la classe que té el criteri que volem. Així:

```
if ((new ComparadorPersonaPerCursEdat()).compare(p1, p2) < 0) {
   System.out.println(p1 + " és menor que " + p2);
}
```

O bé, podrem ordenar una llista d'alumnes (*array*, *ArrayList*...) amb el criteri que vullgam, també amb el *sort*, però passant-li com a segon paràmetre un objecte de la classe que té el criteri que volem:

```
...
Collections.sort(llistaAlumnes, new ComparadorPersonaPerCursEdat() );
System.out.println(llistaAlumnes);
```

En este cas ens haurà ordenat la llista atenent al criteri de curs i edat.

Exercicis

5. Volem tindre una llista de factures i poder-les ordenar per diferents criteris.

    a) Crea la classe *Factura* amb els següents atributs:
        - *numero*: *int*
        - *data*: objecte de la classe *Date* (caldrà importar *java.util.Date*)
        - *import*: float

    b) Defineix tres classes que implementen diferents *Comparator* de factures:
        - Pel número de la factura
        - Per la data i, en cas d'igualtat, per l'import
        - Per l'import i, en cas d'igualtat, per la data

    c) En el programa principal
        - Crea dos factures i mostra la major segons el criteri de l'import (i, en cas d'igualtat, per la data).
        - Crea un *ArrayList* de factures, posa-li'n unes quantes i mostra tres voltes tota la llista, amb els diferents ordres que has definit abans.

5 La interfície *Cloneable*

Recordem que per a copiar un objecte a un altre no podem usar l'operador "igual" (signe =) ja que tindríem només un objecte però amb dos referències a ell. El que farem serà crear un mètode que ens retornarà un objecte idèntic on s'haurà copiat l'original atribut a atribut .

Sobreescrivim el *clone()* d'*Object*

El mètode per a clonar serà *clone()*, que sobreescriurà el *clone()* de la classe *Object*.

L'invocrem així:

```
Cotxe c1 = new Cotxe("Seat", 10);

Cotxe c2 = c1.clone();
```

És de la classe *Object* però estem obligats a sobrescriure'l ja que figura com a *protected* i no podem accedir des d'ací.

Java vol obligar-nos a sobreescriure'l per evitar les còpies superficials (ho vorem després).

I el definirem així:

```
public class Cotxe {
        String marca;
        String model;
        ...

        @Override
        public Cotxe clone() {
                Cotxe clon = new Cotxe();
                clon.marca = this.marca;
                clon.model = this.model;
                ...
                return clon;
        }
}
```

Retornem un nou cotxe amb el mateix estat.

O bé, si disposem d'un constructor adequat:

```
        @Override
        public Cotxe clone() {
                return new Cotxe(this.marca, this.model, ...);
        }
}
```

Retornem un nou cotxe amb el mateix estat.

Invocar del *clone()* d'*Object*

Si no volem fer totes eixes assignacions per a copiar atribut a atribut (ni cridar eixe constructor), el *clone()* de *Cotxe* pot invocar el *clone()* d'*Object*.

En eixe cas cal implementar la interfície *Cloneable* (simplement indicar-ho, sense implementar cap altre mètode). Veiem com seria:

```
public class Cotxe implements Cloneable{
        // ...

        @Override
        public Cotxe clone() {
                try {
                    return (Cotxe) super.clone();
                }
                catch (CloneNotSupportedException ex) {
                    return null;
                }
        }
}
```

Cal indicar que implementa *Cloneable*.

Cal fer casting.

Cal tractar amb *try-catch* la crida al *super.clone()*.

Si la classe no implementara *Cloneable* donaria eixa excepció.

La interfície *Cloneable* és especial perquè no té cap mètode. Per tant, per a què serveix?

Serveix per a que el programador siga conscient que amb la crida al *clone()* de la classe *Object* està fent una còpia camp per camp i, per tant, si algun d'eixos camps és un *array* o un objecte, estarà copiant adreces de memòria, no el contingut dels *arrays* i objectes.

La forma de clonar un objecte camp per camp (siga a mà o amb el *clone()* d'*Object*) s'anomena "**shallow copy**" o "**còpia superficial**". Per tant, caldrà fer una "**còpia en profunditat**" o "**deep copy**". Anem a vore-ho.

Suposem que volem clonar esta classe:

```
class Persona {
    String nom;
    int edat;
    int telefons[] = new int[3];
    Cotxe cotxe;
}
```

Si férem un *clone()* que invocara el *clone()* d'*Object*, no tindríem problema amb les propietats *nom* i *edat*, però les propietats *telefons* i *cotxe* de les 2 persones estarien apuntant a la mateixa zona de memòria. Caldrà fer una còpia en profunditat:

```
class Persona {
   String nom;
   int edat;
   int telefons[] = new int[3];
   Cotxe cotxe = new Cotxe();

   public Persona clone() {
      Persona clon = new Persona();

      clon.nom = this.nom;
      clon.edat = this.edat;

      clon.telefons = this.telefons;
      for (int i = 0; i < this.telefons.length; i++) {
            clon.telefons[i] = this.telefons[i];
      }

      clon.cotxe = this.cotxe;
      clon.cotxe = this.cotxe.clone();

      return clon;
}
```

Copia superficial dels atributs atòmics: copiem *nom* i *edat* amb =, o bé amb *super.clone()* (i, per tant, amb el *try-catch* i *implements Cloneable*).

Copia en profunditat dels arrays: copiem component a component l'array *telefons*.

Copia en profunditat dels objectes: invoquem al *clone()* de la classe *Cotxe (o* bé, si no existeix, copiem component a component els atributs del cotxe).

Exercicis

6. Volem clonar cotxes en el projecte del *Concessionari*.

   a) Crea el mètode *clone()* en la classe *Cotxe*.

   > Fes-ho com cregues convenient: copiant component a component i/o usant un constructor i/o usant el mètode *clone()* d'*Object* i/o còpia profunda (tin en compte que la classe *Cotxe* té un *array* de revisions).

   b) En el *main* crea un *cotxeA* i clona'l a un *cotxeB*. Mostra el *cotxeB*.

   c) Per vore que s'ha fet bé la còpia en profunditat, modifica alguna revisió del *cotxeA* i torna a mostrar el *cotxeB* per vore que no ha canviat.

7. Crea la classe *Propietari* amb un *nom* i un *cotxe*.

   a) Crea el *clone()* de *Propietari*.

   b) Clona 2 propietaris i comprova que s'ha fet la còpia en profunditat.

6 Les interfícies *Iterator* i *Iterable*

6.1 Introducció

Per a recórrer una llista es pot fer de diverses maneres. Hi ha dos formes de fer-ho sense usar un comptador: amb un bucle *foreach* o amb l'ajuda d'una interfície que vorem ara. Per exemple, per a una llista de *strings*:

Bucle *foreach*	Interfície *Iterator*

for (String nom: llista) {
  System.out.println(nom);
}
	Iterator <String> it = llista.iterator();
	while (it.hasNext()) {
	  System.out.println(it.next());
	}

Després ho explicarem en detall.

Veiem que el foreach és més senzill però si esborrem un element de la llista mentre la recorrem amb *foreach* pot donar error ja que és "com si ens furtaren les baldoses per on caminem". El mètode *remove()* d'Iterator ho resol. No el *remove()* propi de la llista, sinó el de la interfície *Iterator*:

Bucle for-each	Interfície Iterator

for (String nom: llista) {
  if (nom.equals("Pep"))
  llista.remove("Pep");
}
	Iterator<String> it = llista.iterator();
	while (it.hasNext()) {
	  String nom= it.next();
	  if (nom.equals("Pep"))
	  it.remove(); // ~~llista.remove("Pep");~~
	}

POT PROVOCAR L'EXCEPCIÓ: java.util.ConcurrentModificationException	NO PROVOCA EXCEPCIÓ

Per tant, usarem la interfície *Iterator* quan volem recórrer una llista si volem esborrar algun element mentre la recorrem.

Per a usar esta interfície cal importar *java.util.Iterator.*

6.2 La interfície Iterator

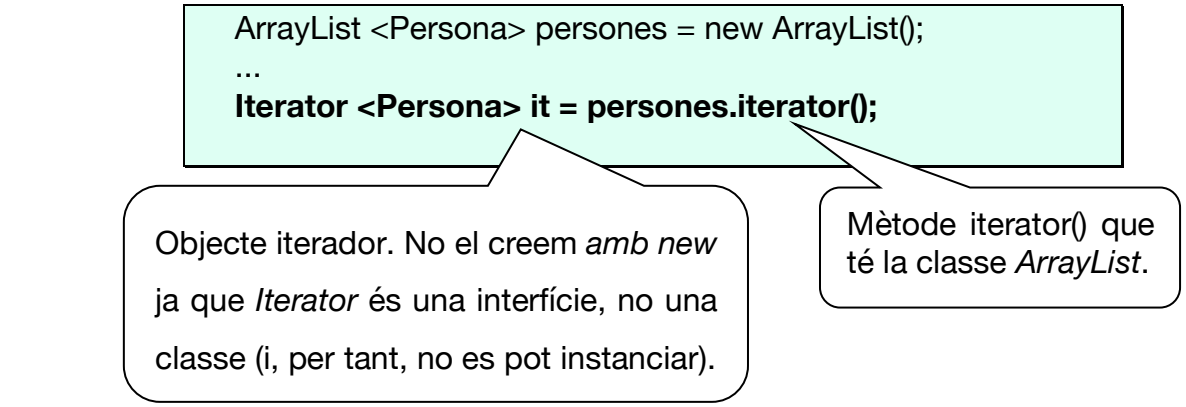
La interfície *Iterator* proporciona uns mètodes per a accedir seqüencialment als elements d'una col·lecció:

```
public Interface Iterator {
        boolean hasNext();    // Si hi ha altre element o no que recórrer.
        Object next();        // Retorna el següent element a recórrer.
        void remove();        // Elimina l'últim element obtingut amb el next().
}
```

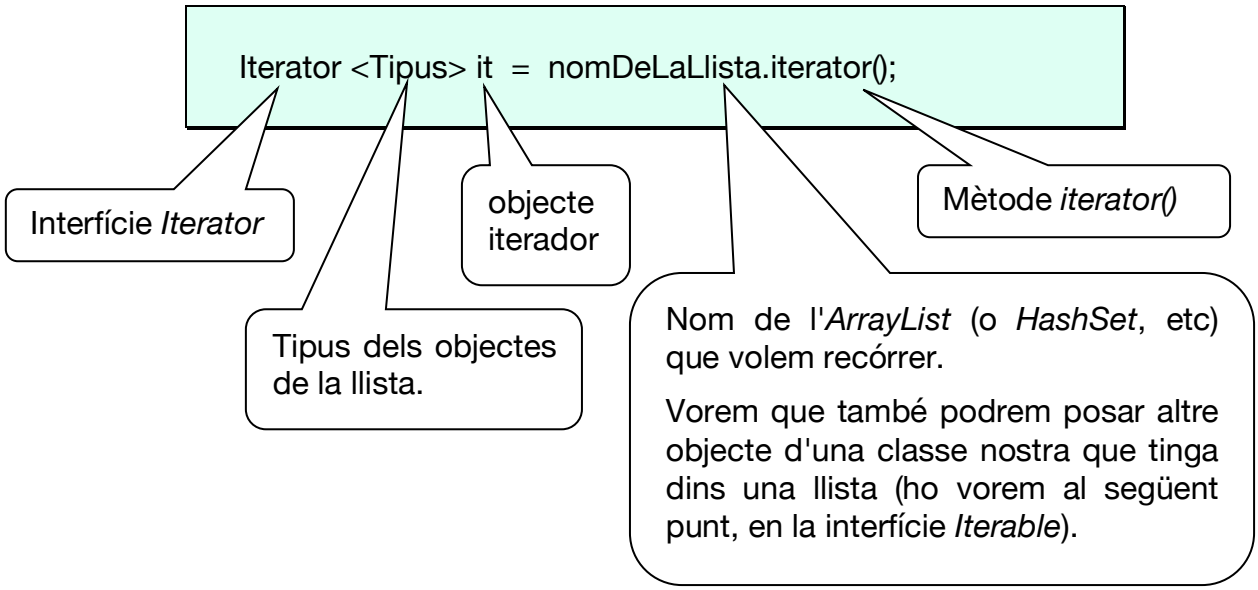
Com Java té moltes col·leccions distintes (*ArrayList, HashSet, array, Queue*...), es pretén recórrer de la mateixa forma eixes col·leccions (o a altres que ens fem nosaltres).

a) Primer cal definir-nos un iterador sobre la llista

Exemple:



Sintaxi:



b) Recorrem la llista amb l'iterador

```
ArrayList <Persona> persones = new ArrayList();
Iterator <Persona> it = persones.iterator();
...
Persona p;    // Objecte temporal
while (it.hasNext()){
        p = it.next();
        System.out.println(p.toString());
        if (p.getEdat() < 18) {
                System.out.println("És menor. L'esborrem");
                it.remove();  // Compte! No llista.remove()
        }
}
```

Compte! Una vegada hem obtingut l'iterador, NO s'ha de modificar la llista (amb *persones.add(…)* o *persones.remove(…)* abans de recórrer-la (o mentre la recorrem).

Sí que podrem esborrar un element però amb l'iterador (*it.remove()*).

És convenient fer ús d'eixe objecte temporal (p) per a no cometre este error:

```
while (it.hasNext()){

        if (it.next().getEdat() < 18)
                System.out.println(it.next().getNom() + " és menor");
        }

}
```

Ací avancem en la llista.

Ací tornaríem a avançar en la llista. Per tant, no tindríem el mateix element obtingut abans.

Exercicis

8. Recorre la llista de cotxes del concessionari de forma que en cada cotxe, després de mostrar les dades es preguntarà per teclat si volem esborrar eixe cotxe. En cas afirmatiu, l'esborrarem.

6.3 La interfície *Iterable*

Hem vist que amb la interfície *Iterator* podem recórrer els elements d'un *ArrayList*, d'un *HashMap*... ja que estes classes ens proporcionen un iterador:

```
ArrayList <Persona> persones = new ArrayList();
Iterator <Persona> it = persones.iterator();
```

Però també podem fer que una classe nostra, que incloga una llista d'elements, puga ser recorreguda amb un iterador. Per exemple, esta clase *Departament*:

```
class Departament {
    private String nom;
    private Empleat[] empleats = new Empleat[100];
    private int qEmpl = 0;
    Departament(String nom) {
        this.nom = nom;
    }
    public void add(String nomEmpleat, String carrec) {
        empleats[qEmpl++] = new Empleat(nomEmpleat, carrec);
    }
}
```

On *Empleat* serà una classe "normal":

```
public class Empleat {
    String nom;
    String carrec;
}
```

Per què volem un iterador? No podríem recórrer els empleats del departament com un array normal? Sí però es tracta que des de fora no tingam per què saber com està implementada eixa llista d'empleats (si en un *ArrayList*, un *array*...). A més, ens permetrà esborrar elements de la llista mentre la recorrem.

Per tant, des del *main*, voldríem fer açò:

```
public static void main(String [] args) {
        Departament dep = new Departament();
        ...
        Iterator <Empleat> it = dep.iterator();

        while (it.hasNext()){
                System.out.println(it.next())
        }
}
```

Volem que el departament ens proporcione un iterador per a recórrer els seus empleats.

Recorrem els empleats amb els mètodes d'*Iterator*, sense saber com està implementada la llista d'empleats.

Per a fer això, caldrà que eixa classe *Departament* siga "iterable". És a dir, que implemente la interfície anomenada *Iterable* per tal que ens retorne un iterador.

```
public Interface Iterable {
        Iterator<T> iterator();
}
```

Caldrà sobreescriure el mètode iterator() de la interfície *Iterable*.

La classe *Departament* tindrà una llista d'empleats, i implementarà *Iterable*: el mètode *iterator()* retornarà un iterador per a recórrer eixa llista.

```
import java.util.Iterator;

public class Departament implements Iterable {
    private String nom;
    private Empleat[] empleats = new Empleat[100];

    // ...

   @Override
   public Iterator<Empleat> iterator() {
      return new IteradorEmpleats();
   }

   protected class IteradorEmpleats implements Iterator<Empleat> {
      private int posicio = -1;

      @Override
      public boolean hasNext() {
         return posicio < qEmpl;
      }

      @Override
      public Empleat next() {
         return empleats[++posicio];
      }

      @Override
      public void remove() {
            if (posicio < qEmpl - 1) {
               System.arraycopy(empleats, posicio + 1,
                  empleats, posicio, qEmpl - posicio - 1);
            }
            qEmpl--;
      }
   }
}
```

El mètode *iterator()* retorna un objecte de la classe *IteradorEmpleats*.

Per tant, caldrà crear eixa classe, que implementarà la interfície *Iterator*.

Esta classe interna és la que implementa l'iterador.

I ara, com havíem dit, ara podrem recórrer la llista i esborrar elements:

```
import java.util.Iterator;
public class Programa {
   public static void main (String arg []) {
      Departament dep = new Departament(“Informàtica”);
      Iterator <Empleat> it;

      Empleat empl;
      dep.add(“Pep”, “no”);
      dep.add(“Pepa”, “programadora”);
      dep.add(“Pepet”, “no”);
      dep.add(“Pepeta”, “analista”);

      System.out.println ("Empleats del departament " + dep.getNom() + ":");
      System.out.println("(esborrarem els que no tinguen càrrec)");

      it = dep.iterator();
      while (it.hasNext()) {
         empl = it.next();
         System.out.println(empl);
         if (empl.carrec.equals(“no”)) {
            it.remove ();
         }
      }

      it = dep.iterator(); // Si volem tornar a recórrer la llista
      ...
   }
}
```

Resumint: per a implementar la interfície *Iterable* hem de sobreescriure el mètode *iterator(). Este mètode ha de retornar un objecte Iterator. I això ho aconseguim* creant una classe interna que implemente la interfície *Iterator*.

Exercicis

9. Llista d'alumnes.

   a) Crea la classe *Alumne* amb estos atributs i mètodes:
      - Constant *QAVA* (quantitat avaluacions): 3
      - Constant *QEXER* (quantitat d'exercicis per avaluació): 5
      - Matriu notes de *QAVA* avaluacions per *QEXER* exercicis.
      - Un mètode que permeta posar una nota a l'alumne (paràmetres: nota, núm. avaluació i núm. exercici).

   b) Volem recórrer la matriu per avaluacions, amb un iterador:
      - Fes que la classe *Alumne* implemente la interfície *Iterable*.
      - Implementa el mètode *iterator()* creant la subclasse corresponent. En el *remove()* simplement posarem la nota a -1.

   c) En la lasse principal, fes un bucle que mostre un menú amb les opcions:
      - Nou alumne
      - Posar una nota
      - Mostrar llista de notes d'un alumne (amb un iterador)
      - Eixir