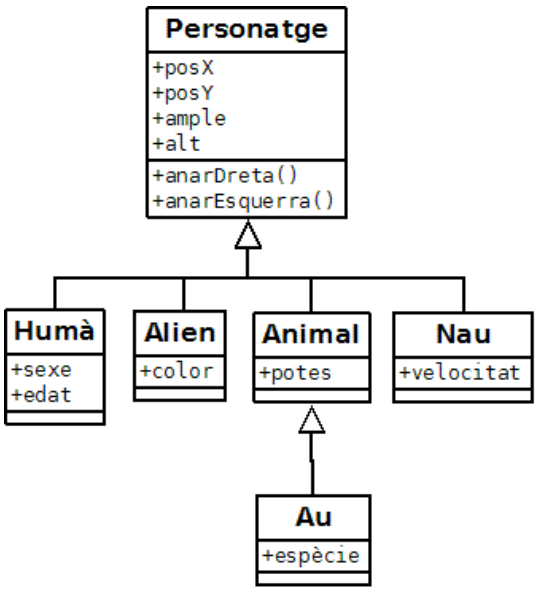
Tema 14

Interfícies

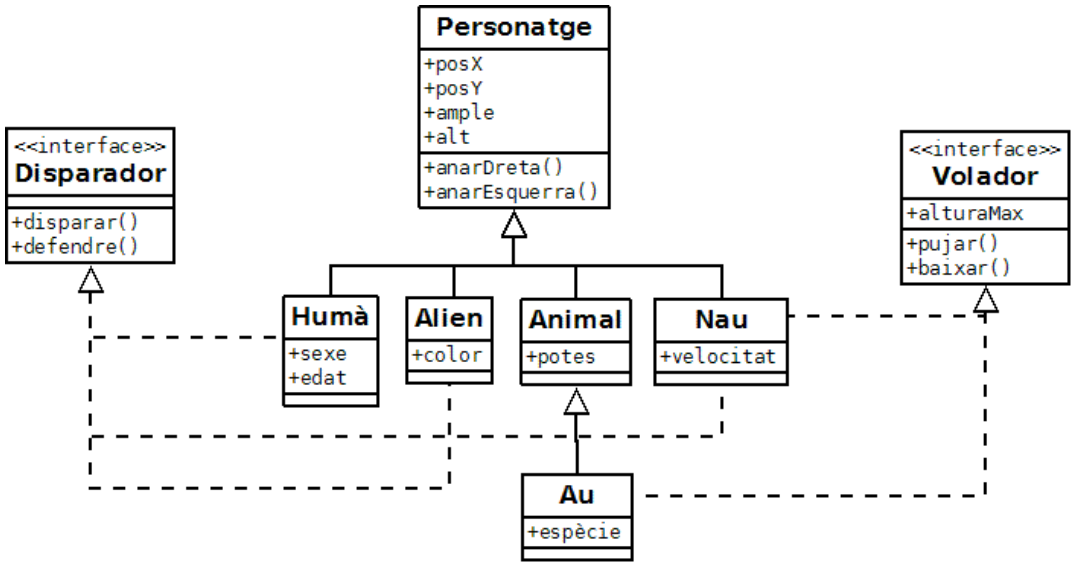
1 Introducció amb un exemple

En un videojoc de marcianets hi ha molts personatges (objectes) per la pantalla. Tots seran de la classe Personatge, però són de tipus 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, que les classes Nau, Humà i Alien tinguen el comportament en comú d'un personatge que siga disparador (disparar i defendre).

La forma ideal de fer-ho seria fer una classe Volador amb eixos mètodes, i altra Disparador amb els altres mètodes. Però en Java (a diferència de C) no podem fer que una classe siga filla de 2 classes. Per a això estan les interfícies.



Per a fer això, ens definirem una **interfície Volador** (i altra Disparador), on posarem els mètodes que hauran d'implementar les classes "voladores". Els mètodes de la interfície estan "buits". És a dir: està només la capçalera però no estan implementats.

Veiem com seria la implementació:

```
public interface Volador {
    int alturaMax = 100;

    void pujar();

    void baixar();
}
```

En una interfície també es poden definir **constants** (sempre seran final i static, encara que no cal indicar-ho).

Els mètodes no s'implementen ací, sinó que els haurà d'implementar cada classe que vullga implementar la interfície Volador. Encara que no es pose, els mètodes són abstract i public.

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

    // IMPLEMENTACIÓ DELS MÈTODES DE LA INTERFÍCIE 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 LA INTERFÍCIE Disparador:
    @Override
    public void disparar() {
        System.out.println("Pinyou, pinyou");
    }

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

Esta classe ha d'implementar tots els mètodes de les interfícies Volador i Disparador.

```
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

2.1 Introducció

Repassem primer uns conceptes previs:

> Mètode abstracte: mètode on no consta la implementació.
>
> Classe abstracta: classe que té algun mètode abstracte. No es pot instanciar.
>
> Les classes filles d'una classe abstracta estan obligades a implementar eixos mètodes abstractes, o bé tornar a declarar-los com a abstractes (i, per tant, eixa altra classe també serà abstracta).

Vist això, una interfície és com una classe abstracta pura (no té implementat cap mètode) amb l'avantatge que una classe pot implementar ("ser filla de") **moltes interfícies** (però sols pot extendre d'**una classe**).

Una interfície també pot extendre d'una altra interfície.

2.2 Definició

Conjunt de mètodes sense implementar que hauran d'implementar aquelles classes que vullguen comportar-se així. També pot incloure constants.

2.3 Utilitats

- Simular herència múltiple (ja que una classe només pot tindre 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

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()~~;
		
		ovni = new Nau(); // new Au();
		
		// o bé:
		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	No (En versió 8 de JDK sí, si poses davant "default").
Es poden definir atributs?	Sí	Només constants

Exercicis

1. Donada la següent interfície:

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

   a. Construeix la clase ArrayListDoubleEstad que tinga un ArrayList de doubles i que implemente la interfície Estadístiques. Nota: primer prova a indicar que implementa la interfície però sense implementar els mètodes. Voràs que et dóna error. Implementa ara els mètodes necessaris.

   b. Crea la clase ArrayDoubleEstad, amb un array (no ArrayList) de doubles, que implemente la interfície Estadístiques. Nota: Netbeans et permet afegir de forma automàtica, una implementació per defecte per als mètodes de la interfície que implementa. Clica en la icona de l'error i després en "*Implement all abstract methods*". Després, canvia l'ordre de llançament d'excepció per les instruccions que han d'implementar eixos mètodes.

   c. En la classe principal, defineix un objecte de cadascuna de les classes anteriors, afegeix valors a les seues llistes i usa els mètodes de la interfície que implementen 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. 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 cadascuna.

   b. Crea la classe *Departament*, amb un nom (*String*) i una quantitat d'empleats (int), que implemente la interfície *Comparar*. Un departament serà més gran que altre si té més empleats.

   c. En el programa principal crea dos departaments i mostra el més gran.

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:

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

 Arrays.sort(edats);   // Cal importar java.util.Arrays
for (int e: edats){
    System.out.print(e + " ");
}
```

D'igual forma, per a ordenar un ArrayList d'enters podem usar el mètode sort de la classe Collections:

```
ArrayList <Integer> edats2 = new ArrayList();
edats2.add(4);
edats2.add(7);
...
Collections.sort(edats2);       // Cal importar java.util.Collections
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 MVJ "no sap 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

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:

```
int compareTo(Object obj)
```

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;
        }
```

Així aconseguim que retorne un número negatiu si *this* és menor que *obj*; un número positiu si *this* és major que *obj*; o un zero si són iguals.

O bé, per a no fer el càsting en el *compareTo*, podem fer-ho 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);
```

És un altre motiu de l'ús d'interfícies: mitjançant la implementació d'interfícies tots els programadors fan servir el mateix nom de mètode i estructura formal per comparar objectes (o clonar, o altres operacions).

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 *Departament* implemente la interfície *Comparable*. Implementa el mètode *compareTo* fent que un *Departament* siga més gran que altre si té més empleats. En cas d'igualtat, serà més gran qui tinga el nom major que l'altre (caldrà cridar al *compareTo* de la classe *String*).
   b. Modifica els mètodes de *Departament* que implementen la interfície *Comparar* per a que ara es basen en la definició del mètode *compareTo*.
   c. En el programa principal crea un *ArrayList* de departaments i posa'n uns quants. Mostra l'*ArrayList*, ordena'l i torna'l a mostrar. Nota: per a mostrar l'ArrayList simplement amb *System.out.println(departaments)* cal que tingues definit el *toString* en el *Departament*.

4.3 La interfície Comparator

Amb la interfície *Comparable* podíem comparar (ordenar) alumnes per un criteri establert: el codi de l'alumne. Però i si decidim ordenar-los pel nom, o 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, igual que el mètode *compare*.

Exemples:

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

```
import java.util.Comparator;
public class ComparadorPersonaCursEdat 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:

```
if ((new ComparadorPersonaNom()).compare(p1, p2) < 0) { ... }
```

O bé, ordenar una llista d'alumnes (array, ArrayList...) amb el sort, com abans, però passant-li també l'objecte que té el criteri de comparació:

```
Collections.sort(llistaAlumnes, new ComparadorPersonaNom() );
```

Exercicis

4. L'objectiu és tindre una llista de factures i poder-les ordenar per diferents criteris.

    a. Crea la classe Factura amb els següents atributs:
        i. numero: enter
        ii. data: objecte de la classe Date (caldrà importar java.util.Date)
        iii. import (float)

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

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

5 Les interfícies Iterator i Iterable

5.1 Introducció

Per a recórrer una llista es pot fer de diverses maneres. Hi ha dos formes de fer-ho sense comptador: unsant la interfície Iterator o un bucle for-each. Per exemple, per a una llista de strings:

Interfície Iterator	Bucle for-each (a partir de Java 5)

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

Veiem que el for-each és més senzill però, per exemple, si esborrem un element de la llista mentre la recorrem amb for-each pot donar error ja que és com si ens furtaren les baldoses per on caminem. El mètode remove() d'Iterator ho resol:

Interfície Iterator	Bucle for-each (a partir de Java 5)

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

NO ERROR	POT PROVOCAR L'EXCEPCIÓ: java.util.ConcurrentModificacionException

Amb les interfícies Iterator i Iterable podrem recórrer una col·lecció. Utilitats d'estes interfícies:

- Recórrer una col·lecció mentre esborrem alguns dels seus elements.
- Recórrer una col·lecció sense saber com està implementada.
- Recórrer diferents tipus de col·leccions de la mateixa forma
- Recórrer una mateixa col·lecció per diferents recorreguts.

Per a l'ús d'estes interfícies cal importar **java.util.Iterator**.

5.2 La interfície Iterator

Sintaxi d'ús:

```
Iterator <TipusARecórrer> it = nomDeLaColleccio.iterator();
```

La col·lecció podrà ser un ArrayList, Vector, Set... o bé qualsevol classe que implemente la interfície Iterable (que vorem en el punt següent) que haurà d'implementar el mètode *iterator()*.

Compte: no cal confondre la interfície Iterator amb el mètode iterator();

Així estem creant un objecte (it) de tipus *Iterator*. No ho fem amb el *new* ja que *Iterator* és una interfície, no una classe (i, per tant, no es pot instanciar).

La interfície Iterator proporciona uns mètodes per a accedir seqüencialment als elements d'una col·lecció. Açò és important perquè Java té moltíssimes col·leccions distintes (en la versió 1.6 Java en té unes 50: List, Set, Queue, ArrayList, Tree, array...). Per això es pretén accedir de la mateixa forma als elements d'eixes col·leccions (o a altres que ems fem nosaltres).

Els mètodes que proporciona la interfície Iterator són:

- public boolean **hasNext()** → retorna si hi ha un altre element o no
- public E **next()** → retorna el següent element (E : classe que vullgam)
- public void **remove()** → elimina l'últim element retornat.

Vegem un parell d'exemples d'ús de la interfície Iterator.

- En el primer usarem una classe de l'API de Java que ja implementa eixa interfície. La classe ArrayList. Recorrerem l'ArrayList amb l'iterador que ens proporciona.
- En el segon crearem nosaltres una classe que tinga una llista d'elements i que implemente la interfície Iterator. Després recorrerem la llista amb l'iterador.

Exemple 1: Usar els mètodes de la interfície Iterator a partir una classe que ja implementa eixa interfície (la classe ArrayList).

```
ArrayList <Persona> llistaPersones = new ArrayList();
Iterator <Persona> it = llistaPersones.iterator();
...
Persona p;    // Objecte temporal
while (it.hasNext()){
        p = it.next();
        if (p.getEdat() < 18) {
                System.out.println("És menor. L'esborrem");
                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");
}
```

En el segon next() tindríem el següent element a l'obtingut en el primer next().

Exemple 2: Usar els mètodes de la interfície Iterator a partir una classe que ens creem que implemente eixa interfície (per exemple: Departament).

Ens definirem la classe Departament, la qual tindrà una col·lecció d'empleats. Eixa col·lecció la farem amb un array però qui use objectes de Departament, no té perquè saber si la col·lecció d'empleats està implementat com un array, ArrayList, Vector, etc.

```
public class Empleat {
        private String nom;
        private String carrec;
        public Empleat(String nom, String carrec) {
            this.nom = nom;
            this.carrec = carrec;
        }
        ... // gets i sets
        @Override
        public String toString() {
            return "Emp{" + "nom=" + nom + ", càrrec=" + carrec + '}';
        }
}
```

Esta classe no té res d'especial.
És la dels elements de la llista.

```
import java.util.Iterator;

public class Departament {
    private String nom;
    private Empleat[] llistaEmpleats = new Empleat[100];
    private int qEmpl = 0;

    Departament(String nom) {   this.nom = nom;    }

    public void add(String nomEmpleat, String carrec){
        llistaEmpleats[qEmpl++] = new Empleat(nomEmpleat, carrec);
    }

  // @Override
  // public Iterator<Empleat> iterator() {
  public Iterator<Empleat> iterador() {
        return new IteradorDEmpleats();
  }

    protected class IteradorDEmpleats implements Iterator<Empleat> {
        private int posicio = 0;

        @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--;
        }
}
```

Esta classe és la que tindrà una llista d'elements i tindrà un mètode que retornarà un iterador.
Serà convenient que esta classe implemente *Iterable*, però ja ho vorem quan vegem eixa interfície.

Serà convenient que, en compte de definir mètode *iterador()*, se sobreescriguera el mètode *iterator()* de la interfície *Iterable*.

El mètode *iterador()* retorna un obecte *IteradorDEmpleats* (és a dir: *Iterator<Empleat>*) que implementa la interfície *Iterator*. Per a fer això cal crear eixa classe.

Esta classe interna és la que implementa l'iterador.
També haguera pogut anar fora. En eixe cas caldria passar-li al constructor els paràmetres: *llistaEmpleats* i *qEmpl*.

El mètode *remove()* no estem obligats a sobreescriure'l, ja que ja n'hi ha una implementació per defecte en la pròpia interfície *Iterator*.

Finalment, el codi del programa principal on fem servir les classes Empleat i Departament amb el seu Iterador. Per exemple, esborrarem de la llista els empleats que tenen de càrrec "no res".

```
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("Marc", "no res");
        dep.add("Pep", "programador");
        dep.add("Alfred", "no res");
        dep.add("Maria", "analista");

        it = dep.iterador();
        while (it.hasNext()) {
            empl = it.next();
            if (empl.getCarrec.equals("no res") { it.remove (); }
        }
        System.out.println ("Empleats del departament:" + dep.toString());
     }
}
```

Exercicis

5. Llista d'alumnes
    a. Crea la classe Alumne amb:
        i. Constant QAVA (quantitat avaluacions): 3
        ii. Constant QEXER (quantitat d'exercicis per avaluació): 5
        iii. Matriu notes de QAVA avaluacions per QEXER exercicis.
        iv. Un mètode que permeta posar una nota a l'alumne (paràmetres: nota, núm. avaluació i núm. exercici).
        v. Un iterador de la matriu per a poder recórrer les seues notes.
    b. En la lasse principal, fes un bucle que mostre un menú amb les opcions:
        i. Nou alumne
        ii. Posar una nota
        iii. Mostrar llista de notes d'un alumne (amb un Iterator)
        iv. Eixir

5.3 La interfície Iterable

Si definim una classe amb una llista que volem que siga utilitzada amb un iterador, és convenient indicar que la classe és iterable. És a dir: que es pot recórrer amb un iterador que ja té definit. Això s'aconsegueix fent que la nostra classe implemente la interfície *Iterable*, que només té el mètode *iterator()*.

En l'exemple que hem vist abans, només caldria indicar que la classe Departament implementa la interfície Iterable i substituir el mètode iterador() per iterator():

```
import java.util.Iterator;

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

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

   protected class IteradorDEmpleats implements Iterator<Empleat> {
       @Override
       public boolean hasNext() { ... }

       @Override
       public Empleat next() { ... }

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

}
```

Resumint: per a implementar la interfície Iterable hem de sobreescriure el mètode *iterator()*, i per a això hem de poder tornar un objecte *Iterator*, la qual cosa aconseguim creant una classe interna que implementa la interfície *Iterator*.

Nota: si volem usar el bucle for-each en una classe haurà d'implementar *Iterable*.

Exercicis

6. En l'exercici anterior hem creat la classe Alumne amb una matriu que volem que siga recorreguda com una llista. Com hem vist, és aconsellable que la classe Alumne implemente la interfície Iterable. Per tant.
    a. Fes que la classe alumne implemente la interfície Iterable.
    b. Això farà que el compilador t'obligue a implementar el mètode iterator(). Fes que Netbeans (o Eclipse) implementen eixe mètode automàticament. Serà una implementació per defecte que hauràs de modificar (ho faràs en el punt següent).
    c. Ara tindràs el mètode iterador() (que has fet tu) i el iterator() (que ha posat l'IDE automàticament). Fes que iterator() faça el mateix que iterador(). Després, ja pots esborrar el mètode iterador().
    d. En el programa principal ara tindràs un error, ja que quan crees un objecte per a iterar cridaves al mètode iterador(). Substitueix la crida a iterador() per la crida a iterator(). El fet de fer este exercici fa que si sabem que una classe implementa Iterable, ja sabrem que disposa del mètode iterator() (no haurem de buscar si té un mètode per a aconseguir un iterador ni com es diu este mètode).

6 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 hauríem de fer és un mètode per a copiar atribut a atribut.

La classe Object ja té eixe mètode, anomenat *clone()*, que retorna un *Object* idèntic. La forma d'usar el clone seria així:

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

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

Però per a poder invocar eixe mètode estem obligats a implementar-lo en la nostra classe, ja que en la classe Object està definit com a protected.

Per a implementar el *clone()* en la nostra classe Cotxe ho podem fer de dos formes distintes: invocant al *clone()* de la classe *Object* o sense invocar-lo.

a) **<u>La còpia la fa la pròpia classe (no invoca a *super.clone()*)</u>**

- Hem de reservar memòria per al nou objecte

- Hem de copiar atribut a atribut al nou objecte

```
public class Cotxe {
    String matr;
    int anys;



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

Reservar memòria

Copiar atributs

Retornar l'objecte creat

O bé:

```
public class Cotxe {
    String matr;
    int anys;

    public Cotxe(String matr, int anys){
        this.matr = matr;
        this.anys = anys;
    }

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

Reservar memòria, copiar atributs i retornar l'objecte al mateix temps

b) **La còpia la fa el pare de la classe, "Object" (invoca a *super.clone()*)**

- No cal reservar mèmòria
- No hem de copiar atribut a atribut
- Cridarem a *super.clone()* dins d'un try-catch
- La classe ha de tindre el ***implements Cloneable***.

```
public class Persona implements Cloneable{
    String nom;
    int edat;
    Cotxe cotxe;

    public Persona(String nom, int edat, Cotxe cotxe) {
        this.nom = nom;
        this.edat = edat;
        this.cotxe = cotxe;
    }

    @Override
    public Persona clone(){
        Persona clon=null;
        try {

            clon = (Persona) super.clone();

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

        } catch (CloneNotSupportedException ex) {
            System.out.println("No es pot duplicar");
        }
        return clon;
    }
}
```

Cal implementar la interfície *Cloneable*.

El *clone()* de la classe pare (en este cas, la classe *Object*) ens retorna un objecte idèntic a l'actual però de la classe *Object*, que cal convertir a *Persona*.

No fa falta però si volem que l'objecte *cotxe* es copie amb el *clone()* de la classe *Cotxe*, cal fer-ho així.

Eixa interfície (*Cloneable*) és especial perquè no té cap mètode. Per tant, per a què serveix? Com clonar un objecte pot ser "perillós", serveix per a dir-li al *clone()* de la classe *Object* (quan és invocat per la nostra classe) que estem d'acord que faça una còpia camp per camp. Si el *clone()* d'*Object* comprova que la nostra classe no implementa *Cloneable*, donarà l'excepció *CloneNotSupportedException*.