

JAVA LIBRARY

- Trier un tableau
- Les collections dynamiques
- La gestion des exceptions/erreurs
- Gestion des fichiers et dossiers
- Sérialisation d objet
- JDBC et accès aux bases de données
- JDBC et pattern DAO



TABLEAUX ET COLLECTIONS

TRIER UN TABLEAU (ARRAYS.SORT)

Utilisation :

```
static void TableauInt(){  
    int[] tab = {10,20,1,3,2};  
    for (int e : tab) System.out.print(e+" ");  
    System.out.println("");  
    Arrays.sort(tab);  
    for (int e : tab) System.out.print(e+" ");  
}
```

```
static void TableauString(){  
    String[] tab = {"abc","ABC","Le rouge et le noir",  
        "Hey ho", "245", "zzzz","XXX"};  
    for (String e : tab) System.out.print(e+" ");  
    System.out.println("");  
    Arrays.sort(tab);  
    for (String e : tab) System.out.print(e+" ");  
}
```

RENDRE DES PERSONNES COMPARABLES

5

```
public class Personne implements Comparable<Personne> {  
    [...]  
    public int compareTo(Personne o) {  
        //compare d'abord selon l'âge puis selon le nom  
        //-1 si this plus petit, 0 si pareil, 1 sinon  
        if (age==o.getAge()){  
            return nom.compareTo(o.getNom());  
        }  
        else return ((Integer)age).compareTo(o.getAge());  
        //ou : else return (new Integer(age)).compareTo(o.getAge());  
    }  
}
```

On peut alors utiliser sort :

```
static void TableauPersonne(){  
    Personne[] tab = {new Personne("Jac", "ALi", 26),new Personne("Jsrc", "Aser", 16),  
        new Personne("Jsrc", "Avqr", 16),new Personne("Bjo", "Mor", 16),  
        new Personne("Tred", "bvtsvr", 21),new Personne("Vef", "Moli", 102),  
        new Personne("Het", "Pad", 26),new Personne("Poujn", "Edgarrrr", 33),  
        new Personne("zffz", "ger", 60)};  
    for (Personne e : tab) System.out.print(e+" ");  
    System.out.println("");  
    Arrays.sort(tab);  
    for (Personne e : tab) System.out.print(e+" ");  
    System.out.println("");}
```

- Cadre : on nous fourni un .jar qui contient une classe `Personne` sur laquelle on souhaiterait pouvoir comparer, on ne peut pas la modifier pour y ajouter un implements
- Pour cela : créer une nouvelle classe implementant `Comparator`

Créer une classe comparator :

```
import java.util.Comparator;

public class ComparePrenom
implements Comparator<Personne>{

    @Override
    public int compare(Personne o1, Personne o2) {
        return o1.getPrenom().compareTo(o2.getPrenom());
    }
}
```

Utilisation :

```
static void TestcomparePrenom(){
    Personne[] tab = {new Personne("Jac", "ALi", 26),
        new Personne("Jsrc", "Aser", 16),
        new Personne("Jsrc", "Avqr", 16),
        new Personne("Bjo", "Mor", 16)};
    Arrays.sort(tab, new ComparePrenom());
    for (Personne e : tab) System.out.print(e+" ");
    System.out.println("");
}
```

On a utilisé la méthode : `Arrays.sort(tab, new ComparePrenom());`

Deuxième argument : un comparateur, dans lequel est implémenter une fonction permettant de comparer des instances du type qui nous intéresse

Pour des int : ==

Pour des String : .equals (ou .equalsIgnoreCase)

.equals existe pour Object : équivalent à ==

.equals surchargeable ! On peut la redéfinir pour nos types.

Parfois utiliser implicitement par des méthodes, doit être recoder pour pouvoir par exemple utiliser des collections (sinon identité de références, pas très utile !)

Globalement toujours bien de la redéfinir

La méthode **public int hashCode()** « va avec » : une méthode qui retourne un entier, en faisant en sorte que chaque objet ait un identifiant unique, nécessaire pour de nombreuses méthodes, doit normalement être codée en même temps que .equals (sert dans le même genre de cas – sauf appels explicite à .equals)

A générer via eclipse ou à déléguer à des hashCode d'autres classes

- Un tableau à une taille fixe, la taille d'une collection évolue en fonction du contexte choisir l'un ou l'autre
- On va utiliser pour cela une classe java : ArrayList

Par défaut, une ArrayList prend des arguments de types Object,
Fait des cast implicite quand on lui donne des int.
On peut utiliser le for each sur une ArrayList

```
import java.util.ArrayList;

public class TestCollections {

    public static void main(String[] args) {
        Test1();
    }

    static public void Test1(){
        ArrayList a1 = new ArrayList();
        a1.add(10); a1.add(20);
        for (Object x:a1) System.out.println(x);
    }
}
```


LA CLASSE ARRAYLIST

On peut l'utiliser sur un type particulier plutôt que sur Object (par défaut)

```
public class ArrayList<E> extends AbstractList<E>  
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Impose la méthode add(E e)

```
static public void Test2(){  
    ArrayList<Integer> a1 = new ArrayList<Integer>();  
    a1.add(10); a1.add(20);  
    for (int x:a1) System.out.println(x);  
}
```

COMMENT ÇA FONCTIONNE ?

- A la création, crée un tableau de capacité fixe (16), quand le tableau est plein, double la capacité.
- Gère à la fois la taille (accessible via **size()**) et la capacité. C'est quand la taille atteint la capacité que le tableau double de taille
- Les technologies évoluent, différents types de listes peuvent fonctionner différemment
- En pratique, ce n'est donc pas dynamique tout le temps côté mémoire mais paraît dynamique pour l'utilisateur.
- On peut décider de la capacité initiale le tableau : si on est sûr d'avoir beaucoup d'éléments, mieux vaut le dire de suite pour améliorer la performance. (l'opération de doubler la capacité du tableau est plutôt coûteuse)

- On peut parcourir la liste des méthodes proposées, voici quelques méthodes utiles :

```
a1.add(40);  
//ajoute l'élément donné en entrée  
a1.size();  
//renvoie la taille (int)  
a1.clear();  
//vide la liste de tous ses éléments sans supprimer la liste  
a1.get(0);  
//renvoie l'élément à l'index donné en argument,  
//l'équivalent de tab[0]  
a1.remove(1);  
//supprime l'élément à la position indiquée en paramètre,  
//et décale tous les suivants pour ne pas avoir de trou  
a1.clone();  
//renvoie une nouvelle instance de a1,  
//contenant les mêmes éléments
```

- `toString()` y ait redéfini pour afficher tous les éléments de la liste dans `AbstractCollection` (classe mère de `AbstractList`, la classe mère de `ArrayList`)

ex :

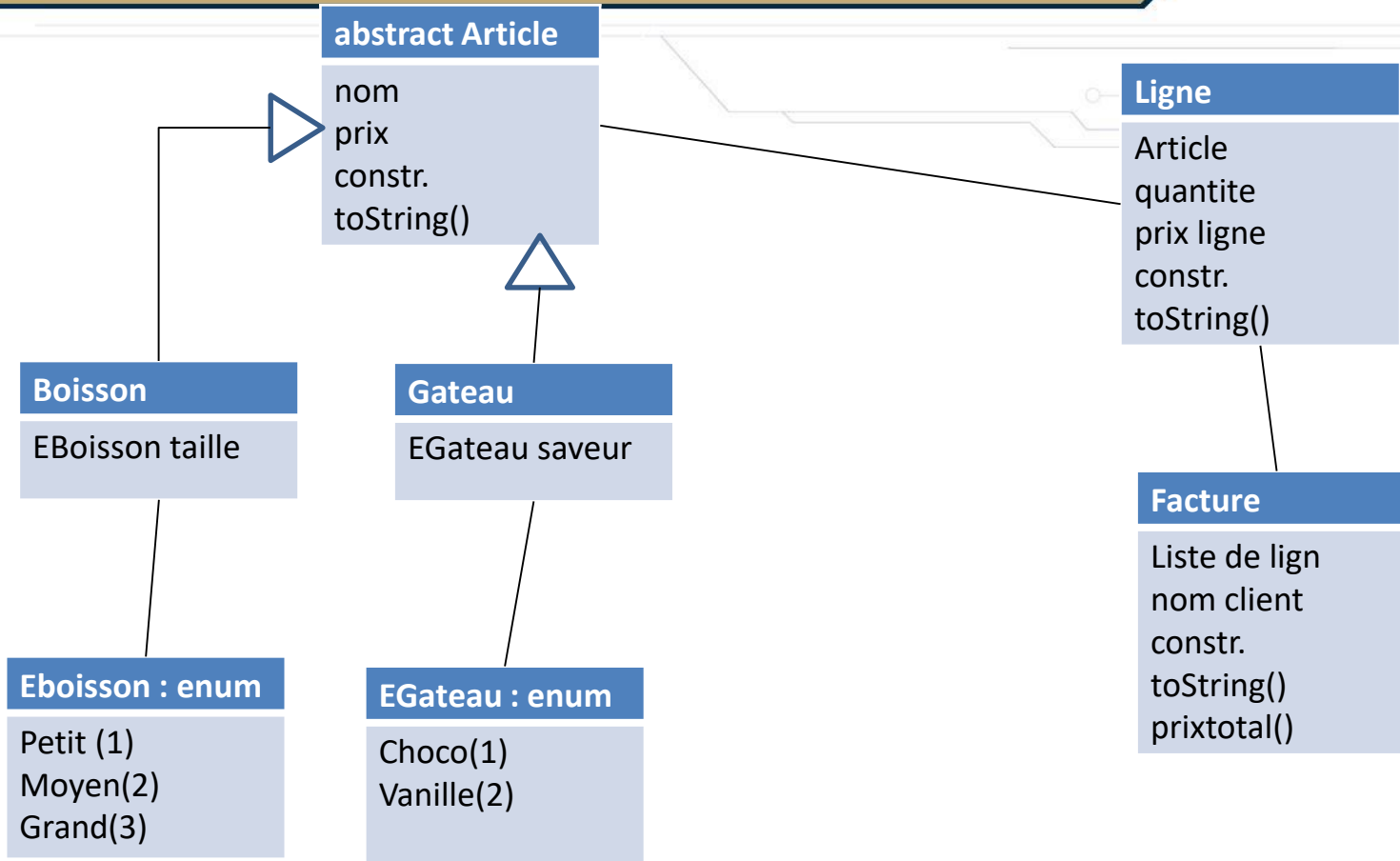
```
ArrayList<Integer> a1 = new ArrayList<Integer>();  
a1.add(10); a1.add(20); a1.add(400); a1.add(22);  
System.out.println(a1);  
donne «[10, 400, 22, 40]».
```

- Méthode `toArray` :

```
ArrayList a1 = new ArrayList();  
a1.add(10); a1.add(20); a1.add(400); a1.add(22);  
Object[] tab = a1.toArray(); //tableau contenant les mêmes éléments mais devient de taille fixe
```

- Peut être 'casté/converti' vers `List` (importer le bon package) : restreint la visibilité aux méthodes de l'interface (utilité : polymorphisme)

```
ArrayList a1 = new ArrayList();  
a1.add(10); a1.add(20); a1.add(400); a1.add(22);  
List l = a1;
```



AUTRES COLLECTIONS

Une autre collection

Pour manipuler des files d'attentes

FIFO : First In First Out : premier arrivé premier sorti

add, ajoute des éléments en fin de liste.

size : le nombre d'éléments dans la liste,...

poll : supprime le premier élément de la liste et le renvoie

peek : renvoie le premier élément sans le supprimer de la liste

```
import java.util.LinkedList;
```

```
LinkedList<String> queue = new LinkedList<String> ();  
queue.add("premier patient"); queue.add("Mr. Truc");  
queue.add("Mme Bidule"); queue.add("dernier patient");  
String a = queue.poll();  
String b = queue.peek();  
System.out.println(queue);  
System.out.println(a+b);
```

- Table de hachage, fonctionne sur l'association clé-valeur
- Aussi appelé dictionnaire `import java.util.HashMap;`
- Chaque éléments est constitué de deux éléments : une clé et une valeur

Type des clé Type des valeurs

```
public class HashMap<K,V> extends AbstractMap<K,V>  
    implements Map<K,V>, Cloneable, Serializable {
```

Un dictionnaire qui range des chaînes de caractères avec des entiers pour clés :
`HashMap<Integer, String> dico2 = new HashMap<Integer, String>();`

Si rien n'est précisé pour les types, alors ce seront par défauts des objets :
`HashMap dico1 = new HashMap(); // dico d'objet avec des objets pour clé`

- Interdiction de doublon pour la clé (unicité des clés) :
Pas deux valeurs différentes avec la même clé !
- On récupère la valeur grâce à la clé, pas l'inverse

```
HashMap<Integer, String> dico2 = new HashMap<Integer, String>();  
dico2.put(34, "toto");  
dico2.put(1, "blabla"); La méthode put ajoute un élément  
dico2.put(22, "valeurlà");  
dico2.put(15, "Toto");  
System.out.println(dico2); //{1=blabal, 34=toto, 22=valeurlà, 15=Toto}  
String s = dico2.get(22);  
System.out.println(s); //valeurlà  
//On peut également récupérer toutes les clés ou valeurs :  
Collection clefs = dico2.keySet();  
System.out.println(clefs); //[1, 34, 22, 15]  
Collection valeurs = dico2.values();  
System.out.println(valeurs); //[blabal, toto, valeurlà, Toto]
```

■ Exemple HashMap

```
public class HashMap<K,V> extends AbstractMap<K,V>
```

Type de la clé
(générique)

Type des données

Utilisation :

```
HashMap<Integer, String> h1 = new HashMap<Integer,String>();
```

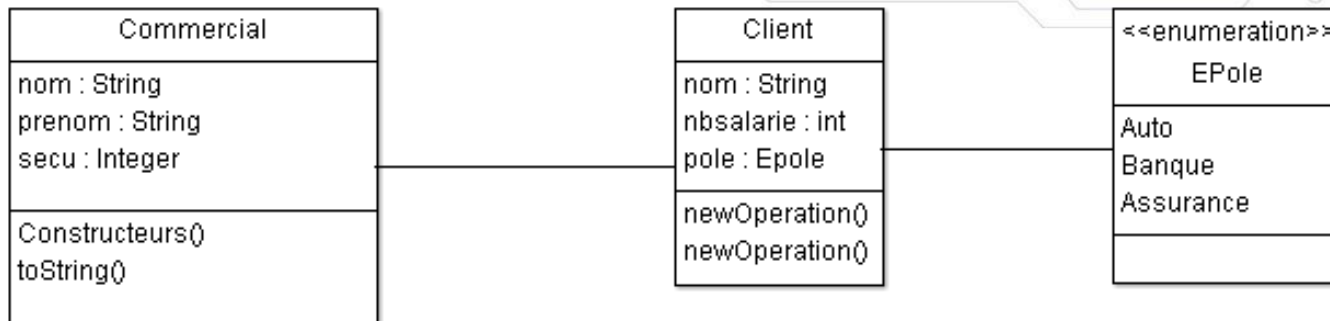
```
HashMap<String, Double> h2 = new HashMap<String, Double>();
```

Spécifier avec quel type on va l'utiliser

```
public final class Integer extends Number implements Comparable<Integer>
```

Implémente comparable sur les Integer

chaque commercial chapeaute tous les clients d'un pole (via une HashMap)



La HashMap :

Commercial =clé	ArrayList<Client> =valeurs

GESTION DES EXCEPTIONS/ERREURS

Souvent, un programme doit traiter des situations exceptionnelles qui n'ont pas un rapport direct avec sa tâche principale.

Ceci oblige le programmeur à réaliser de nombreux tests avant d'écrire les instructions utiles du programme. Cette situation a deux inconvénients majeurs :

- Le programmeur peut omettre de tester une condition ;

- Le code devient vite illisible car la partie utile est masquée par les tests.

Java remédie à cela en introduisant un *Mécanisme de gestion des exceptions* .

Grâce à ce mécanisme, on peut améliorer grandement la lisibilité du Code , en séparant

le code utile (Code métier) de celui qui traite des situations exceptionnelles,

Java peut générer deux types d'erreurs au moment de l'exécution:

Des erreurs produites par l'application dans des cas exceptionnels que le programmeur devrait prévoir et traiter dans son application. Ce genre d'erreur sont de type Exception

Des erreurs qui peuvent être générée au niveau de la JVM et que le programmeur ne peut prévoir dans son application. Ce type d'erreurs sont de type Error.

En Java, on peut classer les exceptions en deux catégories :

- Les exceptions surveillées,

- Les exceptions non surveillées.

Java oblige le programmeur à traiter les erreurs surveillées. Elles sont signalées par le compilateur

Les erreurs non surveillées peuvent être traitées ou non. Et ne sont pas signalées par le compilateur

■ Que se passe-t-il en cas d'erreur d'exécution ?

```
package coursExceptions;  
import java.util.Scanner;
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("saisir votre âge");  
        int age = sc.nextInt();  
        System.out.println("votre âge est : "+age);  
        System.out.println("A bientôt");  
    }  
}
```

Instruction pouvant déclencher une [InputMismatchException](#)

Instructions qui ne s'exécute pas après plantage sur la ligne précédente

Console :

Message affiché par le programme

saisir votre âge

Bonjour

```
Exception in thread "main" java.util.InputMismatchException  
at java.util.Scanner.throwFor(Unknown Source)  
at java.util.Scanner.next(Unknown Source)  
at java.util.Scanner.nextInt(Unknown Source)  
at java.util.Scanner.nextInt(Unknown Source)  
at coursExceptions.Main.main(Main.java:10)
```

Déclenchement d'une exception :

C'est l'exception [java.util.InputMismatchException](#)

Qui c'est déclenché, la suite du message nous dit à travers quels appels.

Le programme **plante et s'interrompt** : la suite du programme n'est pas effectué

PLANTAGES = ERREURS À L'EXÉCUTION

Souvent dû à l'utilisateur mais pas forcément (fichier inexistant, plantage système, tableau dépassé...)

Tout ce qui est extérieur au programme ou qui interagit avec est susceptible de faire planter le programme : il faut donc surveiller (gérer les exceptions potentiels)

- **Bloc try-catch :** Essaye de faire ce qui est dans ces accolades, si tout se passe bien on passe, on ignore le catch
Si une l'exception a été déclenchée lors du bloc try, ce qui est effectué dans le try est interrompu, et on fait à la place les instructions dans le bloc du catch

```
int age = -1;
Scanner sc = new Scanner(System.in);
System.out.println("saisir votre
âge");
try{
age = sc.nextInt();
System.out.println("votre âge est :
"+age);}
catch(Exception e){
System.out.println("DSL, erreur de
saisie");}
System.out.println("A bientôt");
```

Exécution avec un entier saisi :

```
saisir votre âge
4
votre âge est : 4
A bientôt
```

Exécution avec une erreur de saisie :

```
saisir votre âge
bla
DSL, erreur de saisie
A bientôt
```

L'exception a été rattrapé

```
public static void main(String[] args) {  
    try{m();}  
    catch(Exception e){  
        System.out.println("DSL, erreur de saisie");  
        System.out.println("Aurevoir");  
    }  
}
```

- Permet de récupérer après une méthode qui peut planter

La méthode :

```
private static double Div(int num, int denom) {  
    return ( num /  denom);  
}
```

appelée avec 0 comme deuxième argument
déclenche une exception de type ArithmeticException :

```
Exception in thread "main" java.lang.ArithmeticException: / by  
at coursExceptions.Main.Div(Main.java:34)  
at coursExceptions.Main.testDiv(Main.java:28)  
at coursExceptions.Main.main(Main.java:9)
```

Les exceptions ont plusieurs types et se comportent en fait comme des classes, qui héritent (plus ou moins directement) de la classe Exception

```
public static void testthrow(){  
    RuntimeException e = new RuntimeException("toto");  
    throw e;  
}
```

Message affiché

Instance d'une exception de type Runtime

Exception in thread "main" java.lang.RuntimeException: toto
at coursExceptions.Main.testthrow(Main.java:40)
at coursExceptions.Main.main(Main.java:10)

Une instance d'exception peut être catchée :

Try{... throws e ...} catch (Exception exc) {...}

Déclenchera le catch (qu'il soit « throwé » directement ou via une méthode)

ATTRAPER CERTAINS TYPES D'EXCEPTIONS

```
try{...}  
catch(Exception e){...}
```

Tout types d'exception car Exception est la classe la plus haute dans la hiérarchie (par ex. une RuntimeException est une Exception). On récupère également l'instance e particulière à cette exception.

```
try{...}  
catch(RuntimeException e){...}
```

Rattrape uniquement les exceptions de type RuntimeException

Tous les types d'exceptions possèdent les méthodes suivantes :

`getMessage()` : retourne le message de l'exception

`System.out.println(e.getMessage());`

Résultat affiché : / by zero

`toString()` : retourne une chaîne qui contient le type de l'exception et le message de l'exception. `System.out.println(e.toString());`

*Réstat affiché : **java.lang.ArithmeticException: / by zero***

`printStackTrace`: affiche la trace de l'exception `e.printStackTrace();`

Résultat affiché :

`java.lang.ArithmeticException: / by zero`

`at App1.calcul(App1.java:4)`

`at App1.main(App1.java:13)`

```
try{  
throw new ArithmeticException("toto");  
}  
catch(InputMismatchException e){  
System.out.println("dans catch");  
}  
finally{  
System.out.println("à bientôt");  
}
```

S'exécute toujours, ie dans les 3 cas possibles :

- Aucune exception levée, le bloc try s'est effectué normalement
- Une exception attrapée a été levée et le bloc catch s'est déroulé normalement
- Une exception non attrapée a été levée (où que ce soit)

Usage principal : libérer des ressources non gérées

fermer les accès aux ressources, par exemple à une base de données, il faut alors être sûr que l'on aura pas de fuite, fermer dans tous les scenarios possibles

Le bloc finally s'exécute quelque soit les différents scénarios.

- But : être informatif pour l'utilisateur, pour qu'il puisse comprendre le soucis et y remédier (ou que nous/des devs puissions débbugger après rapport d'erreur)

```
try{  
    ArithmeticException monex=new ArithmeticException("Mon message");  
    throw monex;  
}  
catch(ArithmeticException e){  
    System.out.println(e.getMessage());  
}  
finally{  
    System.out.println("à bientôt");  
}
```

Méthode de la classe Throwable (ou Exception hérite) permet d'accéder au message d'erreur, initialiser par le constructeur, c'est un attribut de l'instance

On peut utiliser aussi un constructeur implicite pour l'exception mais alors le message n'est pas initialisé

- Message et getMessage via throwable (que l'on utilise pas directement, on est utilisateur de Exception)
- `printStackTrace()` : permet d'afficher la trace de la pile d'appels (pour savoir quelles ont été les méthodes invoquées de manière imbriquées au moment du plantage)
- `toString()` : renvoie nom de l'exception + message

Certaines classes (comme Exception) sont surveillées :

throw new Exception();

ne peut pas être utilisé sans try catch, cela déclenche une erreur de compilation :

Unhandled exception type Exception

- il faut alors que la méthode qui l'utilise permettent un « throws » :

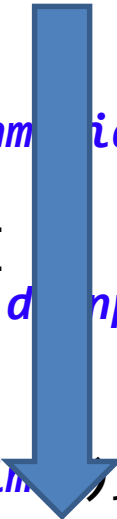
```
public static void m() throws Exception{  
    throw new Exception();}
```

Cela revient à refiler le problème à l'appelant, qui doit alors encapsuler la méthode dans un try catch (ou utiliser throws à son tour)

ATTRAPER DIFFÉRENTS TYPES D'EXCEPTIONS

36

```
try{
...
}
catch(ArithmeticException e){
System.out.println("erreur arithmétique");
}
catch(InputMismatchException e){
System.out.println("erreur type d'input");
}
catch(RuntimeException e){
System.out.println("erreur runtime");
}
catch(Exception e){
System.out.println("erreur non définie");
}
finally{
...
}
```



Les catch sont tentés l'un après l'autre jusqu'à trouver l'exception correspondante, effectue le bloc correspondant puis passe au finally.

Ici, une **ArithmeticException** ne déclenchera donc que le premier bloc.

Il faut donc écrire les catch du plus précis au moins précis (selon l'héritage)

```
public static void gestiondesexc(int a){  
    try {  
        switch (a){  
            case 1:  
                throw new RuntimeException("cas 1");  
            case 2:  
                throw new ArithmeticException("cas 2");  
            default :  
                throw new Exception("cas 3");  
        }  
    }  
    catch(ArithmeticException e){  
        System.out.println(e);  
    }  
    catch(RuntimeException e){  
        System.out.println(e);  
    }  
    catch(Exception e){  
        System.out.println(e);  
    }  
}
```

Différents modules utilisés dans différents cas

Gestion centralisées de certains types d'erreur

Il faut pour cela propager les exceptions à travers les classes métiers via throws.
La gestion des exceptions dépend souvent du support,
donc à gérer plutôt dans le système central
(toujours l'idée ne pas mettre de print dans classes métiers)

- Il est plus professionnel de créer une nouvelle Exception nommée
- En héritant de la classe Exception, nous créons une exception surveillée.
- Pour créer une exception non surveillée, vous pouvez hériter de la classe RuntimeException

```
package metier;  
  
public class MonException extends Exception {  
    public MonException(String message) {  
        super(message);  
    }  
    ....  
    public void Operation(float mt)throws MonException {  
        if(condition) throw new MonException(« message important");  
    }  
    .....  
    try {  
        Operation();  
    } catch (MonException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

JAVA

UTILISATION DES FICHIERS, DOSSIERS

But : extraire des infos des fichiers et dossiers

La classe File

La classe File permet de donner des informations sur un fichier ou un répertoire

La création d'un objet de la classe File peut se faire de différentes manières :

```
File f1=new File("c:/projet/fichier.ext");
```

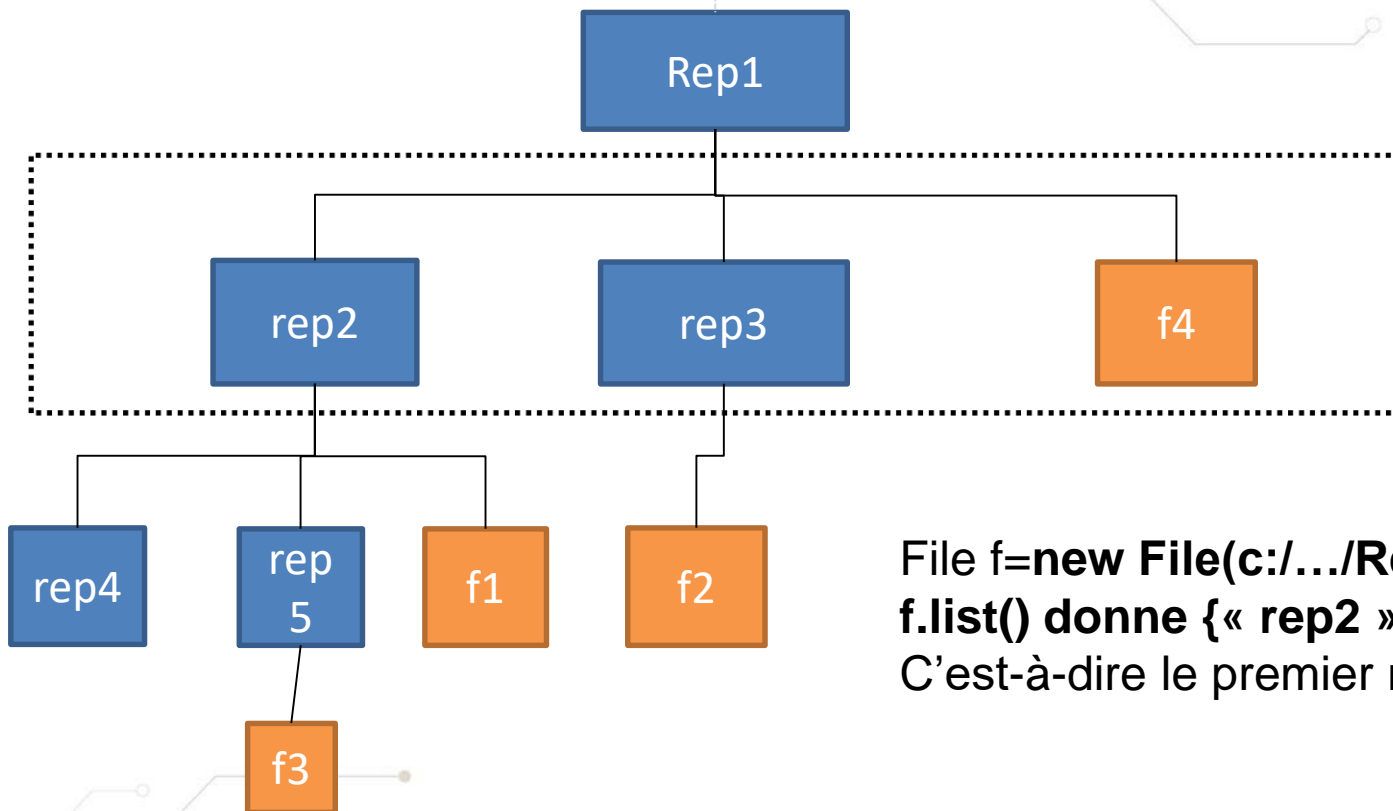
```
File f2=new File("c:/projet", "fichier.ext");
```

```
File f3=new File("c:/projet");
```

- `String getName();` Retourne le nom du fichier.
- `String getPath();` Retourne la localisation du fichier en relatif.
- `String getAbsolutePath();` Idem mais en absolu.
- `String getParent();` Retourne le nom du répertoire parent.
- `boolean renameTo(File newFile);` Permet de renommer un fichier.
- `boolean exists();` Est-ce que le fichier existe ?
- `boolean canRead();` Le fichier est t-il lisible ?
- `boolean canWrite();` Le fichier est t-il modifiable ?
- `boolean isDirectory();` Permet de savoir si c'est un répertoire.
- `boolean isFile();` Permet de savoir si c'est un fichier.
- `long length();` Quelle est sa longueur (en octets) ?
- `boolean delete();` Permet d'effacer le fichier.
- `boolean mkdir();` Permet de créer un répertoire.
- `String[] list();` On demande la liste des fichiers localisés dans le répertoire.

- Afficher le contenu d'un répertoire en affichant si les éléments de ce
- répertoire sont des fichiers ou des répertoires.
- Dans le cas où il s'agit d'un fichier afficher la capacité physique du fichier.

```
import java.io.File;
public class Application1 {
    public static void main(String[] args) {
        String rep="c:/"; File f=new File(rep);
        if(f.exists()){
            String[] contenu=f.list();
            for(int i=0;i<contenu.length;i++){
                File f2=new File(rep,contenu[i]);
                if(f2.isDirectory())
                    System.out.println("REP:"+contenu[i]);
                else
                    System.out.println("Fichier :"+contenu[i]+"
                    Size:"+contenu[i].length());}}
            else{
                System.out.println(rep+" n'existe pas");}
        }}
    }
```



File f=new File(c:/.../Rep1);
f.list() donne {« rep2 », « rep3 », « f4»,
C'est-à-dire le premier niveau sous le répertoire

EXERCICE 1

Ecrire une application java qui permet d'afficher le contenu d'un répertoire y compris le contenu de ses sous répertoires

- Sérialiser un objet
- Desérialiser un objet

La sérialisation est une opération qui permet d'envoyer un objet sous forme d'un tableau d'octets dans une sortie quelconque.(un fichier)

(Fichier, réseau, port série etc..)

Les applications distribuées utilisent beaucoup ce concept pour échanger les objets java entre les applications via le réseau.

Pour sérialiser un objet, on utilise la méthode **writeObject()** de la classe **ObjectOutputStream**

La désérialisation est une opération qui permet de reconstruire l'objet à partir d'une série d'octets récupérés à partir d'une entrée quelconque.(un fichier)

Pour désérialiser un objet, on utilise la méthode **readObject()** de la classe **ObjectInputStream**.

Pour pouvoir sérialiser un objet, sa classe doit implémenter l'interface **Serializable**

Pour désigner les attributs d'un objet qui ne doivent pas être sérialisés, on doit les déclarer **transient**

Binaire	XML
<ul style="list-style-type: none">• Pas lisible par l'humain• Léger donc...• ... rapide à serializer• Échanges plus limités avec les applis spécifiques	<ul style="list-style-type: none">• Lisible : en entrée de plein de prog aussi• Plus volumineux : des balises partout donc...• ... lent• Portable : échange d'info via des prog entre différents système d'exploitation

On sérialise les attributs : les méthodes sont communes pour toute² la classe, pas

- Pour pouvoir sérialiser, il faut que la classe implémente l'interface Serializable :
`java.io.Serializable` : c'est une interface vide, donc rien à coder pour l'implémenter
- Il faut aussi que les attributs soient serializable (les types simples le sont, les listes aussi)

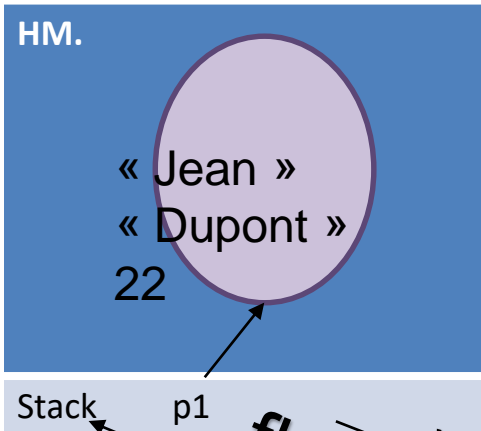
```
Personne p1=new Personne ("toto");  
File f=new File("test.txt");  
FileOutputStream fos=new FileOutputStream(f);  
ObjectOutputStream oos=new ObjectOutputStream(fos);  
oos.writeObject(p1);  
oos.close();  
fos.close();
```

Nb: attention a la gestion des erreurs(exception surveillée)!

```
File f=new File("banque.txt");  
FileInputStream fis=new FileInputStream(f);  
ObjectInputStream ois=new ObjectInputStream(fis);  
Personne p1=(Personne ) ois.readObject();  
ois.close();  
fis.close();
```

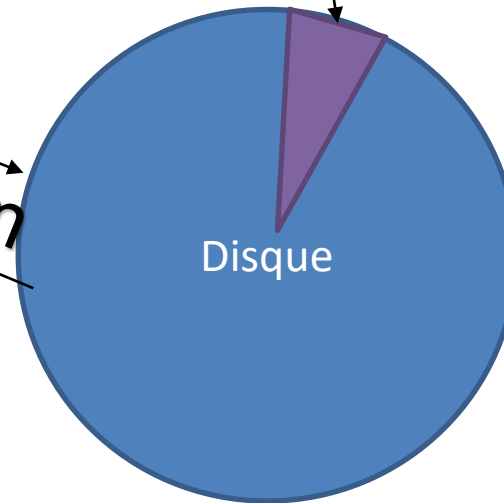
```
System.out.println(" p1 "+p1.toString());
```

Nb: attention a la gestion des erreurs(exception surveillée)!



L'extension choisie ne change
pas le fichier
Crée juste la manière par
default de l'ouvrir

C:/tmp/toto.txt



flux = stream
W (write)
R (read)

```
public static void serialisePersonne() throws  
IOException{  
    Personne p = new Personne(10, "Dupont", "Jean");  
    File f = new File ("c:/tmp/toto.txt");  
    FileOutputStream fos = new FileOutputStream(f);  
    ObjectOutputStream oos = new  
    ObjectOutputStream(fos);  
    oos.writeObject(p);  
    oos.close(); fos.close();}
```

- On peut utiliser le mot clé transient sur un attribut pour qu'il ne soit pas sauvegardé par la sérialisation
- Particulièrement utile pour des données volumineuses de signification éphémère
- Ex : **private transient int age;**
l'age ne sera alors pas sauvegarder lors de la serialisation
(revient à 0 – valeur par défaut – lors d'une sérialisation/deserialization)

SÉRIALISER DE DEUX OBJETS (OU PLUS..)

- But : sérialiser plusieurs objets dans un même fichiers
Il suffit de faire plusieurs writeObject
- On peut sauvegarder n'importe quel type d'objets dans un même fichiers (pas forcément tous de même type) : attention à la récupération
- Pour récupérer plusieurs objets :
 - Première lecture : curseur du stream au début du fichier, lit et récupère l'objet, avance le curseur
 - Deuxième : curseur au niveau du deuxième objet sérialisé dans le fichier, lit l'objet suivant et avance le curseur
- On fait un cast vers le type spécifique : peut déclencher une exception si les objets sérialisés ne sont pas du mêmes types
 - Il vaut mieux tester (via **instanceof**) avant de caster !

- Différences à deux niveaux :
 - XMLEncoder plutôt que ObjectOutputStream

Vers XML :

```
public static void serialisePersonneXML() throws
IOException{
    Personne p = new Personne(10, "Durant", "John");

    File f = new File ("c:/tmp/resultat.xml");

    FileOutputStream fos = new FileOutputStream(f);

    XMLEncoder oos = new XMLEncoder(fos);

    oos.writeObject(p);
    oos.close();
    fos.close();
}
```

Vers binaire :

```
public static void serialisePersonne() throws
IOException{
    Personne p = new Personne(10, "Durant", "John");

    File f = new File ("c:/tmp/toto.txt");

    FileOutputStream fos = new FileOutputStream(f);

    ObjectOutputStream oos = new ObjectOutputStream(fos);

    oos.writeObject(p);
    oos.close();
    fos.close();
}
```

- De même avec XMLDecoder pour la désérialisation
- Il faut les constructeurs par défaut, setters et getters de toutes les classes impliquées

ON PEUT SÉRIALISER UNE LISTE D OBJET

```
public static void serializeXMLArrayList() throws IOException{
    ArrayList<Personne> al = new ArrayList<Personne>();
    Adresse a1 = new Adresse("9 rue rougemont", "Paris");
    Adresse a2 = new Adresse("là-bas", "Paris");
    Personne p0 = new Personne("Jacquot", "Alice", 87, a1);
    Personne p1 = new Personne(10, "Dupont", "Jean");
    Personne p2 = new Personne("Jacqbets", "ze", 17, a2);
    al.add(p0); al.add(p1); al.add(p2);
    File f = new File ("c:/tmp/listeserie.xml");
    FileOutputStream fos = new FileOutputStream(f);
    XMLEncoder oos = new XMLEncoder(fos);
    oos.writeObject(al);
    oos.close();
    fos.close();
}
```

ON PEUT DESÉRIALISER UNE LISTE D OBJET

```
public static void deserialiseXMLliste() throws
IOException, ClassNotFoundException{
    ArrayList<Personne> recup = null;
    File f = new File ("c:/tmp/listeserie.xml");
    FileInputStream fis = new FileInputStream(f);
    XMLDecoder ois = new XMLDecoder(fis);
    recup= (ArrayList<Personne>) ois.readObject();
    ois.close();
    fis.close();
    System.out.println(recup);
}
```

ACCÈS AUX BASES DE DONNÉES VIA JDBC

Pour qu'une application java puisse communiquer avec un serveur de bases de données, elle a besoin d'utiliser les pilotes JDBC (Java Data Base Connectivity)

Les Pilotes JDBC est une bibliothèque de classes java qui permet, à une application java, de communiquer avec un SGBD via le réseau en utilisant le protocole TCP/IP

Chaque SGBD possède ses propres pilotes JDBC.

Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java communiquer avec n'importe quelle source de données via les pilotes ODBC (Open Data Base Connectivity)

Les pilotes ODBC permettent à une application Windows de communiquer une base de données quelconque (Access, Excel, MySQL, Oracle, SQL SERVER etc...)

La bibliothèque JDBC a été conçu comme interface pour l'exécution de requêtes SQL. Une application JDBC est isolée des caractéristiques particulières du système de base de données utilisé.

Pour créer une application élémentaire de manipulation d'une base de données il faut suivre les étapes suivantes :

Chargement du Pilote JDBC ;

Identification de la source de données ;

Allocation d'un objet **Connection**

Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**);

Exécution d'une requête à l'aide de l'objet Statement ;

Récupération de données à partir de l'objet renvoyé **ResultSet** ;

Fermeture de l'objet ResultSet ;

Fermeture de l'objet Statement ;

Fermeture de l'objet Connection.utilisé autre chose, par exemple oracle

- Process qui tourne sur un port de la machine
- Machine à une adresse IP, constituée de plusieurs port pouvant contenir un service, on peut y accéder via son numero de port
- EX : internet port 80
- MySQL : 3306
- Modifiable (pour un port disponible) mais des classiques
- Un serveur peut se démarrer/s'arreter (mysql : server status pour le voir, startup/shutdown pour le changer)
- On travaille pour le moment en local mais rien n'empêche d'accéder à une autre machine

Charger les pilotes JDBC :

Utiliser la méthode `forName` de la classe `Class`, en précisant le nom de la classe pilote.

Exemples:

Pour charger le pilote `JdbcOdbcDriver`:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

Pour charger le pilote jdbc de MySQL:

```
Class.forName("com.mysql.jdbc.Driver") ;
```

Pour créer une connexion à une base de données, il faut utiliser la méthode statique `getConnection()` de la classe `DriverManager`. Cette méthode fait appel

aux pilotes JDBC pour établir une connexion avec le SGBDR, en utilisant les sockets.

Pour un pilote `com.mysql.jdbc.Driver` :

Connection

```
conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/DB",  
"user", "pass" );
```

Pour un pilote `sun.jdbc.odbc.JdbcOdbcDriver` :

Connection conn=

```
DriverManager.getConnection("jdbc:odbc:dsnSCO", "user", "pass" );
```


OBJETS STATEMENT, RESULTSET ET RESULTSETMETADATA

Pour exécuter une requête SQL, on peut créer l'objet Statement en utilisant la méthode `createStatement()` de l'objet Connection.

Syntaxe de création de l'objet Statement

Statement st=conn.createStatement();

Exécution d'une requête SQL avec l'objet Statement :

Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

ResultSet rs=st.executeQuery("select * from PRODUITS");

Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

st.executeUpdate("insert into PRODUITS (...) values(...)");

Pour récupérer la structure d'une table, il faut créer l'objet `ResultSetMetaData` en utilisant la méthode `getMetaData()` de l'objet `ResultSet`.

ResultSetMetaData rsmd = rs.getMetaData();

Pour exécuter une requête SQL, on peut également créer l'objet PreparedStatement en utilisant la méthode `prepareStatement()` de l'objet Connection. Syntaxe de création de l'objet PreparedStatement

```
PreparedStatement ps=conn.prepareStatement("select *  
from PRODUITS where NOM_PROD like ? AND PRIX<?");
```

Définir les valeurs des paramètres de la requête:

```
ps.setString(1,"%"+motCle+"%");  
ps.setString(2, p);
```

Exécution d'une requête SQL avec l'objet PreparedStatement :

Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=ps.executeQuery();
```

Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
ps.executeUpdate();
```

Pour parcourir un ResultSet, on utilise sa méthode `next()` qui permet de passer d'une ligne à l'autre. Si la ligne suivante existe, la méthode `next()` retourne `true`. Si non elle retourne `false`.

Pour récupérer la valeur d'une colonne de la ligne courante du ResultSet, on peut utiliser les méthodes `getInt(colonne)`, `getString(colonne)`, `getFloat(colonne)`, `getDouble(colonne)`, `getDate(colonne)`, etc... colonne représente le numéro ou le nom de la colonne de la ligne courante.

Syntaxe:

```
while(rs.next()){  
    System.out.println(rs.getInt(1));  
    System.out.println(rs.getString("NOM_PROD"));  
    System.out.println(rs.getDouble("PRIX"));  
}
```

L'objet `ResultSetMetaData` est très utilisé quand on ne connaît pas la structure d'un `ResultSet`. Avec L'objet `ResultSetMetaData`, on peut connaître le nombre de colonnes du `ResultSet`, le nom, le type et la taille de chaque colonne.

Pour afficher, par exemple, le nom, le type et la taille de toutes les colonnes d'un `ResultSet` `rs`, on peut écrire le code suivant:

```
ResultSetMetaData rsmd=rs.getMetaData();  
// Parcourir toutes les colonnes  
for(int i=0;i<rsmd.getColumnCount();i++){  
    // afficher le nom de la colonne numéro i  
    System.out.println(rsmd.getColumnName(i));  
    // afficher le type de la colonne numéro i  
    System.out.println(rsmd.getColumnTypeName(i));  
    // afficher la taille de la colonne numéro i  
    System.out.println(rsmd.getColumnDisplaySize(i));  
}  
// Afficher tous les enregistrements du ResultSet rs  
while (rs.next()){  
    for(int i=0;i<rsmd.getColumnCount();i++){  
        System.out.println(rs.getString(i));  
    }  
}
```

Dans la pratique, on cherche toujours à séparer la logique de métier de la logique de présentation.

On peut dire qu'on peut diviser une application en 3 couches:

La couche d'accès aux données: DAO

Partie de l'application qui permet d'accéder aux données de l'application. Ces données sont souvent stockées dans des bases de données relationnelles.

La couche Métier:

Regroupe l'ensemble des traitements que l'application doit effectuer.

La couche présentation:

S'occupe de la saisie des données et de l'affichage des résultats;

D'une manière générale les applications sont orientée objet :

- Manipulation des objet et des classes

- Utilisation de l'héritage et de l'encapsulation

- Utilisation du polymorphisme

D'autres part les données persistantes sont souvent stockées dans des bases de données relationnelles.

Le mapping Objet relationnel consiste à faire correspondre un enregistrement d'une table de la base de données à un objet d'une classe correspondante.

Dans ce cas on parle d'une classe persistante.

Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (Base de données,Fichier, etc..)

- Bonne organisation du code, en trois couches distinctes :
 - Interface (= couche Présentation)
 - Print, scan, affichage etc
 - Métier
 - Traitements algorithmiques, cœur du code
 - Connexion aux données (= couche DAO = Data Access Object)
 - Aux fichiers, à la base de données
- classe persistante = classe dont on sauvegarde les instances (= java bean)

On considère une base de données qui contient une table ETUDIANTS qui permet de stocker les étudiants d'une école. La structure de cette table est la suivante :

Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
ID_ET	int(11)			Non	Aucun	auto_increment
NOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
PRENOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
EMAIL	varchar(25)	latin1_swedish_ci		Non	Aucun	
VILLE	varchar(200)	latin1_swedish_ci		Non	Aucun	

ID_ET	NOM	PRENOM	EMAIL	VILLE
1	A	PA	A@YAHOO.FR	casa
2	B	PB	B@YAHOO.FR	rabat
3	C	PC	C@YAHOO.FR	casa
4	BBCAAC	BBCAAC	ab@yahoo.fr	casa

Nous souhaitons créer une application java qui permet de saisir au clavier un mot clé et d'afficher tous les étudiants dont le nom contient ce mot clé.

Dans cette application devons séparer la couche métier de la couche présentation.

Pour cela, la couche métier est représentée par un modèle qui se compose de deux classes :

La classe `Etudiant.java` : c'est une classe persistante c'est-à-dire que chaque objet de cette classe correspond à un enregistrement de la table `ETUDIANTS`. Elle se compose des :

champs privés `idEtudiant`, `nom`, `prenom`, `email` et `ville`, d'un constructeur par défaut, des getters et setters.

Ce genre de classe c'est ce qu'on appelle un java bean.

La classe `Scolarite.java` :

c'est une classe non persistante dont laquelle, on implémente les différentes méthodes métiers. Dans cette classe, on fait le mapping objet relationnel qui consiste à convertir un enregistrement d'une table en objet correspondant.

Dans notre cas, une seule méthode nommée `getEtudiants(String mc)` qui permet de retourner une Liste qui contient tous les objets `Etudiant` dont le nom contient le mot clé «mc»

Travail à faire :

Couche données :

Créer la base de données « SCOLARITE » de type MySQL

Saisir quelques enregistrements de test

Couche métier. (package metier) :

Créer la classe persistante Etudiant.java

Créer la classe des business méthodes Sclarite.java

Couche présentation (package pres):

Créer une application de test qui permet de saisir au clavier le mot clé et qui affiche les étudiants dont le nom contient ce mot clé.

```
package metier;  
public class Etudiant {  
private Long idEtudiant;  
private String nom,prenom,email;  
// Getters etStters  
}
```

```
package metier;
import java.sql.*; import java.util.*;
public class Sclarite {
public List<Etudiant> getEtudiantParMC(String mc){
List<Etudiant> etds=new ArrayList<Etudiant>();
try {
Class.forName("com.mysql.jdbc.Driver");
Connection conn=
DriverManager.getConnection("jdbc:mysql://localhost:3306/DB_SCO","root","");
PreparedStatement ps=conn.prepareStatement("select * from ETUDIANTS where NOMlike ?");
ps.setString(1,"%"+mc+"%");
ResultSet rs=ps.executeQuery();
while(rs.next()){
Etudiant et=new Etudiant();et.setIdEtudiant(rs.getLong("ID_ET"));et.setNom(rs.getString("NOM"));
et.setPrenom(rs.getString("PRENOM"));et.setEmail(rs.getString("EMAIL"));
etds.add(et);}}
catch (Exception e) { e.printStackTrace(); }
return etds;
}
```

```
package pres;
import java.util.List;import java.util.Scanner;
import metier.Etudiant;import metier.Scolarite;
public class Presentation {
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Mot Clé:");
        String mc=clavier.next();
        Scolarite metier=new Scolarite();
        List<Etudiant> etds=metier.getEtudiantParMC(mc);
        for(Etudiant et:etds)
            System.out.println(et.getNom()+"\t"+et.getEmail());
        }
    }
```

- Plusieurs tables = plusieurs classes métiers **et plusieurs classes DAO** (pour plus de lisibilité et modularité)

Cadre : une table de personne, une classe Personne DAOJDBC pour chercher dans la base

Un jour, on veut faire la même chose mais plus en jdbc, pour par ex. accéder à un fichier excel.

- **Généricité** : créer une interface PersonneDAO qui liste les méthodes à implémenter par un DAO pour nos classes. L'utiliser permet d'avoir une classe Personne restant valide quel que soit le type de données à consulter.
- But : **Modularité** !

```
public interface DAO<T, PK> {  
    T findById(PK Id);  
    List<T> findAll();  
    void create(T obj);  
    void update(T obj);  
    void delete(T obj);  
}
```

```
public interface PersonneDAO extends DAO<Patient, Integer> {  
  
}
```