Travaux Pratiques de Programmation Orientée Objet en Java

Environnement de travail : l'IDE ***Eclipse*** sera notre principal environnement de travail.

Intérêt :

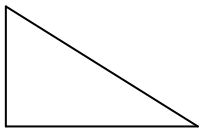
Les exercices qui suivent permettent, d'une part de fixer les notions fondamentales de classe, d'objet, d'encapsulation, la visibilité des méthodes et des champs, les types de base, les différents types d'opérateurs, les structures de contrôle, la notion de paquetage, bref tous les aspects théoriques nécessaires à la bonne compréhension du langage JAVA.

Exercice 1 : (A quoi sert le tableau de chaîne mis en argument dans la méthode principale ***main*** en Java ?)

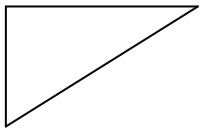
Créer un programme qui permet de calculer et d'afficher la somme de deux entiers donnés par l'utilisateur sur la ***ligne de commande (utilisation de MS-DOS ou un Shell UNIX)***.
On nomme en java, ces deux entiers, ***arguments de la ligne de commande****.*
Attention *: les entiers que vous saisissez seront considérés comme des chaînes de caractères. Donc dans le programme vous penserez à les convertir en nombres entiers en utilisant la méthode statique* **parseInt** *de la classe* **Integer** *du package java.lang.*
Exemple : si on a : String p = "24" ; on fait : int k = Integer.parseInt (p) ; //et on a k=24

Exercice 2 :

On veut réaliser un programme (une classe **DessineTriangle**) qui dessine :
un triangle rectangle dirigé soit vers le haut comme suit:



soit dirigé vers le bas comme suit:



NB : Vous utiliserez des astérix pour représenter les côtés du triangle.
On prévoira deux méthodes dans cette classe:
public void dessineTriangleBas (int m) : cette méthode permettra de dessiner le triangle orienté vers le bas. Le paramètre m passé en argument à la méthode sera la taille du tableau.
public void dessineTriangleHaut (int n) pour dessiner le triangle orienté vers le haut.
Créer ensuite une seconde classe principale **TestTriangle** pour tester le code ci-dessus.
On imposera à l'utilisateur de saisir une taille comprise entre 3 et 10 et l'utilisateur n'a droit qu'à 3 essais au maximum pour donner une valeur acceptable. Passé ce nombre, on met fin à l'exécution du programme (on peut utiliser pour cela **System.exit (0)**).
Le choix de l'orientation du triangle sera la valeur 0 pour dessiner vers le bas et la valeur 1 pour dessiner vers le haut. Le choix sera aussi demandé à l'utilisateur.
L'exécution de votre main sera comme suit :

Donner la taille du triangle : 5
Vous avez choisi de dessiner un triangle dont le coté mesure 5 lignes
Donner le choix de l'orientation : 1
Le programme dessine un triangle orienté vers le haut :
*
* *
* * *
* * * *
* * * * *
Merci d'avoir dessiner ce triangle.

Exercice 3 :

Faire un programme permettant de calculer la somme de la série harmonique suivante : 1+1/2+1/3+1/4+…….+1/n.
La valeur n sera un entier strictement positif demandé à l'utilisateur. L'utilisateur n'a que 5 possibilités pour choisir une valeur comprise nécessairement entre 1 et 100 sinon on met fin au programme.

Exercice 4 :

Réaliser un programme qui permet de calculer les racines d'une équation du second degré en x. ($ax^2 + bx + c$) a, b et c seront donnés au clavier. Voir les cas où le discriminant est nul, négatif strictement ou positif strictement.

Exercice 5 : (*manipulation d'objets*)

Réaliser une classe Factoriel qui permet de calculer le factoriel d'un entier. Il faut ici disposer d'une méthode pour faire uniquement le calcul demandé et une autre méthode pour afficher le résultat du calcul.
Créer aussi une classe de test. On demandera à l'utilisateur de fournir l'entier dont on veut calculer le factoriel au clavier. Le programme rejettera bien entendu les entiers négatifs. On ne calculera que le factoriel des entiers entre 0 et 20.
On verra dans les exercices suivants, une méthode plus évoluée pour calculer le factoriel de grands nombres.

Exercice 6

Réaliser une classe **Point** qui permet qui représenter les points cartésiens du plan.
Un point du plan sera caractérisé par ses deux coordonnées cartésiennes (abscisse et ordonnées) de type flottant et sa couleur qui est un scalaire de type byte.
On prévoira :

- un constructeur à deux arguments pour instancier un point avec seulement ses deux coordonnées,
- un constructeur avec trois arguments (coordonnées et couleur), Ce dernier devra faire appel au constructeur à deux arguments.
- un constructeur sans argument pour créer le point origine. Ce dernier devra faire appel au constructeur à deux arguments.
- une méthode **translater** qui permet de déplacer le point,
- une méthode **affiche** qui imprimer sur la console les propriétés du point,
- une méthode **coïncide** qui prend en argument un point (à comparer avec le point courant) et qui renvoie un booléen (vrai si les deux points coïncident, faux dans le cas contraire).

Vous créerez une classe de test **TestPoint** dans le même package, pour tester votre classe **Point**. Créer aussi une classe **TesPoint_2** dans un package différent et tester.

Exercice 7 :(*manipulation d'objets membre)*

Réaliser une classe **Cercle** permettant de représenter des cercles dans le plan. Un cercle est caractérisé par son centre et son rayon (un scalaire de type double). Le centre du cercle est un objet point du plan. Donc, les champs d'un objet point seront son centre et son rayon.
Créer un constructeur qui permet d'initialiser correctement un objet cercle.
Créer une méthode **surface** qui permet de calculer la surface du cercle.
Créer une méthode **affiche** qui permet d'afficher les caractéristiques du cercle.
Créer une méthode **imprime** qui permet d'afficher la surface du cercle.

Exercice 8 :
Concevoir et implémenter une application de gestion de stocks (de marchandises).
Exercice 9 :
Créer une classe **CompteBancaire** qui crée et manipule des comptes en banque.
Un compte sera identifier par le solde, le nom et le prénom du titulaire, le sexe et le découvert initial.
Imaginer et créer les méthodes nécessaires de manipulation de comptes bancaires.
Tester.
Exercice 10 :
Modéliser et créer une classe permettant de gérer le personnel administratif d'une entreprise. On supposera q'un employé sera identifié par son nom, son prénom, son age, son sexe, son numéro de matricule, sa fonction au sein de l'entreprise.
Exercice 11 :
Ecrire une classe **ChaineInverse** permettant d'inverser des chaînes de caractères (ex : la chaîne « bonjour » a comme inverse la chaîne « ruojnob ».
On prévoira une méthode **inverseChaine** qui se contente de prendre une chaîne et de renvoyer la chaîne inversée.
L'en-tête de cette méthode sera **public String inverseChaine (String s).**
On prévoit aussi une méthode **imprimeChaine** qui se contente d'afficher sur la console la chaîne inversée.
L'en-tête de cette méthode sera **public void imprimeChaine ( ).**
Vous créerez ensuite une classe de test **TestChaine** où vous construisez un objet chaîne pour l'inverser et afficher le résultat de l'inversion.

Vous réaliserez une autre **statique** de cette classe appelée **ChaineInverseStat.**
Exercice 12 :

On se propose de réaliser une classe **NumberOcc** qui permet qui déterminer le nombre d'occurrences d'un caractère dans une chaîne donnée.
On prévoit une méthode **getNbOcc** qui permettra de compter le nombre de fois qu'un caractère (recherché) apparaît dans une chaîne.
L'en-tête de cette méthode sera **public static int getNbOcc (String s, char c).**
Vous prévoyez également une méthode **public static void afficheNbOcc ( )** qui affiche le nombre d'occurrences trouvé.
Dans cette même classe créez une méthode **getNbVoy** qui compte le nombre de voyelles dans une chaîne donnée.
L'en-tête de la méthode est **public static int getNbVoy (String chaine).** Créez également une méthode **public static void afficheNbVoy ( )** affichant le nombre de voyelles trouvé.

Exercice 13 :

Écrire la fonction (dans une classe **ExisteMotDansListe**)

```
public  static boolean existeDans (String mot, String[] liste)
```

qui renvoie vrai si le mot `mot` est dans `liste` et faux sinon.

Exercice 14:

Réaliser une classe **RechercheChaine** qui permet de rechercher une chaîne dans une autre chaîne donnée. Cette méthode retourne vrai si la chaîne cherchée est trouvée, faux sinon.
Vous créerez donc une méthode **trouveChaine** qui fera ce qui est demandé.
L'en-tête de la méthode
static boolean trouveChaine (String chaine, String chainerecherche).Tester.

Exercice 15:

Créer une classe dans laquelle vous mettez les méthodes suivantes :

a)

```
static int extraireSousChaine (String chaine, String sousChaine)
```

qui renvoie le nombre de fois que la chaine *sousChaine* est dans la chaine *chaine*.

b)

```
static int [] extraireChiffres (String s){...}
```

qui renvoie un tableau contenant tous les chiffres éventuellement présents dans s.

NB : on peut se servir de l'exercice 14.

Exercice 16:

Une méthode **boolean estSuffixe (String u, String v)** qui retourne true si le mot u est suffixe de v, false sinon.
Une méthode **String extraire (String s, int début, int fin)** qui retourne la sous chaine extraite de s de l'indice début à l'indice fin-1.

Exercice 17 :

Concevoir un programme où vous définirez les fonctionnalités suivantes :

- créer une méthode prenant une ***chaîne*** et un ***caractère*** en paramètres et rendant une ***chaîne*** où toutes les occurrences du caractère contenues dans la chaîne passée en paramètre sont converties en majuscules.
- Créer une méthode qui permet d'afficher le résultat de la méthode précédente.

Créer une classe de test.

Exercice 18

Créer une méthode qui étant donné une phrase, renvoie sous forme de tableaux de caractères, l'ensemble des voyelles contenues dans cette phrase.

Exercice 19 :

Réaliser une classe **TriString** permettant de trier par ordre croissant des tableaux de chaînes de caractères.
On prévoira :

- la méthode **private static String [ ] triTabChaine (String tab [])** qui prend un tableau de chaînes de caractères et renvoie en résultat le tableau trié.
- la méthode **private static void afficheTabTri (String tab[])** qui affiche les éléments d'un quelconque tableau de chaîne.

- la méthode **private static void echange (String t[] ,int i, int j)** qui permet d'échanger dans le tableau t les valeurs des cases aux indices i et j (c-à-d dans t[i] on place t[j] et dans t[j] on met t[i]).

*NB : la méthode **int compareTo (String anotherString)** de la classe **String** du paquetage java.lang permet de comparer lexicographiquement deux chaînes de caractères.*

*Vous utiliserez uniquement les deux méthodes **compareTo** et **echange** pour implémenter la méthode triTabChaine.*

Exercice 20 :

Réaliser une classe **InverseArray** qui permet de renverser les éléments d'un tableau donné.
Il s'agira de créer une méthode **public static int [ ] inverseTab (int t [])** qui prend un tableau en argument et renvoie un tableau inversé (ie le dernier élément du tableau transmis devient le premier élément du tableau renvoyé, l'avant-dernier devient le deuxième,….., le premier devient le dernier).
On prévoira une méthode affiche **public static void afficheTabInverse ( )** qui ne peut afficher que les éléments du tableau inversé.

Exercice 21 :

La méthode **char [ ] toCharArray ( )**, appliqué à un objet de type String, construit un tableau de caractères.
Le but de cet exercice est d'implémenter cette méthode et non de l'utiliser.
Réaliser une classe **ToCharArray01** permettant de construire des tableaux de caractères à partir d'objets de type **String**.
Créer donc la méthode **public static char [ ] CharArrayFromString (String s)** qui construit un tableau de caractères dont les éléments sont les caractères formant le String s.(par exemple si **s** est la chaîne « bonjour », le tableau qu'on en déduit est { 'b','o','n','j','o','u','r'}).
Créer aussi la méthode **public static void afficheTabChar (char s [ ])** qui affiche les éléments d'un tableau de caractères.

Exercice 22 :

Réaliser une classe **CreateStringArray** qui permet de créer un tableau de chaînes de caractères à partir d'une phrase (chaîne). Une phrase étant constituée de mots séparés par un espace, l'espace sera donc le moyen que vous utilisez pour délimiter les différents éléments de votre tableau.
On prévoira :

- la méthode **public static String [] arrayFromString (String s)** qui prend une chaîne en paramètre et renvoie un tableau dont les éléments sont des chaînes de caractères (qui sont les différents mots constituant la phrase (chaîne) s)).
- la méthode **public static void afficheString (String [] s)** qui affiche les éléments d'un tableau de chaînes.

NB : la méthode **java.util.StringTokenizer** permet de connaître les différents éléments dans une chaîne de caractères selon un (des) délimiteur (s).

Exemple : si vous avez une chaîne s = « je viens**,** auprès de vous **;** en temps que »

```
StringTokenizer st = new StringTokenizer(s, « , ; ») ;// , et ; sont les délimiteurs
  while (st.hasMoreTokens () )
     { st.nextToken () ;              // renvoie trois éléments {je viens} {aupres de vous} et
                                      // {en temps que}
```

}

Exercice 23 :

On veut réaliser une classe **TriTabEntier** qui permet de trier par ordre croissant un tableau d'entiers.
On créera une méthode **public static void tabTrie (int t[ ])** qui permet donc de trier un tel tableau.
Pour trier les valeurs d'un tableau, il va être nécessaire de permuter les valeurs contenues dans les différentes cases du tableau. Pour cela, une fonction de permutation, qui sera appelée **"echanger",** doit être écrite. Cette fonction prend en argument un tableau et deux entiers i et j. Elle récupère la valeur contenue dans la iième case du tableau, affecte à cette case la valeur contenue dans la jième case, puis affecte à la jième case l'ancienne valeur de la iième case.
Ecrire cette méthode dont l'en-tête sera : **public static void echange (int t [], int i, int j).**
On créera une méthode **public static int indiceMIN (int t[],int imin,int imax)** qui permet de connaître l'indice de la case du tableau contenant le plus petit élément.
On utilisera donc la méthode **indiceMIN** et **echange** pour implémenter la méthode **tabTrie**.
NB : en fait on veut implémenter l'algorithme di tri par sélection.
Le tri par sélection est l'un des tris les plus instinctifs. Le principe est que pour classer n valeurs, il faut rechercher la plus grande valeur et la placer en fin de liste, puis la plus grande valeur dans les valeurs restantes et la placer en avant dernière position et ainsi de suite...
Mais ici dans cet exercice on inverse la tendance c-à-d on cherche le plus petit élément qu'on place en début de tableau et ainsi de suite jusqu'au plus grand élément.

Exercice 24 :

Réaliser un programme qui crée un tableau d'entiers à partir d'un nombre fourni au hasard par le programme. Donc le programme fournit un chiffre n et on crée un tableau dont les éléments sont les entiers inférieurs ou égal à n.

Exercice 25 :

Réaliser une classe **Point** qui permet qui représenter les points cartésiens du plan.
Un point du plan sera caractérisé par ses deux coordonnées cartésiennes (abscisse et ordonnées) de type flottant et sa couleur qui est un scalaire de type byte.
On prévoira :

- un constructeur à deux arguments pour instancier un point avec seulement ses deux coordonnées,
- un constructeur avec trois arguments (coordonnées et couleur), Ce dernier devra faire appel au constructeur à deux arguments.
- un constructeur sans argument pour créer le point origine. Ce dernier devra faire appel au constructeur à deux arguments.
- une méthode **translater** qui permet de déplacer le point,
- une méthode **affiche** qui imprimer sur la console les propriétés du point,
- une méthode **coïncide** qui prend en argument un point (à comparer avec le point courant) et qui renvoie un booléen (vrai si les deux points coïncident, faux dans le cas contraire).

Vous créerez une classe de test **TestPoint** dans le même package, pour tester votre classe **Point**.

Exercice 26 :

Réaliser une classe **Cercle** permettant de représenter des cercles dans le plan. Un cercle est caractérisé par son centre et son rayon (un scalaire de type double). Le centre du cercle est un objet point du plan. Donc, les champs d'un objet point seront son centre et son rayon.
Créer un constructeur qui permet d'initialiser correctement un objet cercle.
Créer une méthode **surface** qui permet de calculer la surface du cercle.
Créer une méthode **affiche** qui permet d'afficher les caractéristiques du cercle.

Exercice 27 (*exercice relatif aux tableaux multidimensionnels)*

Réaliser une classe **TabMultiDim** contenant les méthodes suivantes. :

- **affiche (double t[ ][ ])** à raison d'une ligne d' écran pour une ligne de tableau.
- **boolean regulier (double t [ ][ ])** : qui teste si le tableau est régulier c-à-d si toutes les ses lignes ont la même taille .
- **double [ ] sommeLignes (double t [ ][ ])** fournit un tableau correspondant aux sommes des différentes lignes du tableau t.
- **double [ ][ ]somme (double t1[ ][ ], double t2[ ][ ]) :**s'assure que les tableaux t1 et t2 sont réguliers et de mêmes dimensions et fournit dans ce cas leur somme en résultat (ici il faut comprendre que si **resultatTab** est le nom du tableau renvoyé par cette méthode , on doit avoir **resultatTab [i][j] = t1[i][j] + t2[i][j]**), dans le cas contraire, elle fournit une référence nulle.

Créer une classe de test.

Exercice 28 *(gestion d'un répertoire téléphonique)*

Réaliser une classe **Repertoire** permettant de gérer un répertoire téléphonique associant un numéro de téléphone (chaîne de caractères) à un nom. Pour faciliter les choses, on prévoira une classe **Abonne** destinée à représenter un abonné et disposant des fonctionnalités indispensables. Un abonné est identifié par son nom et un numero (deux objets de la classe String). Il faut donc prévoir un constructeur pour bien créer un objet Abonne, une méthode **getNom** qui renvoie le nom de l'abonné et une méthode **getNumero** qui donne le numero de l'abonné.
La classe **Repertoire** devra disposer des fonctionnalités suivantes :

- un constructeur recevant un argument de type entier précisant le nombre maximum d'abonnés que pourra contenir le répertoire.
- une méthode **public boolean addAbonne (Abonne a)** permettant d'ajouter un nouvel abonné ; elle renvoie *false* si le répertoire est plein, *true* sinon.
- une méthode **public String getNumero (Abonne e)** fournissant le numero associé à un nom d'abonné fourni en paramètre,
- une méthode **public int getNAbonnes ( )** qui fournit le nombre d'abonnés figurant dans le répertoire.
- une méthode **public Abonne getAbonne (int num)** fournissant l'abonné de rang num.
- une méthode **public Abonne [ ] AbonnesTries ( )** fournissant un tableau des différents abonnés rangés par ordre alphabétique. Il faut créer d'abord une méthode **public void echanger (Abonne [ ] e, int i, int j)** qui permet de permuter deux abonnés pris quelconques dans l'ensemble des abonnés, la méthode AbonnesTries fera appel à cette dernière).

Créer une classe de test.

Exercice 29

Ecrire une classe **TabUtil** disposant des fonctionnalités suivantes :

- une méthode ***genere*** qui prend en argument un entier et renvoie un tableau des n premiers nombres impairs,
- une méthode ***somme*** qui reçoit en paramètres deux tableaux d'entiers de même taille et qui renvoie un tableau représentant la somme de ces deux vecteurs.
- une méthode ***affiche*** qui affiche les éléments d'un tableau d'entiers reçu en argument.
- une méthode ***incremente*** qui reçoit en argument un tableau d'entiers et renvoie ce même tableau mais chaque case du tableau d'indice impair est incrémentée de la valeur q'elle contient.

Créer un programme de test.

Exercice 30

Réaliser une classe **Etudiant** disposant des fonctionnalités suivantes :
On suppose qu'il y a deux sortes d'étudiants : les étudiants nationaux, les étudiants étrangers
Un étudiant du premier type est identifié par son **nom**, son **prénom** et son **age**, pour un étudiant du second type on rajoute le pays d'origine.
On prévoira donc deux constructeurs qui permettent d'initialiser correctement un étudiant (le deuxième constructeur appellera le premier).

On prévoie deux méthodes **afficheNationaux** qui imprime sur la console l'identité d'un étudiant national et **afficheEtrangers** qui fait pareil pour un étudiant étranger.
On prévoit une méthode **getNom** ( ), **getPrenom** ( ), **getPays** ( ) qui renvoient respectivement le **nom**, le **prénom** et le **pays** d'origine d'un étudiant.
On créera une classe **GroupeTD** qui permet de lier un étudiant à un groupe de TD.
Un groupe de TD est caractérisé par un tableau d'étudiants de taille maximum **maxEt** et le nombre d'étudiants (variable statique) effectivement contenus dans cette liste d'étudiants.
On prévoira un constructeur qui permet de créer une liste initialement vide de groupe de TD dans laquelle on pourra insérer des étudiants.
On prévoira une méthode **public void ajouterEtudiant (Etudiant e)** qui permet d'ajouter un étudiant à un groupe de TD. Pour ajouter un étudiant il faut s'assurer que vous n'avez pas dépasser la taille du tableau.
Créer une méthode **afficherListe** qui permet d'afficher tous les étudiants d'un groupe de TD. Dans cette méthode on fera appel aux méthodes **afficheNationaux** et **afficheEtrangers.**

Exercice 31 :

Ecrire une classe **ConjugVerbe** qui permet de conjuguer au présent de l'indicatif un verbe du premier groupe (verbe terminer par er). On supposera que le verbe est régulier ; autrement dit que l'utilisateur ne fournira pas un verbe comme manger.
On prévoira :

- un constructeur qui permet d'initialiser le verbe à conjuguer,
- une méthode **control** qui permet de demander à l'utilisateur de saisir un autre verbe si celui qui est fourni par le constructeur n'est pas du premier groupe. L'en-tête de cette méthode sera **String control ( )**.
- une méthode **affiche** qui les éléments d'un tableau de chaînes qui sera appelée à chaque fois qu'on veut afficher le résultat de la conjugaison.

Voici le programme de test qu'il faut utiliser:

```
public class TestConjugVerbe {
  public static void main (String [] args) {
   ConjugVerbe conj = new ConjugVerbe ("attendre");
   conj.affiche (conj.conjuguer ( ));
   }
}
```

A la sortie, on a :
le verbe n 'est pas du premier groupe
Donnez un autre verbe **faire**
le verbe n 'est pas du premier groupe
Donnez un autre verbe **demander**
je demande
tu demandes
il/elle demande
nous demandons
vous demandez
ils/elles demandent
NB : on ne demande pas de gérer si le verbe est régulier ou non, le verbe doit simplement se terminer par **er**. C'est à l'utilisateur maintenant d'éviter des verbes comme **manger**.

Exercice 32 :

Ecrire un programme qui lit un mot au clavier et qui indique le nombre de voyelles présentes dans le mot et pour chaque voyelle, le nombre d'occurrences trouvées dans ce mot.
Il faut créer trois méthodes : une méthode **numberVoy** qui renvoie le nombre de voyelles dans le mot, une méthode **numberOccVoy** qui donne le nombre de fois qu'une voyelle est présente dans le mot et une méthode **imprime** qui affiche sur la console les résultats de deux premières méthodes.

Exercice 33 :

Réaliser une classe de tri de tableaux de chaînes de caractères. Il faut que tous les éléments du tableau soient complètement écrits en majuscules avant de réaliser le tri.
Il faut s'assurer qu'un élément du tableau à trier ne contient aucun chiffre et que la taille maximale d'un élément du tableau ne dépasse pas 10 caractères.

Exercice 34 : (calcul de factorielle de grands nombres avec le type **BigInteger**)

Nous avions vu un moyen de calculer le factorielle de « petits nombres » dans les TPs précédents. Il existe, dans le paquetage **java.math**, une classe nommée **BigInteger** qui recèle de méthodes permettant de calculer le factorielle de grands nombres.
Réaliser une classe **FactBigInteger** disposant les fonctionnalités suivantes :
Vous disposerez :

- d'un constructeur pour initialiser le nombre dont on veut calculer le factorielle (ce nombre sera de type entier **long**).
- d'une méthode **long controls ( )** qui vérifie si le nombre est positif ou non et le renvoie. On n'acceptera que les nombres positifs.
- d'une méthode **BigInteger fact ( )** pour calculer le factorielle.
- d'une méthode **afficher (long r)** pour afficher le résultat.

- *NB : n'utilisez aucune instruction import, pour faire usage d'une classe de l'API, recourez à son nom complet.*

Voici une classe de test :

```
public class TestFactBigInteger {
  public static void main(String[] args) {
    int k = Integer.parseInt (javax.swing.JOptionPane .showInputDialog ("donnez un entier positif") );
    FactBigInteger fa = new FactBigInteger(k);
    javax.swing.JOptionPane.showMessageDialog (null,fa.fact (),"Résultat du factorielle",javax.swing.JOptionPane.INFORMATION_MESSAGE,null);

  }
}
```

Un exemple d'exécution (j'ai utilisé ici des boîtes de saisie, des boîtes de message et la console) :

le nombre est négatif *(sur la console)*

Exercice 35 :

Réaliser une classe **ChaineFactoriel** qui prend un tableau de chaînes de caractères à partir duquel on construit un autre tableau d'entiers (en fait de long) dont on calculera le factoriel de chaque élément. Les éléments du tableau d'entiers sont obtenus par conversion en entier des éléments du tableau de chaînes (chaque élément de ce dernier est une chaîne constituée de caractères dont on peut connaître le code ASCII). Le résultat du factoriel sera aussi un tableau d'entiers.

Il faudra créer une :

-méthode **long [ ] converseStringToLong (String ch [])** pour effectuer la transformation du tableau de chaînes en tableau de long

- une méthode **java.math.BigInteger [ ] calculFact (long lg [ ])** pour le calculer le factoriel.
- une méthode **void imprime ( )** pour afficher le tableau résultat du calcul du factoriel.

Pour le calcul du factoriel utilisez l'exercice précédent.

Exercice 36

Réaliser une classe **Etudiant** disposant des fonctionnalités suivantes :
On suppose qu'il y a trois sortes d'étudiants, les étudiants nationaux, les étudiants étrangers et les étudiants sportifs. Un étudiant du premier type est identifié par son **nom**, son **prénom** et son **age**, pour un étudiant du second type on rajoute le **pays d'origine** et pour un étudiant du troisième type, on rajoute le **sport** pratiqué.
On considère qu'on a une classe **Etudiant** et deux autres classes **EtudiantEtranger** et **EtudiantSportif** qui dérivent toutes deux de la première classe.
On prévoira donc :

- un constructeur qui permet d'initialiser correctement un étudiant
- un constructeur qui permet d'initialiser correctement un étudiant étranger (ce constructeur appellera le premier).
- un constructeur qui permet d'initialiser correctement un étudiant sportif (ce constructeur appellera le premier).
- une méthode **affiche** (pour chacune des trois classes) qui imprime sur la console l'identité d'un étudiant.

On prévoit dans la classe **Etudiant** les méthodes **getNom** ( ), **getPrenom** ( ), **getAge** ( ) qui renvoient le nom, le prénom et l'age d'un étudiant.
- on redéfinira convenablement la méthode affiche (en faisant appel à **affiche** de **Etudiant**) dans **EtudiantEtranger** et **EtudiantSportif.**
Maintenant on créera une classe **GroupeTD** qui permet de lier un étudiant quelconque à un groupe de TD.
Un groupe de TD est caractérisé par une liste d'étudiants (un tableau d'étudiants de taille maximum **maxEt)** et le nombre d'étudiants (variable statique) effectivement contenus dans cette liste d'étudiants.
On prévoit un constructeur qui prend en argument la liste d'étudiants et permettant ainsi de créer une liste initialement vide d'étudiants.
On prévoira une méthode **public void ajouterEtudiant (Etudiant e)** qui permet d'ajouter un étudiant à un groupe de TD. Pour ajouter un étudiant il faut s'assurer que vous n'avez pas dépassé la taille du tableau.
Créer une méthode **afficherListe** qui permet d'afficher tous les étudiants d'un groupe de TD. Dans cette méthode on fera appel à la méthode **affiche** (mise en œuvre du polymorphisme)**.**
On donne un programme de test :

```
public static void main (String args [ ])
  {GroupeTD gt = new GroupeTD(4);
  gt.ajouter (new Etudiant (" Sene " , " Pierre " ,12));
  gt.ajouter (new EtudiantSportif (" Fall " , " Fatou " ,5, " Natation "));
  gt.ajouter (new EtudiantEtranger (" Ndiaye " , " Moussa " ,  20 ," Senegal "));
 gt.afficherListe ( );
    }
  }
```

La sortie du programme sera :
Etudiant : nom = Sene prenom = Pierre age = 19
Etudiant Sportif : nom = Fall prenom = Fatou age = 18 sport pratiqué = Natation
Etudiant Etranger : nom = Ndiaye prenom = Moussa age = 20 pays d origine =Senegal.

Exercice 37 (méthode **equals** de la classe **Object**)

On considère le petit programme suivant :

```
public class Point
{private  int x, y ;
 public Point (int x, int y)
    {this.x = x ;
      this.y = y;
     }
public static void main (String args [])
 {Point a = new Point (1, 2);
   Point b = new Point (1 ,2) ;
System.out.println ("comparaison de deux points"+a.equals (b));
 }
}
```

La sortie de ce programme donne : **"comparaison de deux points" faux**
Autrement ce résultat signifie que les objets a et b ne coincident pas (ne sont pas les mêmes) alors qu'on s'attendait au résultat contraire.
Expliquez pourquoi. Et proposez un rectificatif pour que le résultat donne **"comparaison de deux points" vrai.**

Exercice 38 (interface et polymorphisme)
On donne l'interface ci-dessous qu permet de manipuler des valeurs de types primitifs :

```
interface Affichable {
                        void affiche ( );
                      }
```

A partir de cette interface, définissez deux classes **Entier** et **Flottant** qui disposent toutes deux d'un constructeur pour créer un entier, respectivement un flottant (chacune de ces classes aura donc un champ qu'on peut nommé **valeur**) et d'une méthode **affiche** pour afficher l'entier respectivement le flottant.
Créer la classe **HertPoly** disposant des fonctionnalités suivantes :

- on dispose d'un champ de type tableau d'Affichable destiné à contenir une liste d'entiers et/ou de flottant, d'un champ static qui est le nombre actuel d'éléments dans cette liste.
- d'un constructeur qui permet d'initialiser le tableau
- on dispose aussi d'une méthode **public void ajoutValeur (Affichable a)** qui permet d'ajouter un entier ou un flottant dans le tableau
- on dispose d'une méthode **public void affiche ( )** pour imprimer tous les éléments de la liste (les entiers et/ou les flottants)

Voici une classe de test :

```
public class TestHer
{ public static void main (String [] args) {
        HerPoly h = new HerPoly (3);
        h.ajoutValeur (new Entier (25));
        h.ajoutValeur (new Flottant (1.25f)) ;
        h.ajoutValeur (new Entier (42)) ;
        h.affiche ( );}}
```

Sortie du programme :
Entier de valeur 25
Float de valeur 1.25
Entier de valeur 42

Exercice 39 (Normaliser un numéro de téléphone)

On veut créer une classe **NumberTelNorm** qui permet de normaliser un numéro de téléphone :
si le numéro est à 9 chiffres, ajouter en tête (33)
si le numéro est à 10 chiffres avec un zéro en tête, supprimer ce zéro et ajouter (33) .
exemple:
144553823 --> (33)144553823
0144553823 --> (33)144553823
Le numéro de téléphone à normaliser sera considéré comme un champ.
On prévoira :

- un constructeur qui permet d'initialiser ce champ.
- une méthode **controls** qui vérifie si le numéro contient des lettres alphabétiques. Au cas échéant, le numéro est rejeté et on demande à l'utilisateur d'en fournir un autre jusqu'à ce qu'il donne un numéro sans lettres. On renvoie alors ce numéro.
- une fonction **getNumeroNormalise** qui étant donné un numéro de téléphone rend le numero de telephone normalisé.
- Ume méthode **imprime** qui affiche le numéro normalisé.
-

Exercice 40 (vecteur dynamique : la classe ArrayList)

En Java, l'inconvénient d'utiliser les tableaux est le problème de stockage des données ; on ne plus dans un tableau un nombre d'éléments au plus égal à sa taille. Toute tentative d'insertion d'un élément en fin de tableau lève toujours une exception de type **ArrayIndexOutOfBoundsException** qui, si elle n'est interceptée et traitée entraîne l'arrêt brutal du programme.
Pour contourner ce problème, on peut faire usage de vecteurs dynamiques, en l'occurrence utiliser la classe **ArrayList** du paquetage ja**va.util**. Cette classe permet l'insertion d'éléments en fin de liste.
On se propose dans cet exercice de trier un vecteur dynamique contenant des chaînes de caratères.
Le squelette de cette est donnée ci-dessous, les commentaires dans l'en-tête des méthodes décrivent clairement ce qu'il faut implémenter :

```
public class TestTriArrayChaine {

/** rend l'indice du plus petit élément de 'v' compris entre les indices 'imin' et 'imax'
 /* les elements de v sont des chaînes de caractères , imin < imax*/

public static int indice_du_plus_petit (ArrayList v,int imin,int imax)
{

   // rajouter le code de cette méthode.

}

/* échange les éléments 'tab'['i'] et 'tab'['j'] */
public static void echanger (ArrayList v, int i, int j)
{
   // rajouter le code de cette méthode

}
```

```
/* les 'n' premiers éléments de 'v' sont triés par ordre croissant */
public static void triSelectionEchange (ArrayList v,int n)
  {
       // rajouter le code de cette méthode

  }
}// fin de la classe
```

NB : il y a dans la classe ArrayList deux méthodes d'instances qui vous intéresserons : une méthode **get (int)** qui permet de récupérer l'élément dont la position est l'indice passé en paramètres et une méthodes **set (int i, Object o)** qui permet d'insérer dans la liste l'élément o à la position i.
Voici une classe de test

```
public class TestArrayList
{public static void main(String[] args) {
 ArrayList v = new ArrayList();
 v.add("bonjour");v.add("durand");v.add("au revoir");v.add("dupond");v.add("durand");
 TestTriArrayChaine .triSelectionEchange(v,v.size());
 System.out.println(v);
 }
}
```

Exercice 41 (maximum d'une ArrayList)

Réaliser une classe **TestNumber** qui permet de trouver le plus grand élément d'une liste de Number
Voici le squelette de cette classe :

```
public class TestNumber {

/** rend le Number le plus grand de cet ArrayList non vide de Number */
public static Number getMaximum (ArrayList v)
{
       Number max = (Number) v.get (0);

        // rajouter le code manquant
   return max;
}
/** remplace les occurrences de 'ancien' par 'nouveau' dans le vecteur 'v' */
public static void remplacer (ArrayList v, Object ancien, Object nouveau)
{
   // rajouter le code de cette méthode
}
 public static void main (String[] args) {
 ArrayList v = new ArrayList ();
 v.add (new Integer (6));          v.add (new Double (8.2));
 v.add (new Short ((short) 10)); v.add (new Integer (-16));
 v.add (new Float(4));
 System.out.println (getMaximum (v));
```

```
  remplacer (v,"10","144");
  System.out.println (getMaximum (v));

   }
 }
```

NB : attention, la méthode **get** de la classe ArrayList renvoie un Object, donc nécessité de faire un cast (ceci dans la méthode **getMaximum**) pour obtenir un nombre de la classe **Number**.

Exercice 42 (continuité de l'exercice 40)

Dans l'exercice 40, rajouter cette méthode

```
/** rend l'indice du plus petit élément de 'v' compris entre les indices 'imin' et 'imax'
 /* les elements de v sont des Integer  , imin < imax*/

public static int indice_du_plus_petit_entier (ArrayList v,int imin,int imax)
{
// rajouter ce qui manque
}
```

Dans la méthode main, rajouter aussi ces instructions :

```
  ArrayList w = new ArrayList ();
  w.add (new Integer (7));
  w.add (new Integer (17));
  w.add (new Integer (3));
  w.add (new Integer (8));
  w.add (new Integer (0));
  triSelectionEchange (w, w.size ());
  System.out.println (w);
   }
}
```

Exercice 43:

Réaliser une classe **TestStringToken** contenant la méthode
public static boolean isPlusDe4LettresConsecutives (String s) envoyant **true** si la chaîne s contient plus de 4 lettres alphabétiques consécutives, **false** sinon.

Exercice 44 : (héritage te polymorphisme)

Réaliser une classe **Point** disposant des fonctionnalités suivantes :
Un point est caractérisé par ses deux coordonnées cartésiennes (de type entier) **x** et **y** qui seront bien encapsulées.
Prévoir :

- un constructeur qui initialise correctement ces deux champs,
- un constructeur prenant en paramètre un objet Point et appelant le premier constructeur pour l'instanciation d'un Point
- redéfinir la méthode **toString** de la classe ***Object*** pour qu'elle renvoie une chaîne caractérisant l'objet courant
- une méthode **getX** retournant l'abscisse de l'objet courant
- une méthode **getY** retournant l'ordonnée de l'objet courant

- une méthode **setX** prenant en paramètre un entier et permettant de modifier la valeur de l'abscisse du point courant,
- une méthode **setY** prenant en paramètre un entier et permettant de modifier la valeur de l'ordonnée du point courant
- une méthode **deplace** permettant de translater le point courant de **dx** sur l'abscisse et de **dy** sur l'ordonnée,
- surcharger la méthode **deplace** pour qu'elle translate le point courant avec les coordonnées d'un autre point transmis en argument à cette méthode.
- une méthode affiche pour imprimer les caractéristiques d'un point ; cette méthode doit appeler **toString.**

Exercice 45:

Cet exercice utilise les fonctionnalités de la classe **Point** précédente.
Réaliser une classe **Disque** disposant des fonctionnalités suivantes :
Un disque est caractérisé par son ***rayon*** (de type entier) et par son ***centre*** (un objet de type Point).
Prévoir :

- un constructeur prenant en paramètres les deux coordonnées du centre (**cx** et **cy**) et le rayon (r) et permettant d'instancier un objet Disque.
- redéfinir la méthode **toString** pour qu'elle renvoie une chaîne caractérisant l'objet courant (l'objet courant étant caractérisé par les deux coordonnées de son centre et son rayon). Cette méthode appellera son homologue défini dans la classe **Point** qui va se charger de gérer les deux coordonnées du centre du disque.
- une méthode **deplace** permettant de translater le disque de **dx** sur l'abscisse et de **dy** sur l'ordonnée (en fait la translation ne déplace que le centre du disque). Pour ce faire appeler la même méthode définie dans **Point**.
- une méthode **surface** renvoyant l'aire du disque. Utilisez la constante statique **PI** de la classe **Math** du package **java.lang**.
- une méthode affiche pour imprimer les caractéristiques d'un disque ; cette méthode doit appeler **toString.**

Exercice 46 :

Ce programme manipule des **Anneaux**.
Réaliser une classe **Anneau** disposant des fonctionnalités suivantes :
Un anneau est un disque (héritage) avec en plus ***un rayon intérieur*** (qu'on peut appeler **prayon**).
Prévoir :

- un constructeur permettant d'instancier convenablement un objet Anneau. Attention ce dernier doit se baser sur le constructeur de la super classe.
- redéfinir la méthode toString pour qu'elle caractérise parfaitement un objet Anneau, l'objet courant. Cette méthode doit appeler son homologue de la super classe qui se chargera du centre (coordonnées **cx** et **cy**) et du rayon (**r**) ;

- une méthode **deplace** pour translater l'anneau (c à d le centre) de **dx** et **dy**. Il faut se baser sur la méthode **deplace** de la classe mère.
- une méthode affiche pour imprimer les caractéristiques d'un anneau ; cette méthode doit appeler **toString.**

Voici un exemple de classe de test pour ces trois programmes :

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

    Point p = new Point(10,10);    p.affiche ( ) ;
    p.déplace(10,10) ;             p.affiche ( ) ;
    p.déplace (new Point (14,14)) ;p.affiche() ;
    Disque d = new Disque (0, 5, 12);
    d.deplace (40,40);             d.affiche ( ) ;
    System.out.println ("surface du disque" + " " +d.surface ( ));
    d.deplace (100,100) ;          d.affiche  ( ) ;
    d = new Anneau (0, 0, 12, 6); //grace au polymorphisme
    d.affiche ( );
    d = new Anneau (5,5,45,15); //grace au polymorphisme
    d.affich
  }
}
```

```
Point: (10, 10)
Point: (20, 20)
Point: (34, 34)
Disque: (40, 45): 12
surface du disque 452.3893421169302
Disque: (140, 145): 12
Anneau: (0, 0): 12:6
Anneau: (5, 5): 45:15
```

Exercice 47 : (la gestion des exceptions)

Réaliser une classe **FactException** permettant de calculer le factoriel d'entiers naturels avec une bonne gestion des erreurs.
L'entier dont on veut connaître le factoriel sera considéré comme l'unique champ d'un objet quelconque de cette classe.
Prévoir :

- un constructeur à un argument permettant d'initialiser correctement cet attribut,
- une méthode **fact** sans paramètre permettant de calculer le factoriel de l'entier (caractérisant l'objet courant). On considère que lorsque cet entier est strictement négatif, il s'agit d'une exception qui doit être *déclenché* et *traiter simultanément* par cette méthode. Vous créerez donc une sous classe de *Exception* (par exemple **NombreNegatifException**) qui permet de lancer l'exception dont il s'agit et l'exception elle-même sera traitée par un gestionnaire d'exception.
- une méthode **affiche** pour imprimer le résultat du calcul.

Créer une classe de test.

Exercice 48 :

Réaliser une classe **RacineCarreException** permettant de calculer la racine carrée de nombres positifs avec une bonne gestion des erreurs.
Le nombre dont on veut connaître la racine carrée sera considéré comme l'unique champ d'un objet quelconque de cette classe.
Prévoir :

- un constructeur à un argument permettant d'initialiser correctement cet attribut,
- une méthode **racine** sans paramètre permettant de calculer la racine carrée de l'entier (caractérisant l'objet courant). On considère que lorsque ce nombre est strictement négatif, il s'agit d'une exception qui doit être *déclenché* et *traiter simultanément* par cette méthode. Vous utiliserez la classe **AritmeticException** pour lancer l'exception dont il s'agit et l'exception elle-même sera traitée par un gestionnaire d'exception.
- une méthode **affiche** pour imprimer le résultat du calcul.

NB : la méthode statique **sqrt** de la classe **Math** du package **java**.**lang** renvoie sous forme de ***double***, la racine carrée du paramètre de type ***double*** qui lui est transmis.
Créer une classe de test

Exercice 49 :

Réaliser un programme effectuant la division d'un entier par les éléments d'un tableau d'entiers, avec gestion des exceptions dues à une division par zéro, entraînant une **ArithmeticException**.
Prévoir :

- une méthode prenant en paramètres un tableau d'entiers ***tab*** et un entier **n** et renvoyant un autre tableau ***tabres*** dont les éléments sont ceux de la forme **tabres [i] = n/tab[i]**. En utilisant un gestionnaire d'exception adéquat, rendez un message d'erreur correspond au fait qu'un des éléments du tableau transmis en paramètre à cette méthode est **nul** (ce qui provoquera une erreur de division par zéro) et mettez fin au programme.
- une méthode **affiche** permettant d'afficher les éléments d'un tableau quelconque.
- Créer la méthode principale.

Exercice 50 :

Réaliser une classe qui permet de créer un tableau d'entiers à partir d'un entier saisi par l'utilisateur.
On demande un entier **n** et on forme un tableau de taille **n** dont les éléments sont initialisés respectivement par numéro d'indice.
Prévoir :

- une méthode de *classe* **suite** prenant en paramètre un entier et renvoyant un tableau dont chaque élément est initialisé par son propre indice. La taille du tableau étant l'entier transmis à cette méthode, prévoir une classe d'exception qui gère l'éventuel erreur due au fait que l'entier donné est négatif.
- une méthode **imprime** prenant en paramètre un tableau dont on veut afficher les éléments
- une méthode statique **addition** qui prend en argument deux entiers et renvoie la division du premier entier par le second. Prévoir une gestion d'erreur si le diviseur est nul.
- une méthode affiche qui **imprime** le résultat de la fonction **division**.

Exercice 51 : (Synchronisation de threads : blocs synchronisés).

Réaliser une classe **Banque** permettant de créer et de manipuler des comptes en banque.

Un compte sera identifié par le prénom du titulaire, le nom, le numéro de compte, le solde initial et le découvert.
Prévoir :

- un constructeur qui crée convenablement des comptes en banque
- une méthode pour créditer sur le solde. Elle prend en paramètre le montant à créditer
- une méthode pour débiter sur le solde. Elle prend en argument le montant à débiter.
- une méthode pour récupérer le prénom du titulaire du compte
- une méthode pour récupérer le nom du titulaire du compte
- une méthode pour récupérer le solde du titulaire du compte
- une méthode pour récupérer le découvert du titulaire du compte
- une méthode pour modifier le montant du découvert du compte
- une méthode **getHistorique** pour imprimer le solde du compte.

Maintenant, on veut créer deux processus qui gèrent les transactions que le client veut faire sur son compte. Pour simplifier, on suppose que le client peut réaliser deux opérations : créditer ou débiter.
Mais attention ces deux opérations ne devront jamais et en aucun cas se réaliser simultanément, sinon on risque d'avoir un historique sur le compte qui sera faux.
Créer alors deux classes de thread :

- l'une, **ProcessBankCredit**, pour gérer l'opération créditer
- l'autre, **ProcessBankDebit**, pour gérer l'opération débiter.

Expliquer par des commentaires javadoc précis (sur ces deux classes) les dispositions que vous prenez pour que les données (du solde) ne soient pas corrompues si un client lance simultanément ces deux opérations sur un même compte.

Exercice 52 : (course de 1000 m)

Réaliser une classe **Coureur** en étendant la classe **Thread** qui simule une compétition dans une course de 1000 m.
Un coureur n'est identifié que par son nom. Chaque coureur prenant part à cette compétition sera identifié à un processus.
Créer :

- un constructeur qui instancie correctement un coureur
- on redéfinit la méthode d'exécution du thread pour décrire le comportement d'un coureur quelconque : on fixe une pause aléatoire de 1 seconde maximum qui simule le temps mis par chaque coureur pour franchir 100 m. A chaque fois qu'un coureur fait i*100 m, il faut l'afficher comme le montre la sortie de l'exemple.

Voici une classe de test et un exemple de sortie :

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

    // Il s'agit d'une classe de coureurs
    System.out.println ("Passage aux  : ");
    Coureur  j = new Coureur ("Jean");
    Coureur  p = new Coureur ("Paul");
    // On lance les deux coureurs.
    j.start ( );
    p.start ( );
    try { j.join ( ); p.join( ); // attendre la mort de j et p
                                 // avant de commencer k
        }
```

```
Passage aux  :
100 m  par Jean
100 m  par Paul
200 m  par Paul
300 m  par Paul
400 m  par Paul
200 m  par Jean
500 m  par Paul
300 m  par Jean
400 m  par Jean
600 m  par Paul
500 m  par Jean
600 m  par Jean
700 m  par Jean
800 m  par Jean
700 m  par Paul
900 m  par Jean
1000 m  par Jean
800 m  par Paul
```

```
        catch (InterruptedException e) {};
        Coureur k = new Coureur ("toto");
        k.start ( ) ;
    }
}
```

Exercice 53 : reprendre *l'exercice 1* en implantant cette fois-ci l'interface **Runnable**.

Exercice 54 :

Ecrire une classe **Trieur** contenant les méthodes suivantes :

- **public int [ ] triTableau (int t [ ])**, qui prend en argument un tableau d'entiers et renvoie un tableau dont les éléments sont triés par ordre croissant.
- **public void echanger (int t [ ], int i, int j)**, cette méthode prend en paramètre un tableau d'entiers et deux indices i et j qui sont les deux indices du tableau dont on veut échanger le contenu. Cette méthode sera utilisée dans la méthode triTableau pour échanger les valeurs de deux cases du tableau à trier.
- **public void affiche ( )**, qui affiche les éléments du tableau trié.

Créer maintenant deux classes de threads : **ThreadTrieTableau** qui permet de lancer l'opération de trie d'un tableau d'entiers et **ThreadAfficheTableau** qui lance l'affichage des éléments du tableau trié.
Gérer ces deux processus de telle sorte que des ***problèmes d'accès concurrents*** (que vous identifierez) sur une donnée partagée, ne puissent jamais se rencontrer.
Créer une classe **TestTrieur** où vous créerez un tableau en demandant de saisir ses éléments au clavier et vous lancerez les deux threads sur ce dernier.
NB : vous créerez deux versions de ces deux threads : une première où les deux classes de threads dérivent de la classe **Thread** et une deuxième version où elles implantent l'interface **Runnable**.

TP de programmation sur les Entrées-sorties en JAVA

<u>Exercice sur les Entrées-sorties :</u>

Dans cet exercice, on veut créer et manipuler des comptes en banque tout en ayant la possibilité de sauvegarder toutes les transactions dans des fichiers situés sur disque.
Toutes les dispositions doivent être prises pour gérer les éventuelles exceptions que peut suscitée l'utilisation de ce mini logiciel.
L'exercice se déroule en plusieurs étapes :
<u>Etape 1 :</u> (*création de l'interface graphique*)

Réaliser d'abord l'interface graphique ci-dessus qui sera votre interface utilisateur.
Pour cela, votre répertoire de travail sera **C:\gestioncompte** ou **D:\gestioncompte**.
Créer l'interface dans un package nommé ***frame.compte***.

Etape 2 : *(création des comptes en banque)*

Ici, on met en œuvre toutes les opérations de création et de manipuler de comptes.
Pour cela, créer une classe **Compte,** qui permet de créer convenablement un compte.
Ceci se fera dans un package nommé ***gestion.transaction***
Cette classe doit disposer au minimum des fonctionnalités suivantes :

- un constructeur pour initialiser convenablement le compte
- des méthodes **getNom ( ), getPrenom ( ), getSolde ( ), getDecouvert ( ), getNumeroCompte ( ), getHistorique ( )**
- des méthodes **crediter (**double montant**), debiter (**double montant**)**
- une méthode **setDecouvert (montant nouveauDecouvert)**

Etape 3 : (*gestion des transactions sur le compte par des processus synchronisés)*

Pour la gestion des transactions sur le compte, on considère que celui-ci est une ressource critique qui ne devrait jamais être accédée simultanément dans des opérations de dépôt et de retrait. De ce fait les opérations de retrait et de dépôt doivent être gérées séparément par deux processus synchronisés dans le but de gérer l'exclusion mutuelle sur le partage de cette donnée.
On créera donc deux threads **ThreadCompteRetrait** et **ThreadCompteDepot** qui s'occuperont respectivement de l'opération retrait et de celle du dépôt.
Ces deux classes seront mises dans un paquet nommé **gestion.synchro**.

Etape 4 :(G*estion des évènements liés aux composants graphiques et Entrées sorties*)

Il faut revenir dans la classe de l'interface graphique pour réaliser ces opérations.

Le bouton **Creer compte** permet de créer un nouveau compte pour un client bien donné et de l'enregistrer sur disque c à d créer un fichier texte client contenant l'ensemble des informations saisies pour ce client. Pour l'enregistrement, on utilisera pour chaque client un fichier situé dans **C:\gestioncompte\clients**.
Un certain nombre de contraintes sont liées à la création d'un compte :

- aucun champ ne doit être vide, sinon une boîte de dialogue modale le signale
- les champs **prenom** et **nom** ne doivent contenir que des lettres alphabétiques, le fait que l'utilisateur tape des chiffres ou des caractères de ponctuation (par exemple) doit être ignoré. Autrement dit si ces deux zones de texte détiennent le curseur, seul le pavé alphabétique sera actif.
- le **solde** et le **decouvert** doivent contenir exclusivement des chiffres, les lettres doivent être ignorées.
- Le numéro de compte (**NumeroCompte**) doit être un nombre de taille **10** chiffres

Si la création se passe bien, une boîte de dialogue modale doit l'indiquer. S'il y a erreur de création, une boîte modale aussi doit le signaler à l'utilisateur.

Le rôle du bouton **Historique du compte** est d'affiche toutes les informations concernant un client dans la zone de texte multi ligne. Pour ce faire, il suffit de saisir le numéro du compte et de cliquer sur ce bouton.

Les boutons **Depot** et **Retrait** s'occupe respectivement des opérations de dépot et de retrait sur le solde du compte. Attention ces deux opérations sont synchronisées.

Ces deux opérations se doivent se faire selon le numéro du compte saisi.
Il ne faut pas autoriser que l'utilisateur saisisse des montants erronés pour ces deux transactions (pas de mélange de chiffres et de lettres).
On sait que ces deux transactions modifient le solde du client. Par conséquent, les opérations doivent entraîner la ***recréation*** du fichier client pour tenir en compte les dites modifications.

Ajout de fonctionnalités:

Il faut bien gérer la fin de l'application par un événement convenable.
Le fait de cliquer sur un **bouton** doit être équivalent à *y placer le focus et de presser la touche entrée.*
Pour créditer ou débiter, il doit être aussi possible de saisir le montant dans la zone prévue et de presser **Entree**, autrement dit la zone de texte doit être sensible au pressing sur la touche **Entrée**.
IMPORTANT : (Pour bien réaliser l'étape 4)

Pour utiliser les fonctionnalités définies aux étapes 2 et 3 dans l'étape 1, il faut créer *deux fichiers jar* pour les deux packages ***gestion.synchro*** et ***gestion.transaction*** et créer des librairies dans ***frame.compte*** (qui encapsulent ces jars) afin d'importer les classes que ces deux paquetages contiennent.

Exercice 1* :**

On veut gérer les notes des étudiants de master professionnel 1 par un logiciel qui sauvegarde toutes les notes de chaque étudiant dans un fichier sur disque.
Donc pour chaque étudiant, il faut créer un fichier texte dont le nom est le numéro de carte étudiant et ce fichier sera sauvegardé dans le répertoire **D:\masterpro1\allnotes**.
Pour cela dans une classe ***GestionNotes*** créer les méthodes suivantes :
*/*cette méthode prend en argument un tableau de notes concernant un étudiant X et l'enregistre dans le fichier qui concerne l'étudiant dont le numéro de carte est numCarte*/*
public File saveStudentsMarks (String numCarte, double notes [ ])
*/*cette méthode permet de saisir (grâce à un JOptionPane) l'ensemble des notes de l'étudiant et les stocke dans un tableau*/*
public double notes [ ] printMarks ( )
*/*cette méthode permettra d'afficher l'ensemble des notes d'un étudiant sur une zone de texte JTextArea (cette zone de texte sera créée dans la classe qui suit)*/*
public void showMarks ( )
Créer maintenant une classe ***swing*** **InterfaceGestion** où vous allez saisir les notes des étudiants pour ensuite l'enregistrer dans les fichiers correspondants.
Et à partir des fichiers lire (sur un ***JTextArea***) grâce à la méthode showMarks les notes d'un étudiant donné.

Exercice 2* :**

Créer une classe **Lecture** qui permet d'ouvrir le poste de travail de Windows et de lire le contenu de n'importe quel fichier texte dans une zone de texte multiligne.

Exercice 3* :**

Créer une classe **Ecriture** qui permet d'enregistrer dans un fichier texte existant dans n'importe quel répertoire du poste de travail de Windows, ou que vous créez vous-même (c'est-à-dire un nouveau fichier) ce que l'utilisateur saisi au niveau d'un ***JTextArea***.
L'utilisateur valide sur Entrée ou sur un bouton **Enreg** pour effectuer la sauvegarde.

Travaux Pratiques sur les collections

Dans ce TP, vous manipulez principalement les collections et les swing.

Exercice 1C :

Créer une classe **Compte** qui manipule des comptes en banque.
Pour simplifier un compte est défini par le *numero*, le *nom* du titulaire, le *prénom* du titulaire le *solde* initial et le *decouvert* autorisé.
Implémentez convenablement cette classe avec l'ensemble des méthodes de services nécessaires comme :

*/*cette méthode retire montant du solde et retourne le solde après l'opération*/*

public double retirer (double montant)

*/*cette méthode dépose montant dans le solde et retourne le solde après l'opération*/*

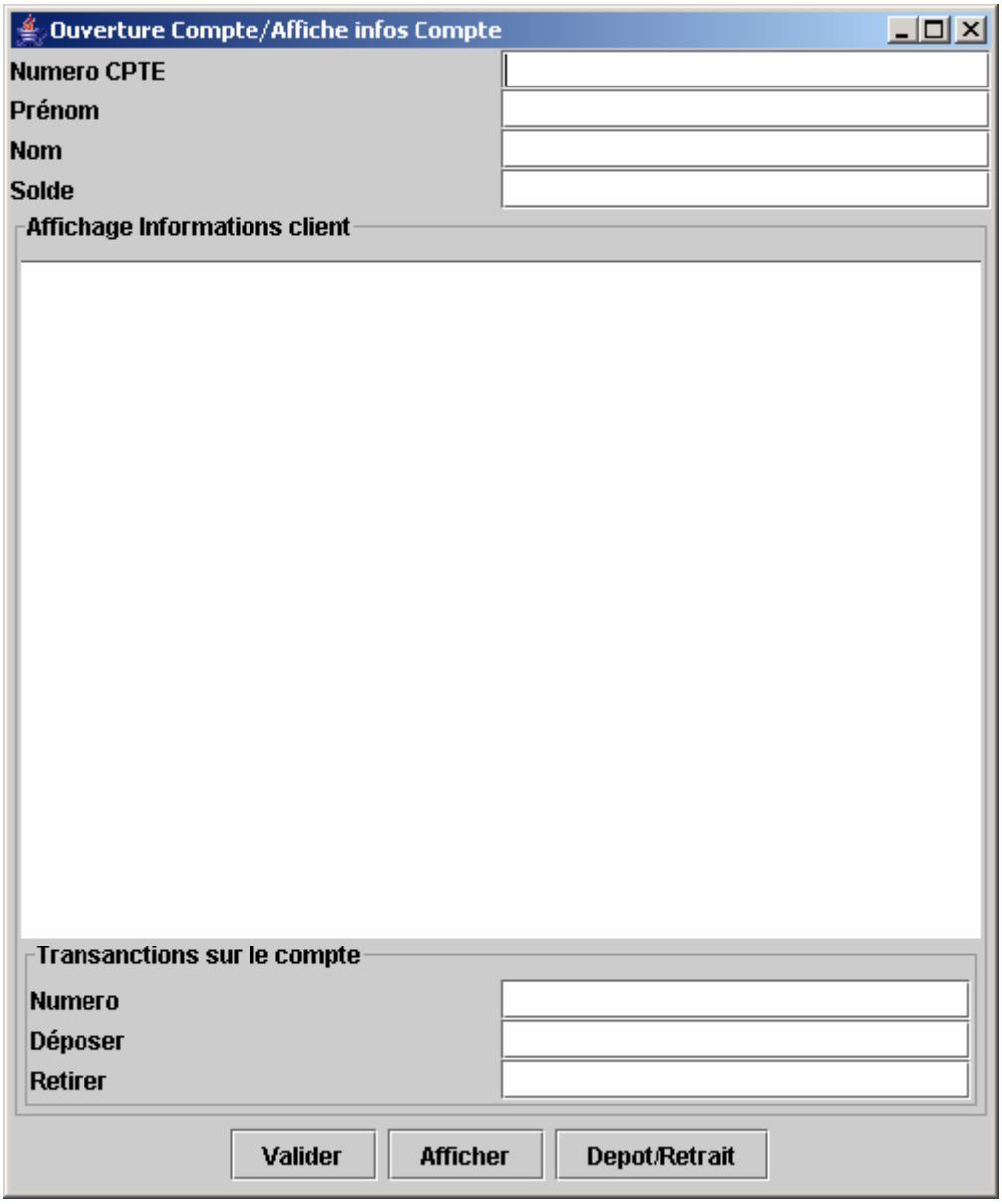
public double deposer (double montant)

Important : dans cette classe, vous disposerez d'une variable de classe de type ***Hashtable*** qui vous permettra, à chaque fois qu'un compte est créé de le stocker en mémoire.
C'est à partir de cet objet que vous retrouverez un compte à partir du numéro pour pouvoir y effectuer des transactions comme débiter ou créditer.
Ces deux opérations spéciales doivent être synchronisées pour éviter les accès concurrents.
Maintenant créez une classe graphique **BankClients** qui gère les comptes clients.
Cette disposera d'un constructeur

public BankClients (String num, String prenom,String nom, double solde, double decouvert)

Vous vous servirez du contructeur de Compte pour definer celui-ci.
Elle disposera aussi des méthodes suivantes qui seront définies grâces aux méthodes déposer et retirer de la classe **Compte**.
public double deposer (String num, double montant) et
public double retirer (String num, double montant)
L'interface de cette classe ressemblera à ceci :

NB : le bouton ***valider*** sert à la création d'un compte une fois les informations saisies.
Le bouton ***afficher*** (on saisie le num du compte) et affiche le infos sur ce compte dans la zone de texte multiligne.
Le bouton ***Depot***/***retrait*** : on saisie le numero du compte et le monatnt du debit ou bien du crédit.



Exercice 2C :

Ici vous allez grâce à JDBC sauvegarder l'ensemble des comptes clients dans une base de données **MySQL**.

A partir de la base de données charger l'ensemble des comptes dans un ***JTable*** que vous ajouterez à votre interface graphique.

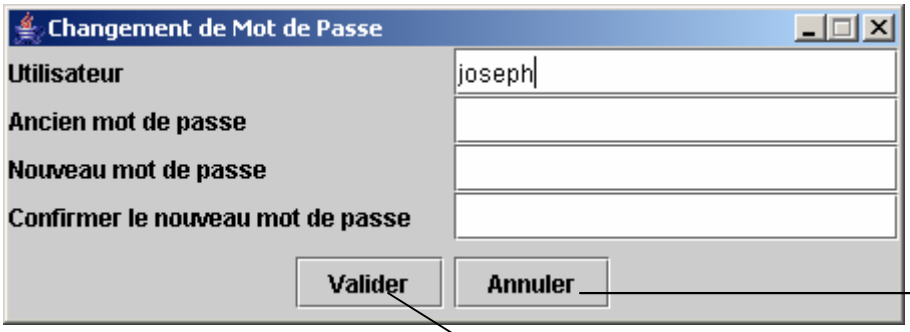
Travaux Pratiques sur JDBC

Objectifs : Dans ce **TP**, il s'agit de créer une interface graphique (à l'aide des Swing Java) à partir de laquelle on se connecte à une base de données (*Access ou MySQL*) pour ensuite lancer des requêtes de sélection et de mises à jour.

Premièrement, il faut s'authentifier pour avoir accès à l'application. Réaliser, pour cela, cette interface d'authentification :

Contraintes pour s'authentifier :

- le bouton ***Entree*** est inactif par défaut, il ne devient actif que lorsque l'utilisateur **commence** à *saisir son mot de passe*. Il permet d'entrée dans l'application si le *login et le mot de passe sont corrects* (la vérification de l'exactitude se fera depuis une table de votre Base de Données). Autrement, une boîte de message indique que vous ne pouvez pas vous connecter à cette application.
- Le bouton ***Annuler*** sert à annuler la frappe (effacer le login et le mot de passe saisis).
- La case à cocher **Changer mot de passe** sert à changer le mot de passe de l'utilisateur en cours. Lorsque vous cliquez dessus, elle renvoie à ceci :



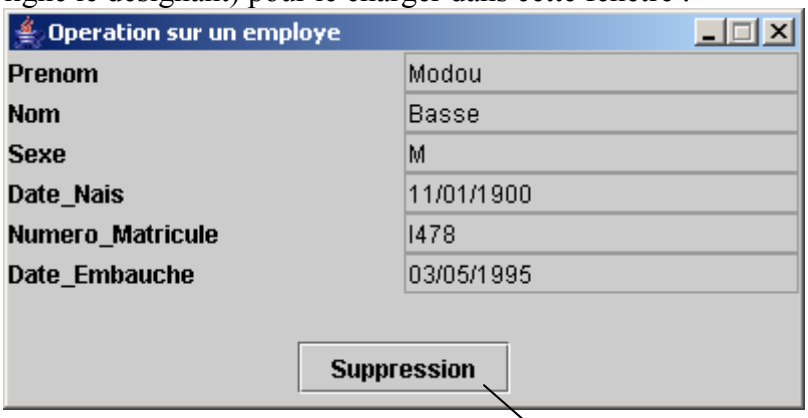
Pour annuler et revenir à l'écran précédent

Pour valider avec une boîte de dialogue

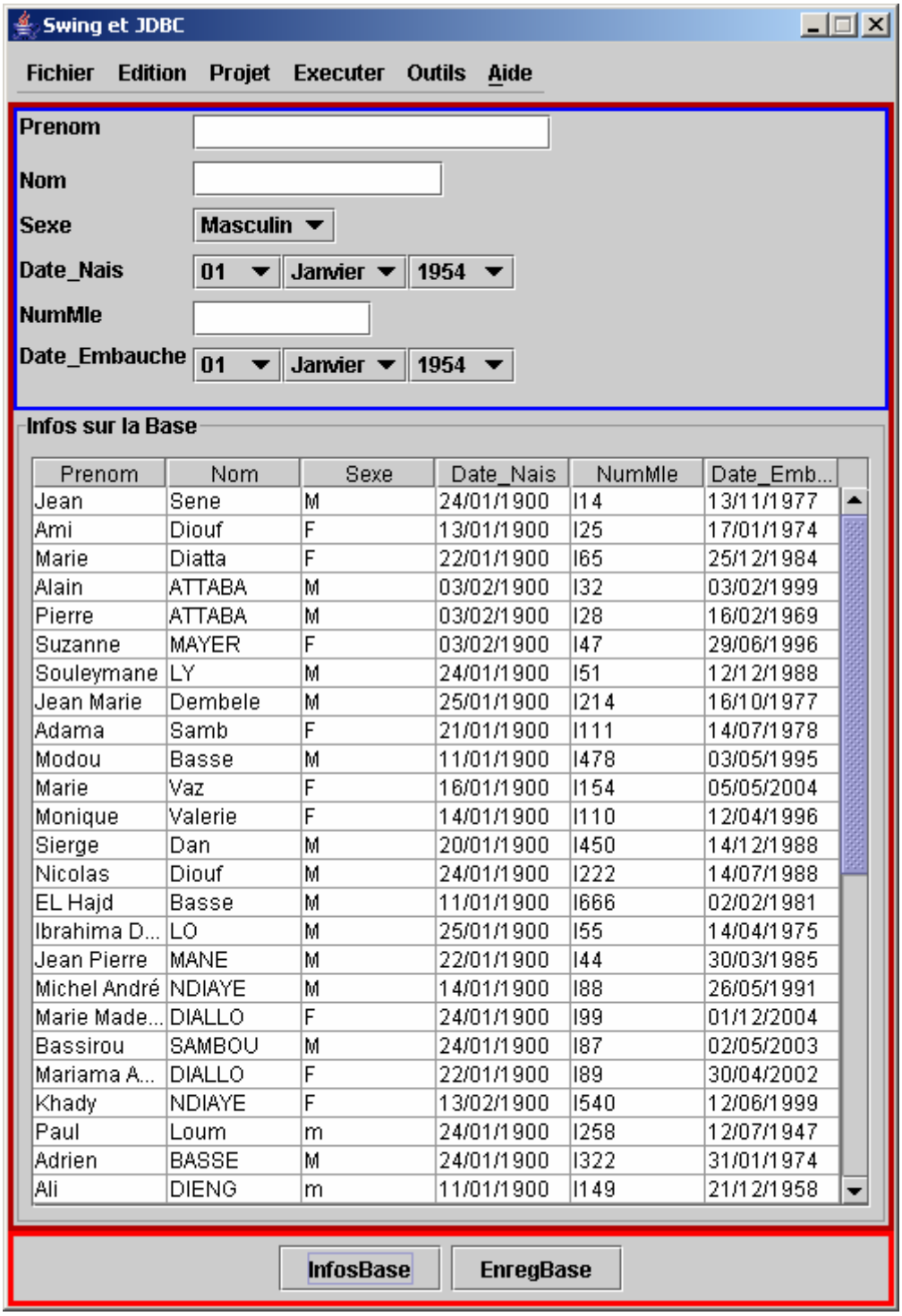
- si l'authentification réussie, on entre dans l'application, décrite par cette interface :

C'est une application simple, consistant à créer une grille (un **JTable**) qu'il faudra remplir (en cliquant sur le bouton **InfosBase**) à partir des enregistrements de la table **Employes** de votre base de données.
Si un employé est embauché, il faut l'enregistrer au niveau de la base. Donc saisir les informations le concernant et cliquer sur le bouton **EnregBase**.
Pour l'enregistrement, aucun champ vide ne sera autorisé.

Pour supprimer un employer, sélectionner le à partir de la grille (il s'agit de cliquer sur la ligne le désignant) pour le charger dans cette fenêtre :



Pour effectuer la suppression ou l'annuler



Prenom	Nom	Sexe	Date_Nais	NumMle	Date_Emb...
Jean	Sene	M	24/01/1900	I14	13/11/1977
Ami	Diouf	F	13/01/1900	I25	17/01/1974
Marie	Diatta	F	22/01/1900	I65	25/12/1984
Alain	ATTABA	M	03/02/1900	I32	03/02/1999
Pierre	ATTABA	M	03/02/1900	I28	16/02/1969
Suzanne	MAYER	F	03/02/1900	I47	29/06/1996
Souleymane	LY	M	24/01/1900	I51	12/12/1988
Jean Marie	Dembele	M	25/01/1900	I214	16/10/1977
Adama	Samb	F	21/01/1900	I111	14/07/1978
Modou	Basse	M	11/01/1900	I478	03/05/1995
Marie	Vaz	F	16/01/1900	I154	05/05/2004
Monique	Valerie	F	14/01/1900	I110	12/04/1996
Sierge	Dan	M	20/01/1900	I450	14/12/1988
Nicolas	Diouf	M	24/01/1900	I222	14/07/1988
EL Hajd	Basse	M	11/01/1900	I666	02/02/1981
Ibrahima D...	LO	M	25/01/1900	I55	14/04/1975
Jean Pierre	MANE	M	22/01/1900	I44	30/03/1985
Michel André	NDIAYE	M	14/01/1900	I88	26/05/1991
Marie Made...	DIALLO	F	24/01/1900	I99	01/12/2004
Bassirou	SAMBOU	M	24/01/1900	I87	02/05/2003
Mariama A...	DIALLO	F	22/01/1900	I89	30/04/2002
Khady	NDIAYE	F	13/02/1900	I540	12/06/1999
Paul	Loum	m	24/01/1900	I258	12/07/1947
Adrien	BASSE	M	24/01/1900	I322	31/01/1974
Ali	DIENG	m	11/01/1900	I149	21/12/1958

Fenêtre de l'application.

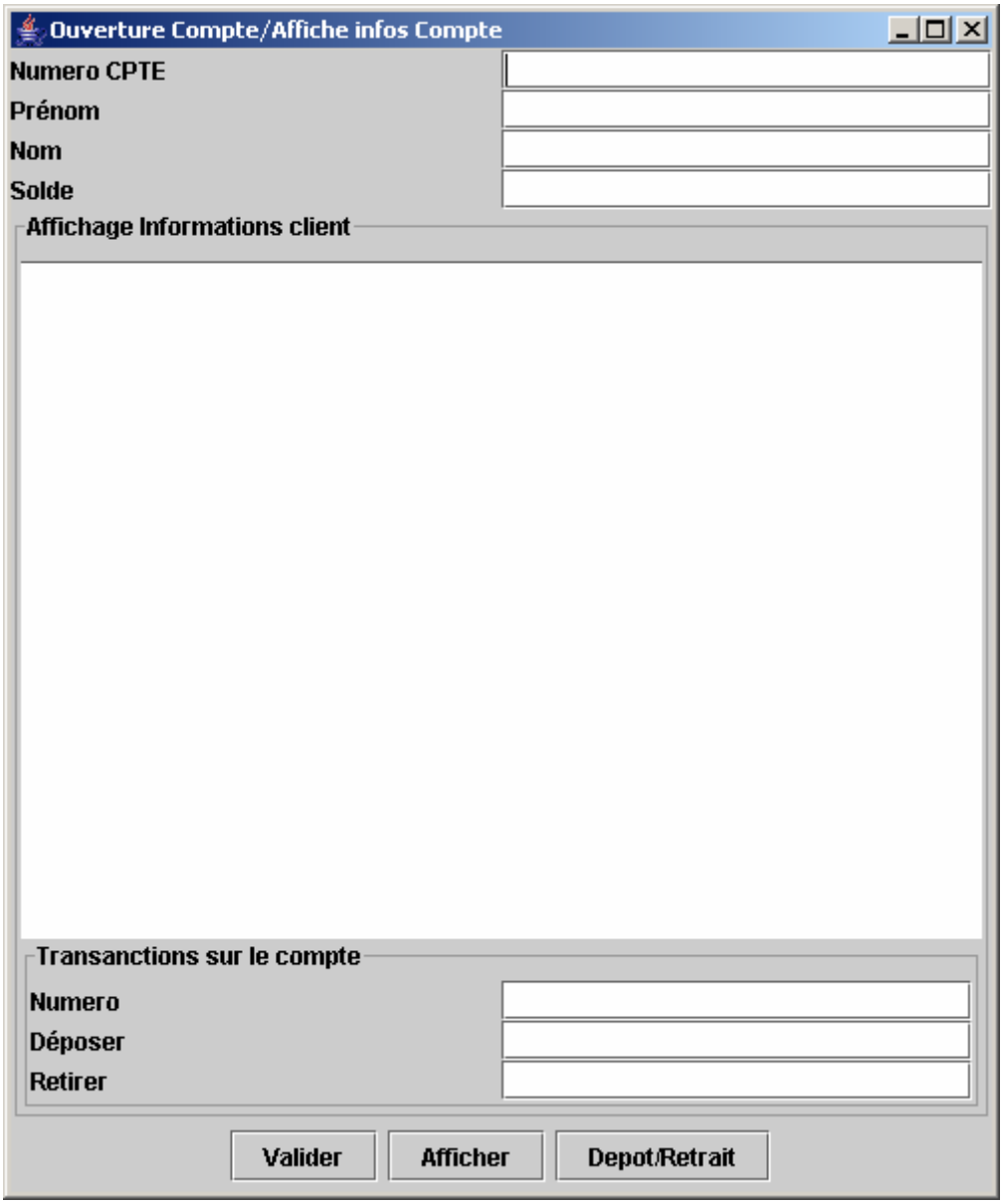
Vous créerez la base, vu la structure de votre JTable. En fait vous n'avez besoin que la table **Employes (Prenom, Nom, Sexe, Date_Nais, NumMle, Date_Embauche)** et d'une table contenant les employés (**secrétaire** et **DRH**) qui ont le droit d'entrer dans l'application.

Exercice complémentaire sur JDBC

Objectif : créer une interface graphique swing, se connecter à une base de données ***MySQL*** via **JDBC**, enregistrer dans la base, lire les informations de la base et les afficher dans l'interface.

Pour simplifier vous créerez une base de données avec MySQL constituée d'une table ***clients (Numero, prenom, nom, solde).***

Voici l'interface qu'il faudra réaliser (c'est la même que pour les Collections) :



Réalisation de l'application:

Il faut d'abord créer une classe ***Compte*** qui permet de créer des comptes en banque et d'effectuer les transactions déposer et retirer et l'affichage des détails du compte.
Cette classe devra disposer au minimum :

- d'un constructeur
- d'une méthode deposer

- d'une méthode retirer
- de méthodes d'accés (getter) pour chaque champ
- d'une méthode d'affichage.

Pour ouvrir un nouveau compte :

Au niveau de l'interface lorsque l'utilisateur saisit les informations sur le client, il doit valider par le bouton **Valider** pour enregistrer les informations dans la table clients de la base.

Pour effectuer une transaction déposer ou retirer :

Pour retirer ou déposer de l'argent, il suffit de saisir le *numero* et le montant du *dépôt* ou du *retrait* et de valider sur le bouton **Depot/Retrait** et l'opération est effectuée dans la base.

Pour l'affichage des détails du compte d'un client :

Pour afficher les détails du compte d'un certain client, il faut saisir son numéro et cliquez sur le bouton Afficher et les données sont affichées dans la zone **« Affichage Informations client »**

FIN