Ecole Nationale des Sciences Appliquées - Al Hoceima
Département Mathématiques et Informatique

Travaux pratiques N°2 : Classes, Tableaux et Chaînes de caractères

Module : Programmation Orientée Objet en Java
Première Année Ingénierie des données

Année universitaire 2022/2023

Exercice 1

Ecrire une classe *ComplexArithmetic* qui permet d'effectuer les opérations habituelles sur les nombres complexes

- Somme de deux nombres complexes
- Produit de deux nombres complexes
- Le conjugué d'un nombre complexe
- Module d'un nombre complexe

Exercice 2

1. **Ecrire une classe implémentant les méthodes suivantes :**

   - ✓ Une méthode *readMatrix* qui permet de lire une matrice d'entiers au clavier
   - ✓ Une méthode *readArray* qui permet de lire un tableau d'entiers au clavier
   - ✓ Une méthode qui retourne les nombre répétés dans un tableau d'entier.
   - ✓ Une méthode qui vérifie si un nombre existe ou non dans un tableau d'entiers.
   - ✓ Une méthode qui retourne le plus grand et le plus petit nombre dans un tableau d'entiers.
   - ✓ Une méthode qui retourne le k-ième plus petit élément dans un tableau d'entiers non trié ?
   - ✓ Une méthode qui retrouve les éléments communs dans trois tableaux triés.
   - ✓ Une méthode qui permet de réorganiser un tableau en alternant nombre positif et négatif
   - ✓ Une méthode qui permet de vérifier si un tableau contient un ensemble de valeur dont la somme est égale à zéro
   - ✓ Une méthode qui permet de trouver la longueur de la plus longue séquence consécutive dans un tableau d'entiers.
   - ✓ Une méthode qui permet d'inverser un tableau.
   - ✓ Une méthode qui permet de pivoter une matrice carrée de 90 degrés dans le sens des aiguilles d'une montre.

- ✓ Une méthode qui permet d'afficher la matrice dans l'ordre Spirale à l'aide de la récursivité.
- ✓ Une méthode qui retourne les points cols d'une matrice (Saddle point). (L'élément est dit point col de la matrice s'il est à la fois un minimum de sa ligne et un maximum de sa colonne ou vice versa.)

2. **Ecrire un programme de test**

Exercice 3

1. Ecrire une méthode *generatePassword* qui permet de générer et renvoyer un mot de passe de N caractères générés aléatoirement en utilisant les deux algorithmes simplifiés données ci-dessous.
2. Analyser les défauts de ces algorithmes et proposer une autre implémentation meilleure.

Algorithme 1 :

Tant que le nombre de caractère généré est inférieur à N faire :
Générer un entier qui représente le code d'un caractère Unicode aléatoirement.
Stocker ce caractère dans un tableau
Fin Tant que
Convertir le tableau de caractères en une chaîne de caractères. grâce au constructeur String (char[] value)) de la calasse String.

Algorithme 2 :

Motdepass = *chaîne vide*
Tant que le nombre de caractère généré est inférieur à N faire :
Choisir un caractère au hasard avec la méthode *random* de la classe *Math* parmi un ensemble de caractères autorisés représenté par un tableau
Ajouter le caractère à *Motdepass*
Fin Tant que
retourner *Motdepass*

Exercice 4

Un nombre complexe z se présente en général en coordonnées cartésiennes, comme une somme a + bi, où a et b sont des nombres réels quelconques et i (l'unité imaginaire) est un nombre particulier tel que $i^2 = -1$.

Deux nombres complexes sont égaux si et seulement s'ils ont la même partie réelle et la même partie imaginaire.

Le conjugué d'un nombre complexe ***a+bi*** est ***a-bi***. L'image d'un nombre complexe z = a + i b est le point M de coordonnées (a, b).

Un nombre réel est un cas particulier d'un nombre complexe ayant b = 0.

1. Ecrire une classe ***Complexe*** qui dispose des méthodes suivantes :

- Un constructeur pour initialiser la partie réelle et imaginaire.
- Une méthode : *Complexe conjugue()* qui retourne le conjugué de l'objet appelant cette méthode.
- Une redéfinition de la méthode *equals* permettant de comparer deux nombres complexes.
- Une méthode *toString* qui retourne la présentation sous forme d'une chaîne de caractère d'un nombre complexe : a+bi :
  *Si $a \neq 0$ et $b \neq 0$ on retourne la chaîne de caractère a+b*i avec a et b les parties réelle et imaginaire.*
  Si b = 0 on retourne une chaîne de caractère représentant le nombre réel correspondant
  *Si a = 0 on retourne une chaîne de caractère représentant le nombre sous format b*i*
- Une méthode qui permet de calculer l'argument d'un nombre complexe
- Une méthode qui permet de calculer la distance entre les images de deux nombres complexes.

2. Ecrire une classe ***Equation*** qui permet de représenter les équations de type $\boldsymbol{ax^2+bx+c=0}$. Cette classe possède :
   - Trois attributs a, b et c de type réel.
   - Un constructeur permettant d'initialiser les coefficients a, b et c de l'équation.
   - Une méthode ***afficheEquation*** permettant d'afficher l'équation sur la console.
   - Une méthode ***resoudreEquation*** qui permet de résoudre l'équation dans l'ensemble *C* et de retourner le résultat sous forme d'un tableau d'objets de type Complexe.

3. Ecrire un programme qui lit les coefficients d'une équation de second dégrée à la console puis elle affiche les solution de cette équation dans l'ensemble C en utilisant la méthode *resoudreEquation* de la classe ***Equation***. L'affiche de la solution doit avoir la forme suivante :

```
Valeur de Delta : la valeur
Nombre de solutions : ….
Type des solutions : Complexes ou réels
Les solutions : …..
```

Exercice 5

La méthode statique *compareTab* de la classe **TabUtils** prend comme paramètres deux tableaux d'entiers *pTab1* et *pTab2*. Si les deux tableaux passés en paramètres sont de même taille, la méthode doit renvoyer un tableau d'entiers de même longueur et qui contient les nombres 1 et 0. Le $i^{ème}$ élément du tableau renvoyé contient 1 *si pTab1[i] ≠ pTab2[i]* et 0 si *pTab1[i] = pTab2[i]*.

Si les deux tableaux passés en paramètres ne sont pas de même taille, la méthode *compareTab* retourne -1.

Par exemple, Si *pTab1 = {1, 2, 3}* et *pTab2 = {3, 2, 1};* l'appel TabUtils.compareTab(pTab1, pTab2) renvoie le tableau {1,0,1}.

Ecrire la classe ***TabUtils*** et sa méthode *compareTab*.

Exercice 6

En utilisant la classe *ArrayList* écrire une classe *MessageManger* qui permet de gérer les messages d'un forum de discussion. Un message est caractérisé par un id, un titre, une date et un contenue.

Dans la classe *MessageManger* Implémenter les méthodes suivantes :

- *getAllMessages : Retourne la liste des messages*
- *deleteMessage : Supprime un message de la liste en connaissant son id.*
- *UpdateMessage : Permet de mettre à jour un message existant dans la liste.*
- *findMessageById: permet de retrouver un message par id*
- *findMessageByTitle: permet de retrouver un message par titre*
- *getNumberOfMessages : retourne le nombre de message de la liste.*
- *getFirstMessage: retourne le premier message dans la liste*
- *getLastMessage: retourne le dernier message dans la liste.*
- *dddMessage : Ajoute un message à la liste.*

Refaire l'exercice en utilisant la classe LinkedList

Exercice 7 :

1. Ecrire une classe ***Produit*** définie par un nom, un prix et le nombre de jours restant avant péremption du produit. Cette classe comportera les méthodes suivantes :
    - Un constructeur initialisant les attributs de la classe.
    - Des méthodes pour accéder ou modifier les valeurs des attributs privés
    - Une méthode ***toString*** renvoyant une représentation des objets sous forme d'une chaîne de caractères.
2. Ecrire une classe ***PileProduit*** implantant une pile de produits. Doter cette classe des méthodes suivantes :
    - Un ***constructeur*** construira la pile vide.
    - Une méthode ***estVide*** permettant de tester si une pile est vide.
    - Une méthode ***empile*** (ajoute un produit au sommet de la pile)
    - Une méthode ***depile*** (retourne le sommet et le retire de la pile)
    - Une méthode ***getSommet*** (retourne le sommet de la pile sans le retirer)
    - Une méthode ***affiche*** qui affiche le contenu de la pile, sans la modifier.
    - Une méthode ***tri*** permettant de trier la pile des produits suivant la date de péremption en plaçant en haut de la pile le produit dont le nombre de jour avant péremption est le plus faible
3. Soit la classe ***Entrepot*** définie par un ensemble de produits. Cet ensemble est vu comme une pile de produits. Cette classe comportera les méthodes suivantes :
    - Un ***constructeur*** construisant un entrepôt avec une pile de produits vide.
    - Une méthode ***suppressionPerime*** qui supprime de l'ensemble tous les produits périmés et renvoie la somme perdue.

Exercice 8 :

On veut écrire une classe de gestion d'une pile générique offrant les opérations classiques suivantes : Initialiser, Afficher, Empiler, Dépiler et Donner le sommet de la pile.

1. Donner une définition de la classe ***Pile*** qui définit une pile générique (pile que l'on représentera à l'aide d'une liste d'objet de type *ArrayList<Object>*). Cette classe comportera les méthodes suivantes :
   a- Un constructeur sans argument construit une pile vide.
   b- Une méthode ***estVide*** permettant de tester si la pile est vide.
   c- Une méthode ***empiler*** ajoute un élément au sommet de la pile.
   d- Une méthode ***depiler*** retourne le sommet et le retire de la pile
   e- Une méthode ***getSommet*** retourne le sommet de la pile sans le retirer

2. On veut définir une méthode qui vérifie si une chaîne (ou expression) est bien " parenthésée " : Une parenthèse fermante est en correspondance logique avec une parenthèse ouverte.

   Exemple :

   - (((a+51) - (c))) Expression correcte
   - ((a+b) Expression incorrecte, ainsi que (a+b))

Ecrire une classe ***ArithmeticGrammarChecker*** disposant d'une méthode statique ***static boolean checkParenthesis(String expression)*** qui permet de vérifier la validité d'une expression passée en paramètre.

Indications :

Utiliser la pile définie dans la question précédente, en suivant la méthode suivante

- Parcourir l'expression et à la rencontre d'un caractère '(' on l'empile sur la pile p et à la rencontre d'un caractère ')' on dépile la pile p
- On ignore tout autre caractère.
- A la fin de l'expression, si la pile est vide l'expression est bien "parenthésée". Sinon, il y a plus de parenthèses ouvrantes que de parenthèses fermantes. Si la pile est vide prématurément, lors d'un dépilement, alors il y a plus de parenthèses fermantes que de parenthèses ouvrantes.

Exercice 9 :

En géométrie euclidienne, un polygone est une figure géométrique plane, formée d'une suite cyclique de segments consécutifs et délimitant une portion du plan. Le polygone le plus élémentaire est le triangle : un polygone possède au moins trois sommets et trois côtés.

L'API Java Standard offre une classe ***Polygon*** dans le package *java.awt*, avec cette classe un polygone est représenté par deux tableaux de nombres, un pour les abscisses et un autre pour les ordonnées, ce qui rend son utilisation inconfortable.

L'objectif de cet exercice est l'écriture d'une nouvelle classe ***Polygon*** où un polygone sera représenté par un tableau d'objets de type Point.

1. Écrire une classe **Point** permettant de décrire les coordonnées d'un point dans le plan. Cette classe aura :
   - Des attributs x et y pou représenter l'abscisse et l'ordonné.
   - Un constructeur public ***Point(double x, double y)*** qui permet d'initialiser le point lors de sa création.
   - Méthode ***equals*** pour comparer deux objets de type point.
   - Méthode ***deplacer***(double dx, double dy) qui permet de déplacer un point dans l'espace. (Exemple *M(x,y)* deviendra *M(x+dx, y+dy)* après le déplacement)
   - Méthode ***distance(Point p)*** permet de calculer la distance entre deux points.
   - Des méthodes public ***double getX()*** et public ***double getY()*** qui retournent respectivement les valeurs de x et de y.
2. Ecrire une classe ***Polygone*** représentant un polygone à n sommet dans le plan. Un polygone sera représenté par un tableau de n points. Cette classe comportera les méthodes suivantes :
   - Une méthode ***randomPolygone(n)*** prenant un entier n et retournant un nouveau polygone de n sommets et dont les coordonnées des sommets seront choisies au hasard dans [0,200[. (On peut utiliser la méthode ***nextDouble*** de la classe ***Random*** ou la méthode ***Math.random()***).
   - Une méthode ***perimetre()*** retournant le périmètre du polygone.
   - ***translater(double dx, double dy)*** permet de translater le polygone avec le vecteur de translation $\vec{V}$***(dx,dy)***
3. Dans la classe ***Polygone*** ajouter une méthode ***getRectangleBoite()*** qui retourne le plus petit rectangle ayant des cotés parallèles aux axes du repère et contenant le polygone.

Exercice 11 :

Le but de cette partie de l'exercice est l'écriture d'une liste simplement chaînée générique capable de stocker des objets de n'importe quel type. Avant la version *JDK 5.0* de Java, la seule façon de mise en œuvre de la généricité consistait à utiliser la classe *Object*. Comme toute classe dérive de cette classe, et donc hérite de son type, il est possible de stocker dans un objet de type ***Object*** n'importe quel autre type d'objets. C'est cette façon de procéder que vous allez utiliser dans cet exercice.

Première partie

1. Implémenter une classe ***Node*** qui permet de représenter une cellule d'une liste chaînée. Ci-dessous le squelette de la classe Java à implanter :

```
public class Node {
   /** la valeur stockée */
   private Object value;
```

```
    /** la référence à l'élément suivant de la liste */
    private Node next;
….
}
```

2. Implémenter la classe ***SingleLinkedList*** qui permet d'implémenter la liste simplement chaînée et fournit les services nécessaires pour la manipulation de cette liste (l'ajout, la suppression des données, la taille, le parcours de la liste …)
   Ci-dessous le squelette de la classe Java à implanter :

```
public class SingleLinkedList {

    private Node begin = null;
    private Node end = null;
    private Node current = null;

     //constructeurs
    public SingleLinkedList () {};

    public SingleLinkedList (Object pVal) {
        current = new Node(pVal);
        end = current;
        begin = current;
    }

    /** ajouter un élément à la liste */
    public void addElement(Object pValue) {
        …
    }

    // Retourner la valeur de l'élément courant de la liste
    public Object getCurrentElement() {
    ….

    }

    /** retourner le premier élément de la liste */
    public Node getFirstElement() {
….

    }

    /** retourner l'élément suivant dans la liste */
    public Node next() {
     ….

    }

    /** retourner la taille de la liste */
    public int size() {
     ….
```

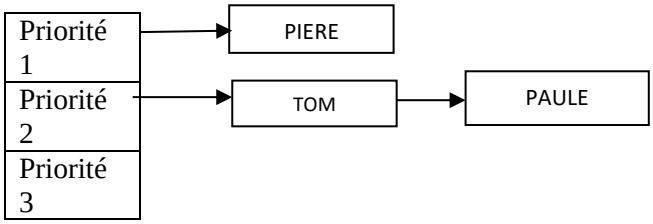
```
    }


  /** Supression d'un élément d'indice i */
public  …..  deleteElement(…..){
….
}

  /** Supression d'un élément passé en argument  */
public  …..  deleteElement(Object pElement){
….
}

}
```

3. Est-ce qu'il est possible de stocker dans cette liste des valeurs entières ? si oui alors expliquer comment ?

Deuxième partie

On utilisant la classe ***SingleLinkedList*** précédente on veut gérer la liste d'attente dans un service d'urgences à l'aide d'une file de priorité de patients. Un patient est caractérisé par son nom, son prénom, son numéro de sécurité sociale et sa priorité, un entier compris entre 1 et 3, 3 étant la priorité maximale (3 : Très urgent, 2 : Urgence moyenne, 1 : Peut attendre). Les patients sont insérés et enlevés de la file selon leur priorité.

Si un patient avec priorité k arrive, il sera rangé en dernier parmi les patients de même priorité. On prend un patient dans la file s'il est le premier arrivé parmi tous ceux ayant la priorité maximale. Il est alors enlevé de la file. Exemple : Supposons que 3 patients arrivent dans l'ordre suivant : Pierre a priorité 1, Tom a priorité 2 et Paul a priorité 2. Si on organise la file en trois sous-files correspondant à chacune des 3 priorités, on aura :



Il n'y a aucun patient de priorité 3. Le patient de priorité maximale est donc Tom. Si on affiche la liste de patients dans l'ordre dans lequel ils seront traités cela devra donner : *Tom, Paul, Pierre*.

Dans cette partie de l'exercice vous devez implanter :

1. Ecrire la classe ***Patient*** qui dispose des méthodes suivantes :

    a- Un constructeur adéquat qui permet d'initialiser un objet de type Patient

b- des méthodes pour donner la priorité d'un patient et pour afficher un patient.

2. Un programme principal qui crée les trois patients de l'exemple et qui affiche les données de chacun.
3. La classe ***FilePrioritePatient*** qui devra être implantée à l'aide d'un tableau de 3 cases, chacune avec une liste de patients pour l'une des 3 priorités. (Pour la liste des patients utiliser la classe ***SingleLinkedList*** précédente en lui ajoutant d'autres méthodes si nécessaire). Le squelette de la classe Java à implanter est :

```
class FilePrioPatients {
SingleLinkedList [ ] f i l e p = new SingleLinkedList [ 3 ] ;
void  insererPatient (Patient p){
….
}
Patient  enleverPatientMax ( ) {
….
}
void  affichePatient ( ) {}}
```

Vous ajouterez dans le programme principal de la question précédente : la création d'une file de priorité en y insérant un par un les trois patients, et enfin, l'affichage de cette file.

Exercice 12 :

Il s'agit de programmer une application simple qui simule la gestion des comptes bancaires. Cette application permet de créer et utiliser autant de comptes bancaires que nécessaires, chaque compte étant un objet, instance de la classe Compte.

Un compte bancaire est identifié par un numéro de compte. Ce numéro de compte est un entier positif permettant de désigner et distinguer sans ambiguïté possible chaque compte géré par l'établissement bancaire. Chaque compte possède donc un numéro unique. Ce numéro est attribué par la banque à l'ouverture du compte et ne peut être modifié par la suite. Dans un souci de simplicité (qui ne traduit pas la réalité) on adoptera la politique suivante pour l'attribution des numéros de compte :

- Générer un nombre de 8 numéros aléatoirement
- Vérifier que ce numéro n'est encore attribué à une personne.
- Si il est déjà pris en régénère un nouveau numéro.

Un compte est associé à une personne (civile ou morale) titulaire du compte, cette personne étant décrite par son nom. Une fois le compte créé, le titulaire du compte ne peut plus être modifié. La somme d'argent disponible sur un compte est exprimée en DH. Cette somme est désignée sous le terme de solde du compte. Ce solde est un nombre décimal qui peut être positif, nul ou négatif. Le solde d'un compte peut être éventuellement (et temporairement) être négatif. Dans ce cas, on dit que le compte est à découvert. Le découvert d'un compte est nul si le solde du compte est positif ou nul, il est égal à la valeur absolue du solde si ce dernier est négatif.

En aucun cas le solde d'un compte ne peut être inférieur à une valeur fixée pour ce compte. Cette valeur est définie comme étant - (moins) le découvert maximal autorisé pour ce compte. Par exemple pour un compte dont le découvert maximal autorisé est 2000 DH, le solde ne pourra pas être inférieur à -2000 DH. Le découvert maximal autorisé peut varier d'un compte à un autre, il est fixé arbitrairement par la banque à la création du compte et peut être ensuite révisé selon les modifications des revenus du titulaire du compte. Créditer un compte consiste à ajouter un montant positif au solde du compte. Débiter un compte consiste à retirer un montant positif au solde du compte. Le solde résultant ne doit en aucun cas être inférieur au découvert maximal autorisé pour ce compte.

Lors d'une opération de retrait, un compte ne peut être débité d'un montant supérieur à une valeur désignée sous le terme de débit maximal autorisé. Comme le découvert maximal autorisé, le débit maximal autorisé peut varier d'un compte à un autre et est fixé arbitrairement par la banque à la création du compte. Il peut être ensuite révisé selon les modifications des revenus du titulaire du compte.

Effectuer un virement consiste à débiter un compte au profit d'un autre compte qui sera crédité du montant du débit.

Lors de la création d'un compte seul le nom du titulaire du compte est indispensable. En l'absence de dépôt initial le solde est fixé à 0. Les valeurs par défaut pour le découvert maximal autorisé et le débit maximal autorisé sont respectivement de 800 DH et 3000 DH. Il est éventuellement possible d'attribuer d'autres valeurs à ces caractéristiques du compte lors de sa création.

Toutes les informations concernant un compte peuvent être consultées : numéro du compte, nom du titulaire, montant du découvert maximal autorisé, montant du débit maximal autorisé, situation du compte (est-il à découvert ?), montant du débit autorisé (fonction du solde courant et du débit maximal autorisé).

1- A partir du cahier des charges élaborer une spécification puis une implémentation sous forme de classes JAVA :
   - Définir les attributs (variables d'instance, variables de classe) de la classe Compte.
   - Identifier et implémenter les méthodes proposées par la classe Compte. Pour chaque méthode on prendra soin, outre la définition de sa signature, de spécifier son comportement sous la forme d'un commentaire JAVA DOC.
   - De proposer un ou plusieurs constructeurs pour la classe Compte. Là aussi on complétera la donnée de la signature de chaque constructeur avec un commentaire JAVA DOC détaillant son utilisation.

2- Tester les méthodes ajoutées.

3- Réaliser une application console permettant d'effectuer les opérations décrites dans le menu ci-dessous :
   1. Créer Compte
   2. Créditer Compte
   3. Débiter Compte

4. Effectuer un virement
5. Afficher un Compte
6. Modifier un Compte