LU2IN002 : Éléments de programmation par objets avec JAVA

Licence de Sciences et Technologies Mention Informatique

Exercices de TME supplémentaires

Année 2022-2023

Introduction

Ce PDF contient des exercices de TME supplémentaires, ainsi que des exercices d'annales possibles de faire sur machine. Le titre de chaque exercice précise la séance de TME *à partir de* laquelle vous pouvez faire ces exercices ainsi que les principales difficultés Java abordées dans l'exercice. Par exemple, si le titre de l'exercice indique "TME3", cela veut dire que cet exercice peut être fait à partir de la séance du TME3.

Exercices possibles de faire à partir du TME1

Exercice 101 – TME1 : Alphabet (boucles, manipulation des caractères)

Q 101.1 Ecrire la classe `Alphabet` qui ne contient que la méthode `main` qui réalise le traitement suivant, en utilisant une boucle `for` :

Q 101.1.1 Afficher les chiffres de 0 à 9, ainsi que leur code ASCII.

Q 101.1.2 Afficher les lettres de l'alphabet de 'A' à 'Z', ainsi que leur code ASCII.

Q 101.2 Recommencer en utilisant une boucle while.

Exercice 102 – TME1 : Rétro-engineering (boucles)

Écrire le programme permettant d'obtenir l'affichage suivant en utilisant une boucle (suite de Fibonacci) :

```
1 2 3 5 8 13 21 34 55
```

Exercice 103 – TME1 : Formule de Newton (classe simple, boucles)

La suite de Newton définie ci-dessous converge vers la racine carrée du nombre `x` :
$U_0 = \frac{x}{2}$ et $U_i = \frac{1}{2}(U_{i-1} + \frac{x}{U_{i-1}})$ pour tout $i \geq 1$
Écrire une classe `SuiteNewton` qui étant donné un nombre x et un réel ε calcule la valeur de la racine de x avec une précision de ε en utilisant la suite de Newton.

Exercice 104 – TME1 : Libellé d'un chèque (*exercice long et fastidieux...*)

Écrire une classe `Libelle` qui traduit en toutes lettres un nombre entier inférieur à 1000 selon les règles usuelles de la langue française.
Par exemple : 123 s'écrit "cent vingt-trois" et 321 s'écrit "trois cent vingt et un".
Le programme doit prendre en compte les règles d'orthographe essentielles :

- les nombres composés inférieurs à cent prennent un trait d'union sauf vingt et un, trente et un, ..., soixante et onze.
- les adjectifs numéraux sont invariables sauf cent et vingt qui se mettent au pluriel quand ils sont multipliés et non suivis d'un autre nombre.

Exercices possibles de faire à partir du TME2

Exercice 105 – TME2 : Et si on jouait un peu au ballon ? (composition simple)

Q 105.1 Un ballon a un diamètre (`int`) et une couleur (`String`). Ecrire une classe `Ballon` avec 2 variables d'instance, un constructeur prenant 2 paramètres et une méthode `toString()` qui retourne par exemple `"ballon bleu 21 cm"`.

Q 105.2 Un joueur a un nom et peut avoir un ballon. Ecrire une classe `Joueur` qui contient les variables d'instance et méthodes suivantes :

- variable `nom` : le nom du joueur (`String`)

- variable `bal` : le ballon du joueur (variable de type `Ballon`)
- constructeur `Joueur(String nom, Ballon bal)`
- constructeur `Joueur(String nom)` qui initialise la variable d'instance `bal` à `null` (le joueur n'a pas le ballon)
- méthode `boolean avoirBallon()` qui retourne vrai si le joueur a le ballon, faux sinon
- méthode `toString()` qui retourne par exemple : `"Olive a le ballon bleu 21 cm"` ou bien `"Olive n'a pas le ballon"`

Q 105.3 Dans la méthode `main` d'une classe `Test`, créer le joueur Olive qui a un ballon, un joueur Tom qui n'a pas de ballon, puis afficher ces joueurs.

Q 105.4 Dans la classe `Joueur`, ajouter une méthode `passerBallon(Joueur j2)` qui affiche différend message en fonction des cas. Par exemple, pour le joueur qui s'appelle Olive et qui veut passer le ballon au joueur appelé Tom, cette méthode peut en fonction des cas afficher l'un des 3 messages suivants :

- `Olive ne peut pas passer le ballon, car il ne l'a pas`
- `Olive ne peut pas se passer le ballon à lui-même`
- `Olive passe le ballon à Tom`

Ajouter des instructions dans le `main` pour tester chacun des cas.

Q 105.5 Dans la classe `Joueur`, ajouter une méthode `Ballon tirer()` qui :

- si le joueur a un ballon :
  - sauve le ballon dans une variable temporaire
  - affiche par exemple `"Olive tire !"`
  - enlève le ballon au joueur
  - retourne le ballon
- si le joueur n'a pas le ballon :
  - affiche par exemple `"Olive n'a pas le ballon, il ne peut pas tirer"`
  - retourne `null` (en Java, si une méthode a un type de retour, il faut qu'elle retourne quelque chose dans tous les cas)

Q 105.6 Dans la méthode `main`, simuler des échanges de ballon entre Olive et Tom (en utilisant une boucle), puis faire tirer au but le joueur qui a le ballon.

Exercices possibles de faire à partir du TME3

Exercice 106 – TME3 : Course avec équipe (composition d'objets)

On propose ici d'étendre l'exercice intitulé "Course de relais 4 fois 100m" du TME2 pour y ajouter le concept d'équipe de coureurs. Remarque : si vous n'avez plus vos fichiers du TME2, il vous suffit de réécrire une classe coureur avec un constructeur sans paramètre, puis de rajouter au fur et à mesure dans cette classe les quelques attributs et méthodes nécessaires pour répondre aux questions de cet exercice.

Q 106.1 Une équipe est composée de 4 coureurs. Ecrire une classe `Equipe` qui comprend :

- `ville` : le nom de la ville de l'équipe (`String`)
- les 4 coureurs de type `Coureur`
- un constructeur `Equipe(String v)` qui crée une équipe de quatre coureurs pour la ville `v`
- la méthode `public String toString()` qui retourne une chaîne de caractères décrivant l'équipe avec ses coureurs. On appellera pour ce faire la méthode `toString()` des coureurs de cette équipe.

Q 106.2 Dans le main, créer une équipe et l'afficher.

Q 106.3 Écrire une méthode `double tempsFinal()` qui rend le temps mis par cette équipe pour courir le 400m.

Q 106.4 Écrire une méthode `Equipe rencontrer (Equipe e)` qui retourne l'équipe gagnante (c'est-à-dire la référence de l'équipe gagnante) dans la compétition opposant l'équipe courante à l'équipe passée en paramètre. L'équipe gagnante est celle qui a mis le moins de temps à courir les 400m. Si les temps sont égaux, la méthode retourne null.

Q 106.5 Créer dans la méthode main deux équipes, les faire courir, afficher les temps mis par chaque équipe ainsi que le résultat de leur rencontre.

Exercice 107 – TME3 : Machine à schtroumpfer (composition et copie d'un objet composite)

On veut construire une machine à schtroumpfer les alipois. Une telle machine a une couleur et est composée d'une partie avant et d'une partie arrière. La partie avant est composée de deux pièces de type A et d'une pièce de type B. La partie arrière est composée d'une pièce de type A et d'une pièce de type C. Une pièce A a un numéro de série et est formée d'une matière (type `String`). Une pièce B a un numéro de série et a une longueur

(type `double`). Une pièce C a un numéro de série et est de luxe ou ordinaire (booléen). Voici les classes `PieceA`, `PieceB`, et `PieceC` telles qu'elles sont fournies :

```
// classe PIECE A
public class PieceA {
        private int noSerie;
        private String matiere;
        public PieceA(int n) { noSerie=n; matiere="bois"; }
        public PieceA(int n,String m) { noSerie=n; matiere=m; }
        public String toString() {
                return "pieceA no "+ noSerie+ " en "+ matiere+"\n";
        }
}
// classe PIECE B
public class PieceB {
        private int noSerie;
        private double longueur;
        public PieceB(int n) { noSerie=n; }
        public PieceB(int n,double l) { noSerie=n; longueur=l; }
        public void setLongueur(double l) { longueur=l; }
        public String toString() {
                return "pieceB no "+ noSerie+ " de longueur "+ longueur +"\n";
        }
}
// classe PIECE C
public class PieceC {
        private int noSerie;
        private boolean luxe;
        public PieceC(int n) { noSerie=n; }
        public PieceC(int n,boolean b) { noSerie=n; luxe=b; }
        public void setLuxe(boolean b) { luxe=b; }
        public String toString() {
                String s="pieceC no "+ noSerie;
                if (luxe) s +=" de luxe \n";
                else s +=" ordinaire\n";
                return s;
        }
}
```

Q 107.1 Écrire les classes `Avant`, `Arriere` et `Machine` (constructeur, toString...). Dans la classe `Avant`, ajouter une méthode pour modifier la longueur de sa pièce B. Dans la classe `Machine`, ajouter une méthode pour modifier la couleur de la machine et une autre méthode pour modifier la longueur de sa pièce avant.

Q 107.2 Écrire une méthode main qui crée une machine `m1`. Il faudra évidemment créer auparavant toutes les pièces, et l'avant et l'arrière dont elle est formée. L'afficher à l'écran.

Q 107.3 Dans cette méthode main, on veut créer une copie de `m1`. Déclarer `m2` ainsi : `Machine m2=m1;` Modifier ensuite la couleur de `m2` et la longueur de la pièce B de son avant. Afficher `m1` et `m2`. Que s'affiche-t-il ? Expliquez le problème et ce qui s'est passé.

Q 107.4 Pour éviter le problème précédent, on va utiliser des constructeurs de copie. Pourquoi faut-il écrire un constructeur par copie dans chaque classe, et non pas seulement dans la classe `Machine` ? Ecrire le constructeur de copie dans chaque classe.

Q 107.5 Créer maintenant une copie `m3` de la machine `m1`, puis modifier la couleur et la longueur de la pièce B de l'avant de `m3`. Affichez `m1` et `m3`. Que s'affiche-t-il maintenant ? Bien vérifier que la couleur et la longueur de la pièce B de l'avant de `m1` n'ont pas été modifiées.

Exercices possibles de faire à partir du TME4

Exercice 108 – TME4 : Histogramme de notes (tableau)

Dans cet exercice, il s'agit d'écrire un programme qui permet de représenter un histogramme de notes entières comprises entre 0 et 20 (c'est-à-dire il y a 21 notes possibles).

Par exemple, si dans une classe, il y a 10 étudiants qui ont obtenus les notes suivantes : 2, 3, 4, 3, 0, 0, 2, 3, 3, 2 (c'est-à-dire 2 étudiants ont obtenu la note 0, 0 étudiant ont obtenu la note 1, 3 étudiants la note 2, 4 étudiants la note 3, 1 étudiant la note 4), le tableau représentant l'histogramme sera : [2, 0, 3, 4, 1].
L'affichage de l'histogramme correspondant donnera :

```
0 | **
1 |
2 | ***
3 | ****
4 | *
```

Q 108.1 On souhaite écrire une classe `Histo` qui affiche un histogramme des notes. Pour cela, vous définirez :
- un attribut tableau `hist` représentant l'histogramme,
- un constructeur sans paramètre qui initialise le tableau hist et met toutes les cases du tableau à la valeur 0,
- une méthode d'ajout d'une note,
- un constructeur qui prend en paramètre un tableau de notes et qui initialise l'histogramme à partir des notes.

Q 108.2 Ajouter à cette classe, une méthode `afficheHistogrammeTableau()` qui affiche l'ensemble des valeurs du tableau histogramme.

Q 108.3 Ajouter à cette classe, une méthode `afficheHistogramme()` qui affiche le résultat sous forme d'un histogramme, c'est-à-dire en associant à chaque élément du tableau une ligne comprenant autant de `*` que la valeur de cet élément. (comme dans l'exemple de l'énoncé)

Q 108.4 Écrire une classe `TestHisto` dont la méthode `main` crée un tableau de notes aléatoires (150 étudiants), une instance de `Histo`, puis qui affiche le résultat sous les deux formes proposées.

Exercice 109 – TME4 : Le jeu de la vie (tableau à 2 dimensions)

Présentation du jeu
Le monde est un damier de 20 cases sur 20 appelées cellules. Une cellule est soit vivante soit morte. Au départ, il y a 80% de cellules mortes. Chaque cellule a 8 voisins (monde torique type chambre à air), c'est-à-dire par exemple que le voisin de droite d'une cellule du bord droit du damier a comme voisine de droite la cellule opposée du damier. Il en est de même pour toutes les cellules du bord du damier, qui ont pour voisines les cellules du bord opposé.
Les règles de vie à chaque nouvelle génération sont les suivantes :
- une cellule meurt si elle a strictement moins de 2 voisins (mort par isolement) ou strictement plus de 3 (mort par surpopulation).
- une cellule naît sur une case vide si celle-ci a exactement 3 voisins.

On visualisera le monde en représentant une cellule vivante par le caractère `"*"` et une cellule morte par le caractère `"."`.

Q 109.1 Classes de base :
- Définir la classe `Monde` qui a quatre variables : la largeur du damier `maxX`, sa hauteur `maxY`, le numéro de la génération courante `noGener` et un tableau de booléens à 2 dimensions `tabCells` (true signifiant que la cellule est vivante, false qu'elle est morte).
- Définir le constructeur `Monde(double seuil)` de cette classe qui crée un monde de `maxX` cellules sur `maxY` cellules avec un pourcentage approximatif de cellules vivantes déterminé par le seuil.
  On pourra utiliser la méthode `random` de la classe `Math` qui génère une valeur aléatoire de type double comprise entre 0.0 inclus et 1.0 exclu.
- Écrire la méthode `public String toString()` qui retourne une chaîne de caractères décrivant ce monde avec son numéro de génération et le tableau de l'état de ses cellules.
- Définir une classe `TestJeuVie` avec la méthode `main` qui crée un monde avec 80% (seuil = 0.8) de cellules mortes et le visualise.

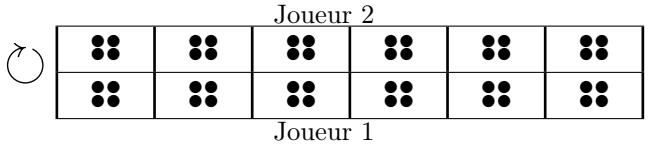
Q 109.2 Ajouter dans la classe `Monde` la méthode `nbVoisins(int nuLign,int nuCol)` qui retourne le nombre de cellules voisines vivantes de la cellule `tabCells(nuLign, nuCol)`. Penser à utiliser la fonction % (modulo). Calculer le nombre de voisins d'une cellule revient à compter le nombre de cellules vivantes dans le carré de 3x3 dont le centre est cette cellule. Un moyen simple de gérer la sortie du damier, dans un monde torique, est de prendre la formule : `(i + maxX) modulo maxX ou (i + maxY) modulo maxY`. Tester cette méthode.

Q 109.3 Ajouter, dans la classe Monde, la méthode `void genSuiv()` qui, à partir du tableau `tabCells` de la classe `Monde`, crée un nouveau tableau en y appliquant les règles du jeu de la vie données plus haut afin d'obtenir

la génération suivante, puis bascule `tabCells` sur ce nouveau tableau. Tester le jeu de la vie.

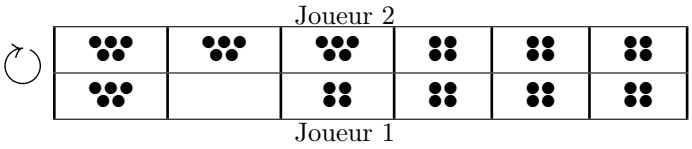
Exercice 110 – TME4 : Awélé (modélisation et tableau)

Règle du jeu :
L'awélé est un jeu à deux joueurs qui se joue avec un plateau formé de 12 cases et de 48 billes réparties au début du jeu à raison de 4 par case (voir figure).

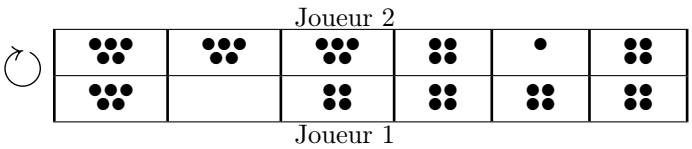


Chaque joueur possède les 6 cases Nord ou les 6 cases Sud du jeu. Les cases sont attribuées au joueur en début de la partie. Les cases du plateau sont toujours parcourues dans le sens indiqué sur la figure (sens des aiguilles d'une montre sur la figure).
Le jeu se joue de la façon suivante :

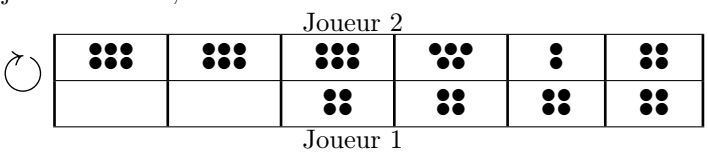
— Un joueur choisit l'une de ses propres cases, puis répartit son contenu en déposant une bille et une seule dans chacune des cases consécutives. Par exemple, si le joueur 1 choisit la case 5 (deuxième case en partant de la gauche des cases au Sud) pour jouer, cela donne le résultat suivant :



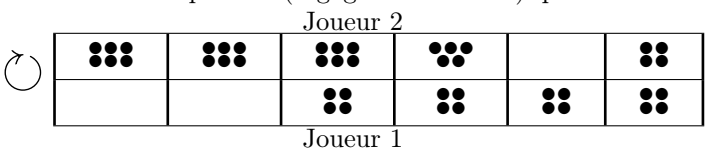
— Un joueur marque un coup gagnant lorsqu'il dépose sa dernière bille (du coup en cours) dans une case adverse qui ne contient qu'une seule bille. Dans cette situation, le joueur gagne alors ces deux billes et les retire du plateau.
Situation avant le coup gagnant :



Le joueur 1 choisit de jouer sa case 6, on obtient alors :



Le joueur 1 retire alors deux billes du plateau (il gagne deux billes) qui devient :



Le joueur peut aussi cumuler des gains si la case précédente a aussi deux billes à la fin du coup. Il en gagne alors deux de plus. Les joueurs jouent tour à tour jusqu'à ce que le nombre total de billes sur le plateau soit inférieur à 12. Le gagnant est le joueur qui a remporté le plus de billes.

Le jeu électronique :
Le jeu électronique de l'Awélé se joue lui aussi à deux joueurs. Le plateau est affiché à l'écran. Pour manipuler les billes, les joueurs désignent les cases du plateau soit en donnant le numéro de la case, soit en cliquant à la souris. Le jeu électronique contrôle le bon déroulement du jeu. Au début du jeu, il affecte les cases du plateau aux joueurs. Il fait jouer les deux joueurs à tour de rôle, comptabilise les points de chacun et désigne le gagnant. Pour jouer, un joueur désigne la case dont il veut répartir les billes. Le jeu répartit automatiquement ses billes dans les cases suivantes et gère les gains en attribuant les billes au joueur gagnant.

Q 110.1 Déterminer les classes nécessaires à la modélisation du jeu électronique en précisant les attributs et les méthodes de chacune des classes.

Q 110.2 Définir une classe `Case` caratérisée par un nombre de billes (`nbBille`), et son propriétaire (`prop` de type `Joueur`). Écrire un constructeur sans paramètre et un constructeur ayant comme paramètre un joueur, ainsi

qu'une méthode `toString()`.

Q 110.3 Définir la classe `Joueur` qui se caractérise par le numéro et le gain du joueur. Écrire un constructeur sans paramètre et un constructeur ayant comme paramètre le numéro du joueur.

Q 110.4 Définir une classe `Jeu` caractérisée par un plateau de cases, un tableau de joueurs et un indicateur pour savoir qui a la main (de type booléen). Écrire un constructeur qui initialise la partie comme indiquée sur la première figure et la méthode `toString()` pour visualiser le plateau.

Q 110.5 Compléter la classe `Jeu` pour, dans le constructeur, définir le joueur qui a la main. Écrire la méthode changeJoueur() qui donne la main à l'autre joueur et une méthode `choisirCase()` qui demande au joueur qui à la main de choisir ue case et qui valide ce choix. Ecrire une méthode `unCoup(int i)` qui réalise le coup à partir de la case numéro i.

Q 110.6 Compléter la classe `Jeu` en y ajouter une méthode `finDePartie()` qui affiche si on est en fin de partie et une méthode `boolean jouer()` qui assure le bon déroulement de la partie tant que la partie n'est pas finie.

Q 110.7 Définir la classe `Awele` qui lance une nouvelle partie et qui gère la fin de partie.

Exercices possibles de faire à partir du TME5

Exercice 111 – TME5 : Génération de noms (tableau de caractères, méthode de classe)

On veut écrire une classe `Nom` qui offrira une *méthode de classe* générant des noms de façon aléatoire. On écrira cette classe avec les variables et méthodes suivantes qu'on testera au fur et à mesure :

Q 111.1 Écrire une méthode de classe `rendAlea(int inf, int sup)` qui rend un entier naturel aléatoire entre inf et sup compris. Aide : lisez la documentation Java (voir site web de l'UE) de la classe `Random`.

Q 111.2 Écrire une méthode de classe `boolean estPair(int n)` qui vérifie que `n` est pair.

Q 111.3 Déclarer en variable `static` deux tableaux de `char` de noms voyelles et consonnes. Initialiser lors de la déclaration le premier avec les consonnes et le second avec les voyelles.

Q 111.4 Écrire les méthodes `rendVoyelle()` et `rendConsonne()` qui rendent respectivement une voyelle et une consonne de façon aléatoire.

Q 111.5 Écrire une méthode `genereNom()` qui rend un nom de longueur aléatoire comprise entre 3 et 6 caractères en générant alternativement une consonne et une voyelle.

Q 111.6 Écrire une classe `TestNom` dont la méthode `main` génère et affiche, dans une boucle, une dizaine de noms générés.

Exercices possibles de faire à partir du TME7

Exercice 112 – TME7 : Compagnie de chemin de fer (héritage, tableau ou `ArrayList`, `instanceof`)

Une compagnie de chemin de fer veut gérer la formation de ses trains, à partir de la description suivante. Un train est formé d'éléments de train. Un élément de train possède un numéro de série et une marque. Un élément de train est soit une motrice, soit un wagon. Une motrice a une puissance. Un wagon a un nombre de portes. Un wagon peut être soit un wagon voyageurs, auquel cas il possède un nombre de places, soit un wagon de marchandise, auquel cas il possède un poids maximum représentant la charge maximale qu'il peut transporter.

Q 112.1 Dessiner la hiérarchie des classes `Train`, `ElemTrain`, `Motrice`, `Wagon`, `WVoyageur` et `WMarchandise`.

Q 112.2 Écrire les classes `ElemTrain` (abstraite), `Wagon` (abstraite), `Motrice`, `WVoyageur` et `WMarchandise` avec au moins un constructeur avec paramètres et une redéfinition de la méthode `public String toString()` qui retourne pour un élément son type et son numéro de série, par exemple : « Wagon Marchandise 10236 (Alstom)». Pour plus de propreté, vous utiliserez un compteur `static` pour générer les numéros de série commençant à 10000.

Q 112.3 Un train possède une motrice et une une suite de wagons. Écrire la classe `Train` avec au minimum un constructeur a un paramètre de type `Motrice` qui construit un train réduit à cette motrice, et ayant donc un

ensemble vide de wagons. Si vous connaissez les `ArrayList`, gérez la suite de wagons à l'aide d'une `ArrayList` de wagons (voir la documentation sur Internet), sinon gérez la suite à l'aide d'un tableau de wagons.

Q 112.4 Ajouter une méthode `void ajoute(Wagon w)` qui ajoute un wagon à la suite de wagons du train.

Q 112.5 Redéfinir la méthode `public String toString()` qui retourne la composition de ce train.

Q 112.6 Écrire une méthode `poids()` qui retourne le poids maximum de marchandise que peut transporter le train. *Indication* : On peut utiliser l'opérateur `instanceof` qui rend vrai si et seulement si un objet est instance d'une classe. Exemple d'utilisation : `if (a instanceof A)...`

Q 112.7 Écrire la méthode principale `public static void main(String[] args)` dans une classe `MainTrain`. Cette méthode crée une motrice, des wagons de voyageur et des wagons de marchandise, crée un train formé de ces éléments, affiche la composition de ce train ainsi que le poids transporté.

Exercice 113 – TME7 : SuperCalculateur (héritage, abstract, tableau, ArrayList)

On considère les ordinateurs d'une entreprise. Un ordinateur a une adresse IP. Les ordinateurs personnels sont des ordinateurs qui ont un seul processeur et qui sont associés au nom d'une personne. Les supercalculateurs sont des ordinateurs qui possèdent entre 2 et 8 processeurs. Une entreprise possède des ordinateurs.

Q 113.1 Un processeur a un identifiant unique et a une fréquence. Ecrire la classe `Processeur` qui contiendra au moins les variables et méthodes suivantes :

- `cpt` : un compteur initialisé à 1 000 000,
- `id` : l'identifiant non-modifiable du processeur au format `"CPUXXXXXXX"` où XXXXXXX est un nombre supérieur à 1 000 000,
- `frequence` : le nombre d'opérations effectuées par le processeur en une seconde, ce nombre sera exprimé en GHz et stocké dans une variable de type `double`,
- un constructeur qui prend en paramètre la fréquence du processeur,
- un constructeur sans paramètre qui appel le premier constructeur pour initialiser aléatoirement la fréquence du processeur entre 1 GHz et 5 GHz (non compris).
- l'accesseur de la variable frequence,
- une méthode `String toString()` qui retourne une chaîne de caractères représentant le processeur. Par exemple, pour le processeur dont l'id est "CPU1000001" et dont la fréquence est 3.06GHz, cette chaîne a le format : `"Processeur CPU1000001 (3.06GHz)"`.

Q 113.2 Un ordinateur a une adresse IP. Ecrire la classe `Ordinateur` qui contiendra au moins les variables et méthodes suivantes :

- `adresseIP` : une chaîne de caractères,
- un constructeur prenant en paramètre l'adresse IP de l'ordinateur,
- une méthode `String toString()` qui retourne l'adresse IP de l'ordinateur,
- une méthode `double getFrequence()` dont le but est de retourner le nombre d'opérations que peut effectuer l'ordinateur en une seconde. Si l'ordinateur possède un seul processeur alors la fréquence de l'ordinateur est égale à celle du processeur. Si l'ordinateur possède plusieurs processeurs, alors la fréquence de l'ordinateur est égale à 90% de la somme totale des fréquences des processeurs de l'ordinateur.

Questions : La fréquence de l'ordinateur dépend du nombre de processeurs de cet ordinateur, qu'en concluez-vous ? Que faudra-t-il faire dans les classes qui héritent de la classe `Ordinateur` ?

Q 113.3 Un ordinateur personnel est un ordinateur qui possède un processeur et connaît le nom de son propriétaire. Ecrire la classe `OrdinateurPersonnel` et qui contient notamment les variables et méthodes suivantes :

- `pro` : le processeur de l'ordinateur,
- `nomProprio` : le nom du propriétaire de l'ordinateur,
- un constructeur qui prend en paramètre le nom du propriétaire et l'adresse IP, et qui initialise le processeur de l'ordinateur avec une fréquence initialisée aléatoirement entre 1 et 5 (non compris) GHz.
- l'accesseur de `nomProprio`
- une méthode `String toString()` qui retourne une chaîne au format suivant : `"Ordinateur personnel 192.168.0.2 Processeur CPU1000024 (1.79GHz) appartenant à Jacques"`

Q 113.4 Un supercalculateur est un ordinateur qui possède entre 2 et 8 processeurs (le nombre de processeurs est choisi aléatoirement. Ecrire la classe `SuperCalculateur` qui contient notamment un tableau de processeurs `tabPro` et un constructeur prenant en paramètre l'adresse IP.

Q 113.5 Une entreprise a un nom et des ordinateurs représentés par une liste au sens `ArrayList` d'ordinateurs (voir la documentation de `ArrayList` sur Internet). Ecrire la classe `Entreprise` qui possède notamment :

- une méthode pour ajouter un ordinateur dans l'entreprise
- une méthode `double getFrequence()` qui retourne la somme des fréquences de tous les ordinateurs

— une méthode `afficherNoms()` qui affiche le nom de toutes les personnes de l'entreprise qui ont un ordinateur personnel

Q 113.6 Dans une classe `Test`, ajouter une méthode `main` qui crée une entreprise, puis qui ajoute 10 ordinateurs (aléatoirement 30% de supercalculateurs et 70% d'ordinateurs personnels). Le nom des propriétaires d'ordinateurs sera Bob1, puis Bob2, puis Bob3 et ainsi de suite. L'adresse IP de chaque ordinateur sera `"192.168.0.0"`, puis `"192.168.0.1"`, puis `"192.168.0.2"` et ainsi de suite. Afficher ensuite la fréquence totale disponible dans l'entreprise, ainsi que le nom des personnes qui ont un ordinateur personnel.

Exercice 114 – TME7 : Les Robots Pollueurs (héritage et abstract, tableau)

Le monde est constitué d'une matrice de cases. Différents types de robots agissent dans ce monde : des robots pollueurs et des robots nettoyeurs. Les robots pollueurs se baladent dans le monde et déposent des papiers gras sur les cases où ils se trouvent. Les robots nettoyeurs, quant à eux, ne supportent pas la vue d'un papier gras et l'enlèvent dès qu'ils en voient un sur une case.
Les robots pollueurs sont de deux sortes : robots sauteurs et robots qui vont tout droit. Chaque robot pollueur suit son parcours et dépose un papier gras sur chaque case rencontrée. Le parcours précis des robots pollueurs sera donné dans la suite.
Les robots nettoyeurs parcourent méthodiquement le monde en "boustrophédon", c'est-à-dire qu'ils parcourent la première ligne case par case, puis arrivé en bout de ligne font demi-tour, parcourent la deuxième ligne en sens inverse, et ainsi de suite jusqu'à la dernière case de la dernière ligne. Ils enlèvent, s'il s'en trouve un, le papier gras de la case où ils se trouvent. Parmi les robots nettoyeurs, certains sont distraits et n'enlèvent qu'un papier sur deux.
Les constructeurs, accesseurs, modifieurs et les méthodes `toString()` seront créés dans chaque classe suivant les besoins.

Q 114.1 Dessiner la hiérarchie des classes correspondant à la description ci-dessus.

Q 114.2 Ecrire la classe `Monde` qui contient les variables d'instance, le constructeur et les méthodes suivantes (déterminer si elles sont d'instance ou de classe) :
- le nombre de lignes `nbL`, le nombre de colonnes `nbC`, et une matrice booléenne `mat` de `nbL` lignes et `nbC` colonnes (vrai signifiant la présence d'un papier gras).
- un constructeur avec paramètres et un sans paramètres `Monde()` : ce constructeur par défaut crée un monde 10*10 sans papiers gras.
- `public String toString()` : retourne une chaîne de caractères décrivant le monde. On représentera un papier gras par le caractère `'o'`, rien par le caractère `'.'`(point). Le caractère de passage à la ligne est `"\n"`.
- `metPapierGras(int i, int j)` : met un papier gras dans la case (i, j).
- `prendPapierGras(int i, int j)` : enlève le papier gras de la case (i, j).
- `estSale(int i, int j)` : teste si la case (i, j) a un papier gras.
- `nbPapiersGras()` qui rend le nombre de papier gras dans le monde.

Q 114.3 Ecrire dans une classe `TestRobot` la méthode `main` et y créer un monde de 10 lignes et 10 colonnes. Y tester les méthodes `metPapierGras`, `prendPapierGras`, `estSale`.
Dans la suite on n'oubliera pas de reporter la hiérarchie construite en question 1 dans la définition des différentes classes (mot clé extends). Chaque fois qu'on écrira une variable ou une méthode, on en enrichira le dessin de la hiérarchie des classes.

Q 114.4 Ecrire la classe `Robot` qui est abstraite car elle n'aura aucune instance, et qui contient les champs et méthodes suivantes :
- `posx, posy` : position du robot sur le monde.
- `m` : variable de type Monde. En effet il faut que le robot connaisse le monde pour pouvoir s'y déplacer et agir.
- deux constructeurs : `Robot(int x,int y,Monde m)` qui crée un robot à la position (x,y) et `Robot(Monde m)` qui crée un robot avec une position aléatoire. Le constructeur `Robot(Monde m)` doit appeler l'autre constructeur.
- `vaEn(int i, int j)` : se déplace en (i, j).
- `parcourir()` : méthode abstraite qui sera définie dans les sous-classes.

Q 114.5 Ecrire la classe `RobotPollueur` (également abstraite car elle n'aura pas d'instance) qui contient les champs et méthodes :
- `polluer()` : met un papier gras là où ce robot se trouve dans le monde.
- constructeurs

Q 114.6 Ecrire la classe `PollueurToutDroit` qui contient les méthodes suivantes :

— un constructeur
— `parcourir()` : cette méthode définit la méthode parcourir abstraite de la classe Robot. Elle décrit un parcours de ce robot dans le monde. Le robot se positionne d'abord dans sa colonne de départ en case (0, ColDepart), puis il va tout droit vers le sud en visitant chaque case de la colonne et dépose un papier gras sur chacune d'elle. Il s'arrête dans la dernière case de la colonne.

Déclarer dans la méthode main deux variables de type Robot qui référenceront deux instances de PollueurToutdroit et tester leur méthode parcourir(). Afficher l'état du monde après chaque parcours. Comment est choisi le corps de la méthode parcourir pour ces deux instances ?

Q 114.7 Ecrire la classe `PollueurSauteur`, qui contient les champs et méthodes suivants :
— `deltax` représentant la taille du saut effectué à chaque déplacement du sauteur ;
— un constructeur ;
— `parcourir()` : cette méthode définit la méthode parcourir abstraite de la classe `Robot`. Elle décrit un parcours de ce robot dans le monde. Le robot se positionne d'abord dans sa colonne de départ en case `(0, coldepart)`, puis il saute de façon analogue au cavalier des Echecs et va en `(1, coldepart+deltax)`, puis en `(2, colDepart)`, puis en `(3, colDepart+deltax)`, etc. Si la colonne sort de l'échiqier, on revient au début. Par exemple, la colonne `nbC`, qui n'existe pas, sera tranformée en `nbC modulo nbC`, c'est-à-dire colonne 0. Chaque case rencontrée est souillée d'un papier gras. Le robot s'arrête lorsqu'il atteint la dernière ligne.

Déclarer dans la méthode main deux variables de type Robot qui référenceront deux instances de `PollueurSauteur` avec des `colDepart` et des `deltax` différents et tester leur méthode `parcourir()`.

Q 114.8 Ecrire la classe `RobotNettoyeur`, qui contient les méthodes suivantes :
— `nettoyer()` : enlève le papier gras de la case où se trouve ce robot,
— un constructeur,
— `parcourir()` : cette méthode définit la méthode parcourir abstraite de la classe `Robot`. Elle décrit un parcours complet de ce robot dans le monde. Il part de la case $(0, 0)$ et parcourt le monde en "boustrophédon". Si la case est sale, il enlève le papier gras qui s'y trouve.

Créer de même dans la méthode `main` deux `RobotNettoyeur` dont les références sont de type `Robot` et tester leur méthode `parcourir()` après le passage des robots pollueurs.

Q 114.9 Ecrire la classe `NettoyeurDistrait` qui contient les méthodes suivantes :
— un constructeur,
— `parcourir()` : cette méthode redéfinit la méthode `parcourir()`. Elle décrit un parcours complet de ce robot dans le monde. C'est le même parcours que celui des robots nettoyeurs mais comme il est distrait, il n'enlève qu'un papier sur deux.

Créer dans le main un `NettoyeurDistrait` dont la référence est de type `Robot` ; tester sa méthode `parcourir()` après le passage d'un pollueur.
Indications :
— prendre un compteur qu'on incrémentera pendant le parcours sur chaque case sale.
— utiliser l'opérateur % qui exprime la fonction modulo.

Q 114.10 Supposons qu'il y ait au départ n papiers gras dans le monde. Combien faudra-t-il lancer de nettoyeurs distraits pour qu'il n'y ait plus un seul papier ?

Partie facultative

Il s'agit dans cette partie d'enrichir le programme précédent des Robots pour en faire un jeu. L'utilisateur pourra parier sur l'état de propreté du monde après le lancement d'un ensemble de robots tirés au sort. L'état de propreté est défini par un entier qui est le seuil de propreté : si le nombre de papiers gras est en dessous de de ce seuil, le monde sera considéré comme propre, et comme sale sinon.

Q 114.11 Écrire dans la classe `Robot` une méthode `Robot robotAlea(LeMonde m)` qui rend un robot déterminé aléatoirement.

Q 114.12 Écrire une classe `Fabrique` qui a comme champ (entre autres) un tableau de `Robot`. On y mettra un constructeur, la méthode `ajoute(Robot r)` qui ajoute un robot dans la table, et une méthode `lancerRobots()`, qui lancera successivement la méthode `parcourir()` pour chaque robot du tableau, et affichera l'état du monde après chaque parcours.

Q 114.13 Écrire une classe `Jeu`, qui contiendra les paramètres du jeu : taille du monde, nombre de robots total, seuil de propreté, ainsi que les entités du jeu : le monde, la fabrique, le pari du joueur, l'état du monde après le passage des robots. On y écrira les méthodes suivantes :
— `initJeu()` qui initialise le monde et la fabrique
— `prendrePari()` qui demande à l'utilisateur de choisir entre deux options « le monde sera propre » ou bien « le monde sera sale » (après le passage des robots)

— `lancerJeu()` qui prend le pari et lance les robots de la fabrique
— `finjeu()` qui détermine si le joueur a gagné et affiche le résultat.

Q 114.14 Lancer le jeu dans le main.

Q 114.15 Enrichir le jeu avec de nouveaux robots : robot rebelle qui ne fait pas ce qu'on lui dit, robot qui tombe en panne, robot traitre qui tout d'un coup change de camp, etc..

Exercices possibles de faire à partir du TME8

Exercice 115 – TME8 : Interface `Reversible` (interface)

Une interface correspond à une propriété. Nous envisageons dans cet exercice la propriété de reversibilité. Pour une chaine de caractères, il s'agit de pouvoir la lire à l'envers lorsqu'on le souhaite, pour un tableau, de prendre les éléments dans l'ordre opposé. Nous choisissons arbitrairement de définir cette propriété sans argument et sans retour : il s'agit juste de modifier l'élément invoquant la méthode.

Q 115.1 Donner le code de l'interface `Reversible`.

Q 115.2 Donner le code de la classe `StringReversible`. Nous rappelons que la classe `String` est immutable et `final`. Vous aurez besoin des méthodes de `String` suivantes : `int length()` qui renvoie la longueur de la chaine et `char charAt(int i)` qui renvoie le caractère à la position `i`.

Q 115.3 Donner deux exemples d'utilisation dans un `main`. Est-il possible de déclarer une variable de type `Reversible` ?

Q 115.4 Nous souhaitons maintenant créer une structure de type `ArrayList` réversible. Donner le code étendant l'`ArrayList<Object>`, ajoutant les méthodes nécessaire (dont `toString()` et surchargeant la méthode `get`.
NB : ajouter un attribut booléen indiquant si la structure est renversée ou pas.

Q 115.5 Ajouter quelques lignes de code pour rendre la réversibilité récursive quand c'est possible dans la structure précédente. Par exemple, quand la liste contient des `StringReversible`, nous souhaitons renverser la liste ET renverser les éléments de la liste si c'est possible.

Q 115.6 (Option) Proposer une seconde implémentation de la structure de données récursive basée sur la composition et non plus sur l'héritage (attribut `ArrayList` au lieu de `extends ArrayList`)

Exercice 116 – TME8 : Interface `Comparable` (interface)

Nous nous plaçons maintenant comme utilisateur d'un cadre défini pour les interfaces. Nous avons besoin de trier une liste de vecteurs en fonction de leur norme. Nous disposons de la classe de base :

```
1 public class Vecteur {
2     private double x,y;
3     public Vecteur(double x, double y){
4         this.x = x;
5         this.y = y;
6     }
7     public double norme(){return Math.sqrt(x*x+y*y);}
8 }
```

La Javadoc indique : (1) dans la classe `Collections` :

```
1 static <T extends Comparable<? super T>> void sort(List<T> list)
2 // Sorts the specified list into ascending order, according to the natural ordering of
      its elements.
3 static <T> void sort(List<T> list, Comparator<? super T> c)
4 // Sorts the specified list according to the order induced by the specified comparator.
```

(2) Interface Comparable

```
1 int      compareTo(T o) // Compares this object with the specified object for order.
2 // si x<y alors, x.compareTo(y) < 0
3 // si x.equals(y) alors  x.compareTo(y) == 0
4 // sinon x.compareTo(y) > 0
```

(3) Interface Comparator

Q 116.1 Indiquer les modifications à effectuer dans la classe `Vecteur` pour utiliser `Comparable`

Q 116.2 Donner le code d'un main effectuant le tri d'une liste de `Vecteur` générée aléatoirement par rapport à leurs normes.

Q 116.3 Donner la procédure et le code pour utiliser un `Comparator`. Quel est l'avantage de cette approche ?

Exercices possibles de faire à partir du TME11

Exercice 117 – TME11 : Traitement de texte (flux)

Rappel : `String` est une classe immutable, c'est-à-dire qu'une variable de type `String` ne peut pas être modifiée. Lorsque l'on pense modifier un objet `String`, en vérité, on crée un nouvel objet `String` à partir de l'ancien.

Q 117.1 Écrire une méthode `String saisie()` qui demande à l'utilisateur de saisir une ligne de texte tant que la ligne entrée par l'utilisateur est différente de la chaîne "_fin_". Cette méthode retourne une chaîne de caractères contenant la concaténation de toutes les lignes saisies. Proposez une première solution utilisant des concaténations de `String`. Puis proposez une deuxième solution utilisant un seul objet `StringWriter`.

Q 117.2 Écrire une méthode `affiche(String fichier)` affichant le contenu du fichier dont le nom est passé en paramètre.

Q 117.3 Écrire une méthode `afficheLignes(String fichier)` affichant, en numérotant les lignes, le contenu du fichier passé en paramètre.

Q 117.4 Écrire une méthode `ecrireTexte(String fichier)` permettant de créer un nouveau fichier contenant un texte saisi par l'utilisateur.

Q 117.5 Écrire une méthode `ajouteTexte(String fichier)` permettant d'ajouter, en fin de fichier passé en paramètre, du texte saisi par l'utilisateur.

Q 117.6 Écrire une méthode `replace(int num, String newLigne, String fichier)` permettant de remplacer, dans le fichier passé en paramètre, la ligne numéro `num` par la nouvelle ligne `newLigne`.

Q 117.7 Écrire un programme proposant à l'utilisateur un menu lui permettant d'éditer un fichier dont le chemin est passé en argument. Exemple :

```
Fichier "Texte.txt"
1. Ajouter texte
2. Afficher fichier
3. Remplacer ligne
4. Quitter
```

Exercice 118 – TME11 : Copie de fichiers binaires (flux)

Q 118.1 Écrire un programme permettant de copier un fichier binaire dont le nom est donné en premier argument sous le nom donné en second argument.

Exercice 119 – TME11 : Mise en mémoire tampon (flux)

La mise en mémoire tampon des données lues permet d'améliorer les performances des flux sur une entité. Par l'utilisation directe d'un objet `Reader`, les caractères sont lus un par un dans le flux, ce qui est très peu efficace. La classe `BufferedReader` (existe aussi pour `BufferedInputStream` pour les octets) permet la mise en mémoire tampon des données lues avant leur transmission au programme.
En outre, elle simplifie l'utilisation du `Reader` en définissant notamment une méthode `String readLine()` permettant de lire les données ligne après ligne plutôt que caractère après caractère (toutes les méthodes de `Reader` sont disponibles dans cette classe mais avec une meilleure gestion de la mémoire).

Q 119.1 Sachant que la construction d'un `BufferedReader` se fait en passant un flux `Reader` en paramètre, écrivez l'ouverture d'un flux de lecture avec utilisation de la mémoire tampon sur un fichier "text.txt" du répertoire

courant.

Q 119.2 Écrire une méthode `afficheLignesFichier(String fichier)` qui affiche ligne après ligne le texte du fichier dont le chemin est passé en paramètre.

Q 119.3 Sachant qu'il est également recommandé par souci d'efficacité d'encapsuler tout flux en écriture dans un objet `BufferedWriter` (resp. `BufferedStream` pour l'écriture d'octets), écrire une classe `Ecrivain` ouvrant un flux en écriture sur un fichier à sa construction et disposant des méthodes données ci-dessus pour la classe `PrintWriter` (sauf méthode `flush()`). On pourra donner une version avec héritage et une version sans.

Exercice 120 – TME11 : Production automatique de compte rendu TME (flux)

L'objectif de cet exercice est d'utiliser les connaissances acquises sur la lecture et l'écriture de fichier pour programmer un outil de production automatique de compte rendu de TME.
On considère que l'utilisateur dispose d'une arborescence de fichiers (telle que celle de votre répertoire) prenant racine en un répertoire LU2IN002. Ce répertoire contient un répertoire par TME (numérotés de TME1 à TME11), chacun d'entre eux contenant eux mêmes un répertoire par exercice (Exo1, Exo2, ... ExoN). On considère également que l'on dispose d'un fichier "etudiants.txt" dans le répertoire LU2IN002 contenant les prenoms, noms et numeros d'etudiants des utilisateurs du programme (une ligne par étudiant). Le fichier doit se terminer par une ligne "Groupe : <numero du groupe>". Enfin, chaque répertoire d'exercice contient deux fichiers "intitule.txt" et "executions.txt", le premier contenant l'énoncé de l'exercice, le second contenant les résultats d'exécution des programmes ainsi que les observations qui ont pu avoir été faites.

Q 120.1 Écrire un programme `RenduTMEProducer` prenant en argument le chemin du répertoire de TME concerné par le compte rendu et produisant en racine de ce répertoire un fichier "compteRenduTME.txt" de la forme de celui que vous avez l'habitude de rendre en fin de TME.

Exercice 121 – TME11 : Classe Clavier (flux)

Nous avons vu la manière d'écrire ou lire dans des fichiers. L'écriture sur la sortie standard (tel qu'on l'a souvent pratiqué par `System.out.println` sans trop savoir à quoi cela correspondait) ou la lecture à partir de l'entrée standard utilisent également des flux en lecture/écriture :

- La sortie standard `System.out` correspond à un flux `PrintWriter` (c'est pourquoi on peut utiliser la méthode `println` sur cet objet)
- L'entrée standard `System.in` correspond à un flux `InputStream` (flux permettant de lire des octets à partir d'une source)

Pour la sortie, aucun problème, on sait déja le faire : `System.out.println("texte a afficher");`
Pour l'entrée, c'est un peu plus compliqué : il s'agit de transformer les octets lus à partir de l'objet `InputStreamReader` en caractères que l'on sait manipuler.

Q 121.1 Sachant que le paquetage `java.io` contient une classe de flux `InputStreamReader` permettant de lire des caractères à partir d'un flux entrant d'octets, réécrire le code de la classe `Clavier`, notamment :

- La fonction statique `String SaisirLigne(String message)`
- La fonction statique `int SaisirEntier(String message)`

Exercices d'annales possibles de faire en TME

Exercice 122 – Examen 2015-2016 : machine à café (héritage,equals,ArrayList,exceptions)

On considère une machine à café. A la base du système, nous allons d'abord considérer les **ingrédients** : le **café**, l'**eau**, le **lait**, le **chocolat**. La **machine à café** est ensuite constituée de **réservoirs** pour les **ingrédients** et de **recettes** (expresso, café allongé, café au lait, chocolat chaud...). Afin d'éviter de recharger la machine en eau trop souvent, elle sera reliée à un **robinet** qui sera modélisé comme un réservoir de capacité infinie.

Éléments de base

Q 122.1 Hiérarchie(s) de classes. Situer tous les éléments en gras de la description dans une arborescence de classes en utilisant les liens d'héritage et de composition. Indiquer la/les classe(s) abstraite(s).

Q 122.2 Donner le code de la classe abstraite `Ingredient` contenant un attribut `String nom`, un constructeur à un argument pour initialiser ce nom, un méthode `toString` (dans la suite de l'exercice, le nom servira à décrire

l'ingrédient, e.g. `"café"`, `"chocolat"`...). Une classe abstraite a-t-elle forcément une méthode abstraite ?

Q 122.3 Ajouter une méthode standard `equals` testant l'égalité structurelle entre 2 ingrédients (c'est à dire entre leurs `noms`) en traitant le cas où `nom` n'est pas instancié (`null`).
Note : attention à bien considérer la classe `String` comme un objet et pas un type de base.

Q 122.4 Donner le code de la classe `Cafe` héritant d'`Ingredient`. Le constructeur ne prend pas d'argument : le `nom` étant toujours `cafe`.
On imagine dans la suite du problème que tous les ingrédients (classes `Eau`, `Lait` et `Chocolat`) ont également été créés avec un constructeur sans paramètre.

Reservoir

Q 122.5 Donner le code d'une classe `RecuperationIngredientException` qui étend les exceptions et qui a un constructeur à un argument (`String message`).

Q 122.6 Le réservoir a pour attribut un `ingredient`, une `capacite` (un réel exprimé en litre), un `niveau` (également réel exprimé en litre). A la construction le réservoir est plein. Le réservoir dispose d'un accesseur sur l'ingrédient et d'une méthode `remplir` qui le remplit totalement. Donner le code de la classe.

Q 122.7 Le réservoir a aussi une méthode `recuperer` qui prend en argument un réel (la quantité souhaitée par le client) : la méthode teste si le `niveau` est suffisant et, dans l'affirmative, décrémente le `niveau`, sinon elle lève une `RecuperationIngredientException` avec un message expliquant le problème (type d'ingrédient et quantité disponible). Une fois sur mille (aléatoirement), le réservoir connait une défaillance et n'est pas capable de délivrer l'ingrédient : vous lèverez une exception dans ce cas, également avec un message explicite.
Donner le code de la méthode en portant une attention particulière à la signature.

Q 122.8 Donner le code de la classe `Robinet`, qui est un réservoir de capacité infinie (`Double.POSITIVE_INFINITY`) qui rencontre un problème quand on s'en sert une fois sur 500 (aléatoirement). Le message à donner à l'utilisateur est de vérifier que le robinet est bien ouvert.

Recette Une recette est composée d'un tableau d'ingrédients de taille fixe et d'un tableau de réels indiquant les quantités (toujours en litre). Elle a aussi un prix (réel, en euros) et un nom. Le constructeur prend en arguments tous les éléments nécessaires à l'initialisation des attributs. La classe possède des accesseurs sur tous ses champs.

Q 122.9 Donner le code de la classe `Recette` : attributs + constructeur + accesseurs.

Machine La machine est composée de deux listes (voir la documentation de `ArrayList` sur Internet) de recettes et de réservoirs. La machine gère un crédit (réel, en euros) et elle a un identifiant entier unique que vous initialiserez avec un compteur static. De manière générale, la machine fonctionne de la manière suivante. D'abord, l'utilisateur ajoute de l'argent à la machine, puis sélectionne une recette, enfin la machine prépare la mixture en récupérant chaque ingrédient dans son réservoir. La machine sait rendre la monnaie (dans la pratique, on mettra simplement le crédit disponible à 0).

Q 122.10 Donner le code de base de la classe `Machine` avec un constructeur sans argument et deux méthodes `public void ajouterReservoir(Reservoir r)` et `public void ajouterRecette(Recette r)` pour configurer la machine. La classe possède aussi une méthode `public void ajouterCredit(double d)` pour mettre de l'argent (en euros) et une méthode `public void rendreLaMonnaie()` qui met le crédit à 0.

Q 122.11 Donner le code de la méthode `public void remplir()` qui correspond à l'action de l'agent d'entretien consistant à remplir tous les réservoirs au maximum.

Q 122.12 Nous allons maintenant procéder au *check-up* de la machine pour vérifier que tout est OK. Nous avons besoin d'une méthode `private Reservoir trouverReservoir(Ingredient i)` qui retourne le réservoir associé à l'ingrédient `i` s'il existe dans la machine et `null` sinon. On suppose qu'il n'y a qu'un réservoir par ingrédient pour éviter les complications. Donner le code de cette méthode et expliquer pourquoi nous l'avons déclaré `private`.

Q 122.13 Donner le code de la méthode `public boolean checkup()` qui vérifie si toutes les recettes sont réalisables, c'est à dire, si tous les ingrédients de toutes les recettes sont bien disponibles dans la machine (à la première erreur, le test est interrompu et retourne `false`). Cette méthode affiche dans la console le nom des recettes avec la mention OK à coté si la recette est réalisable. Elle retourne `true` si toutes les recettes sont OK.

Q 122.14 Donner le code de la méthode `public boolean commander(int ri)` correspondant à l'appui sur le bouton `ri` de la machine. La machine indique la recette sélectionnée (si elle existe), vérifie que l'utilisateur a assez de crédit et prépare la recette en affichant des messages au fur et à mesure de la préparation. En cas de problème lors de la préparation, la machine affiche un message et retourne `false`. Le client n'est débité que si

tout s'est bien passé (dans ce cas, on retourne `true`). Note : on considère ici que le *check-up* a été passé avec succès, tous les réservoirs associés à une recette sont toujours disponibles (ils peuvent simplement être vides ou en panne).

Q 122.15 Donner le code du programme `TestMachineACafe` permettant de valider le bon fonctionnement de la machine : création d'une recette simple à 2 ingrédients (et ajout dans la machine), création des 2 réservoirs associés (et ajout dans la machine), check-up, test sur la monnaie disponible, test pour distribuer des boissons jusqu'à provoquer une erreur.

Q 122.16 Pour créer une machine connectée, on invente un nouveau type de `ReservoirConnecte` qui est construit avec 3 arguments `(Ingredient ingredient, double capacite, String adresse)`. Il a une méthode `void mailTo(String message)` permettant d'envoyer un mail à la compagnie qui gère la machine (en utilisant l'`adresse` donnée lors de la construction). La machine envoie automatiquement un mail lorsqu'un réservoir passe à un niveau de remplissage inférieur à 10% (dans la pratique, on affiche simplement un message dans la console). Donner le code de cette classe.

Q 122.17 L'architecture de la machine est-elle satisfaisante et évolutive ? Justifier en expliquant la démarche pour ajouter un programme soupe à la tomate.

Exercice 123 – Examen 2007-2008 S1 : Aquarium (héritage, ArrayList, exceptions)

L'objet de ce problème consiste à réaliser un aquarium virtuel de taille 500×500 dans lequel évoluent des thons et des requins. On suppose que l'on dispose de la classe `Point` ci-dessous :

```
1 public class Point {
2   public int x,y;
3   public Point(int x, int y){ this.x=x;this.y=y; }
4   public Point() {       /* initialise x et y entre 0 et 499 */
5     this((int)(Math.random() * 500),(int)(Math.random() * 500));
6   }
7   public String toString() { return "("+x+","+y+")␣"; }
8   public double distanceTo(Point p) {
9     int dx = p.x-x;
10    int dy = p.y-y;
11    return Math.sqrt(dx*dx+dy*dy);
12  }
13 }
```

La classe `ArrayList` est une une classe prédéfinie en java qui se trouve dans le package `java.util`. Elle permet de stocker des objets (un peu comme dans un tableau mais sans se préoccuper de la taille du tableau). Pour utiliser cette classe, il faut indiquer dans quel package elle se trouve, en rajoutant en haut de votre fichier : `import java.util.ArrayList;` L'utilisation de cette classe nécessite de préciser le type E des objets qui sont dans la liste. Pour cela, on indique le type des objets entre `<...>`.
Par exemple, voici un exemple d'utilisation des méthodes de cette classe quand ont veut créer une liste de chaines de caractères `(String)` :

```
1 ArrayList<String> liste=new ArrayList<String>();
2 liste.add(new String("Bonjour"));
3 String s=liste.get(0); // recuperation objet a la position zero
4 System.out.println("Taille␣du␣tableau~:␣"+liste.size());
```

Vous utiliserez les deux classes ci-dessus pour ce problème.

Q 123.1 Classe `Poisson`

Q 123.1.1 Définissez une classe abstraite `Poisson` qui possède un attribut `protected position` de type `Point`, avec son constructeur qui assigne à ce poisson une position aléatoire entre $(0, 0)$ et $(499, 499)$. Y mettre l'accesseur public de la position ainsi qu'une méthode abstraite `void move(Point cible)` qui sera définie dans des classes filles. Le point passé en paramètre est le point visé par le mouvement.

Q 123.1.2 Définissez dans la classe `Poisson` une méthode `void verifPosition()` qui replace le poisson dans l'aquarium s'il n'y est pas : si son abscisse ou son ordonnée sont en dehors de l'intervalle $[0, 499]$, on l'y ramènera par un modulo 500.

Q 123.1.3 Définissez une autre classe `PoissonInconnuException` qui étend `Exception` et servira à gérer les cas où un poisson n'est pas d'un type connu.

Q 123.2 Classe `Requin`

Q 123.2.1 Définissez une classe `Requin` qui hérite de `Poisson` et représente un requin. La méthode

`toString()` rend la chaîne `"requin"` suivie des coordonnées du requin, par exemple `"requin(450,200)"`. Définissez-y la méthode `void move(Point cible)` qui assure le déplacement du requin. Le comportement de cette méthode consiste à parcourir la moitié du chemin qui le sépare du point cible, puis à vérifier que le requin est toujours dans l'aquarium (et si ce n'est pas le cas, de l'y replacer) en appelant la méthode définie plus haut `verifPosition()`.

Q 123.3 Classe `Thon`

Q 123.3.1 Définissez une classe `Thon` qui hérite de `Poisson` et représente un thon. Écrivez-y, en plus du constructeur, la méthode `void move(Point cible)`, qui assure le déplacement du thon. Le comportement est le suivant : si le point cible est à une distance supérieure à 60, le thon se déplace aléatoirement en ajoutant à chaque coordonnée de sa position une valeur aléatoire comprise entre -15 et +15. Sinon, le thon parcourt la moitié du chemin qui le sépare du point. Puis on remet si besoin le poisson dans l'aquarium en appelant la méthode `verifPosition()`. La méthode `toString()` rend la chaîne "thon" suivie des coordonnées du thon, par exemple `"thon(450,200)"`.

Q 123.4 Classe `PoissonList`
Cette classe `PoissonList` est destinée à gérer la liste des poissons présents dans l'aquarium.

Q 123.4.1 Définissez la classe `PoissonList` qui hérite de `ArrayList`, avec un constructeur sans paramètres et un constructeur de copie, comme dans la classe `ArrayList` dont elle hérite.

Q 123.4.2 Dans cette classe `PoissonList` ajouter une méthode `nbThons()` qui rend le nombre de thons dans cette liste.
Indication : on pourra utiliser l'opérateur `instanceof` qui permet de savoir si un objet est instance d'une classe : l'expression `unObjet instanceof UneClasse` rend `true` si et seulement si l'objet `unObjet` est une instance de la classe `UneClasse`.

Q 123.4.3 Dans cette classe, ajoutez une méthode `int rangPoissonProche(int index)` qui renvoie l'indice du poisson le plus proche dans l'aquarium du poisson dont l'indice est passé en paramètre.

Q 123.4.4 Dans cette classe, ajoutez une méthode `void bougeTousPoissons()`. Cette méthode déplace tous les poissons (thons et requins) en appelant leur méthode `move(Point cible)`, où cible est soit la position du poisson le plus proche (au sens de la question précédente) si celui-ci est un thon, soit le centre de l'aquarium - le point de coordonnées (250,250) - si le poisson le plus proche est un requin (autrement dit : tout poisson "fuit" les requins, tout poisson est attiré par les thons).
On dira que deux poissons sont "voisins" si la distance entre eux est inférieure à 2. Chaque fois qu'un requin est "voisin" d'un thon, il le mange et le thon disparaît. Chaque fois que deux thons sont "voisins" l'un de l'autre, ils se reproduisent et un nouveau thon est ajouté à une position aléatoire de l'aquarium entre $(0, 0)$ et $(499, 499)$.
On veut gérer ces cas d'apparition et de disparition de poissons en créant une nouvelle liste de poissons. Ceci sera fait par la méthode `faireUnPas()` dont voici la description : Elle commence par mettre à jour les positions de tous les poissons en appelant la méthode `bougeTousPoissons()` et elle crée un double L2 de cette liste de poissons. Puis elle parcourt la liste originale, et pour chaque poisson de cette liste et son poisson le plus proche (au sens de la question précédente) elle applique les règles d'ajout et de suppression décrites ci-dessous, et fait la modification dans la nouvelle liste. Lorsqu'elle a parcouru toute la liste elle renvoie la nouvelle `PoissonList` obtenue ainsi.
Règles d'ajout et de suppression :

- S'il s'agit de deux thons, on ajoute un nouveau thon dans la nouvelle liste de poissons.
- S'il s'agit d'un thon et d'un requin, on supprime le thon dans la nouvelle liste de poissons.
- S'il s'agit de deux requins, il ne se passe rien.
- S'il ne s'agit ni de thon ni de requin, on lève une instance de la classe `PoissonInconnuException` qui sera traitée dans le main.

Q 123.4.5 Toujours dans la classe `PoissonList`, complétez la méthode ci-dessous.

```
1  public PoissonList faireUnPas() throws PoissonInconnuException {
2    bougeTousPoissons();
3    // creation d'un double de this :
4    // parcours de this :
5    for (int i=0;i<size();i++) {
6      // on recupere le poisson courant et son plus proche dans l'aquarium :
7      // traitement du couple si les deux poissons sont differents :
```

Q 123.5 Classes `Aquarium` et `TestAquarium`

Q 123.5.1 Définissez une classe `Aquarium` qui contient un attribut liste de type `PoissonList` représentant la liste des poissons présents dans l'aquarium. Écrivez-y le constructeur `Aquarium` qui prend en argument deux

entiers, `nbThons` et `nbRequins`, représentant le nombre initial de thons et de requins dans la simulation. Le constructeur remplit la liste de poissons avec le nombre adéquat de thons et de requins. Écrivez-y aussi la méthode `toString()` qui rend la liste des poissons contenus dans l'aquarium.

Q 123.5.2 Ecrivez dans une classe `TestAquarium` la méthode principale, `public static void main(String [] args)`, qui récupère en arguments de la ligne de commande le nombre de thons et le nombre de requins, appelle le constructeur de `Aquarium` avec ces valeurs, affiche la liste des poissons avec leurs coordonnées, appelle la méthode `faireUnPas` et réaffiche la liste des poissons. Pensez à attraper les exceptions qui sont susceptibles d'être levées et à les traiter en affichant un message idoine.
Rappels de cours :

- les arguments passés sur la ligne de commande sont récupérables par le tableau de chaînes de caractères args, paramètre de la méthode main.
- la méthode `int Integer.parseInt(String s)` rend l'entier représenté par la chaîne `s`, ou bien lève une exception `NumberFormatException` si la chaîne `s` ne repésente pas un entier.

Voici deux exemples d'exécution possibles :

```
>java TestAquarium 5 6m
  donnee non entiere
>java TestAquarium 3 2
  la liste des poissons :
  [thon(474,286) , thon(30,301) , thon(27,417) , requin(181,127) , requin(98,400)]
  liste des poissons apres un pas:
  [thon(471,277) , thon(43,305) , thon(25,411) , requin(112,216) , requin(61,405)]
```

Q 123.5.3 Dans la classe `Aquarium`, ajoutez une méthode `void run()` qui réalise une boucle perpétuelle dans laquelle on met sans cesse à jour la liste des poissons (méthode `faireUnPas()`)avec une temporisation de 300 ms (utilisez la méthode `sleep` de la classe `Thread`) et on affiche le nombre total de poissons ainsi que le nombre de thons et de requins. On traitera dans cette méthode les exceptions éventuellement levées.
Rappel de cours : la méthode `static void sleep(long millis)` de la classe `Thread` suspend l'exécution pendant millis millisecondes et peut lever l'exception `InterruptedException`, classe dérivée de `Exception`.