Corrigé

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

**Li230**
**Éléments de programmation par objets avec Java**

# Examen du 21 décembre 2007

durée : 2h
carte de référence seule autorisée
le barême sur 41 est indicatif

## A Problème Aquarium (28 points)

L'objet de ce problème consiste à réaliser un aquarium virtuel de taille 500x500 dans lequel évoluent des thons et des requins. On suppose que l'on dispose de la classe `Point` ci-dessous :

```
class Point {
      public int x,y;

      public Point(int x, int y){
         this.x=x;this.y=y;
      }
      public Point() {
     /* initialise x et y entre 0 et 499 */
           this((int)(Math.random() * 500),
                (int)(Math.random() * 500));
      }
      public String toString() {
        return "("+x+","+y+") ";
      }
      public double distanceTo(Point p) {
          int dx = p.x-x;
          int dy = p.y-y;
          return Math.sqrt(dx*dx+dy*dy);
      }
}
```

On donne aussi un extrait de la documentation java sur la classe `java.util.Vector`. Un `Vector` est une sorte de tableau **d'objets** dont la taille croît suivant les besoins de façon transparente pour l'utilisateur.

- `Vector()` construit un vecteur vide
- `Vector(Vector v)` crée un vecteur identique à celui passé en paramètre (c'est un constructeur de copie)
- `public int size()` renvoie le nombre d'objets présents dans le vecteur
- `public String toString()` rend la liste des éléments de ce vecteur, éléments auquels a été appliquée la méthode `toString()`
- `public void add(Object o)` ajoute `o` à la suite des objets du vecteur
- `public void remove(Object o)` supprime l'objet `o` des objets du vecteur
- `public void remove(int index)` supprime l'objet d'indice `index` du vecteur
- `public Object get(int index)` renvoie l'objet d'indice `index`

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

## I. Classe Poisson

1. (2 points) 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.

```
abstract class Fish {
      protected Point position;

      public Fish() {
         position = new Point();
      }
      public abstract void move(Point target);
      public Point getPosition() {
          return position;
      }
 ........
```

2. (1 point) 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.

```
public void checkBorders() {
   position.x %= 500;
   position.y %= 500;
}
```

3. (1 point) Définissez une autre classe `PoissonInconnuException` qui étend `Exception` et servira à gérer les cas où un poisson n'est pas d'un type connu.

```
class PoissonInconnuException extends Exception {
    PoissonInconnuException(String s) {super(s);
}
```

## II. Classe Requin

1. (2 points) 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 de 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 en I.2. `verifPosition()`.

```
class Shark extends Fish {
    public Shark() {
        super();
     }
    public void move(Point target) {
       position = new Point((position.x + target.x)/2,
                            (position.y + target.y)/2);
       checkBorders();
    }
    public String toString() {
       return "requin" + getPosition();
    }
  ..............
```

On peut aussi modifier directement la position sans en créer une nouvelle :

```
public void move(Point target) {
     position.x = (position.x + target.x)/2;
     position.y = (position.y + target.y)/2;
    checkBorders();
}
```

## III. Classe Thon

1. (2,5 points) 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)`".

```
class Tuna extends Fish {
   public Tuna() {
   super();
   }
   public void move(Point target) {
    double d = position.distanceTo(target);
    if (d>60) {
     position=new Point(position.x + (int)(1+Math.random()*30)-15,
                        position.y + (int)(1+Math.random()*30)-15);
    }
    else {
     position=new Point((position.x+target.x)/2,
                        (position.y+target.y)/2);
    }
    checkBorders();
    }
   public String toString() {
     return "thon"+this.getPosition();
    }
}
```

On peut aussi modifier directement la position sans en créer une nouvelle.

**IV. Classe PoissonList**

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

1. (2 points) Définissez la classe `PoissonList` qui hérite de `Vector`, avec un constructeur sans paramètres et un constructeur de copie, comme dans la classe `Vector` dont elle hérite.

```
import java.util.*;
class FishList extends Vector {
    public FishList() {
      super();
    }
    public FishList(FishList f) {
      super(f);
    }
.............
```

Suivant la version de java (1.4 ou 1.5 et 1.6) on peut aussi mettre :

```
class FishList extends Vector<Fish> {
```

2. (1,5 points) 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`.

```
int nbThons() {
    int k=0;

    for (int i=0; i<size(); i++) {
      if (get(i) instanceof Tuna) {
        k++;
      }
    }
    return k;
}
```

3. (3 points) 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.

```
public int getClosestFish(int index) {
    int retour = 0;
    Fish current = ((Fish)get(index));
    double dmin = Double.MAX_VALUE; // ou 1000, ça suffit
    for (int i=0;i<size();i++) {
      if (i!=index) {
        Fish autre = ((Fish)get(i));
        double d =
                current.getPosition().distanceTo(autre.getPosition());
        if (d<dmin) {
          dmin = d;
          retour = i;
        }
       }
    }
    return retour;
}
```

4. (2 points) 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).

```
public void moveAllFishes() {
    for (int i=0;i<size();i++) {
      Fish f1 = ((Fish)get(i));
      int closest = getClosestFish(i);
      Fish f2 = ((Fish)get(closest));
      Point target = new Point(250,250);
      if (f2 instanceof Tuna) {
        target = f2.getPosition();
      }
      f1.move(target);
    }
}
```

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 `this` 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 IV.3.) 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 :

a) S'il s'agit de deux thons, on ajoute un nouveau thon dans la nouvelle liste de poissons.
b) S'il s'agit d'un thon et d'un requin, on supprime le thon dans la nouvelle liste de poissons.
c) S'il s'agit de deux requins, il ne se passe rien.
d) 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.

5. (4 points) Toujours dans la classe PoissonList, complétez la méthode ci-dessous `PoissonList faireUnPas()`.

```
public PoissonList faireUnPas() throws PoissonInconnuException {
     bougeTousPoissons();
      // creation d'un double de this :




   FishList v2 = new FishList(this);


      // parcours de this :

      for (int i=0;i<size();i++){
     // on recupere le poisson courant et son plus proche dans l'aquarium :









      Fish f1 = ((Fish)get(i));
      int closest = getClosestFish(i);



       // traitement du couple si les deux poissons sont différents :
      if (closest>i) { // pour ne traiter q'une fois le couple
        Fish f2 = ((Fish)get(closest));
        double d = (f1.getPosition()).distanceTo(f2.getPosition());
        if (d>2) continue;
        if (f1 instanceof Tuna && f2 instanceof Tuna) {// thon et thon
          v2.add(new Tuna());
          continue;
        }
        if (f1 instanceof Tuna && f2 instanceof Shark) {//thon et requin
             v2.remove(i);
             continue;
        }
       if (f1 instanceof Shark && f2 instanceof Tuna){ // requin et thon
          v2.remove(closest);
          continue;
        }
          if (f1 instanceof Shark && f2 instanceof Shark){ //requin requin
            continue;
        }
        throw new PoissonInconnuException("poisson inconnu");
      }
    }
    return v2;
}
```

## V. Classes Aquarium et TestAquarium

1. (2 points) 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.

```
class Aquarium {
   FishList list;

    Aquarium(int nbtunas, int nbsharks) {
       list = new FishList();
       for (int i=0;i<nbtunas;i++) {
          list.add(new Tuna());
       }
       for (int i=0;i<nbsharks;i++) {
          list.add(new Shark());
       }
    }
    public String toString() {
       return list.toString();
    }
   ..................
```

2. (3 points) 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)]
```

```
class TestAquarium {
  public static void main(String args[])  {
    Aquarium m=null;;
    try {
       int nbtunas = Integer.parseInt(args[0]);
       int nbsharks = Integer.parseInt(args[1]);
       m= new Aquarium(nbtunas,nbsharks);
       System.out.println("la liste des poissons : \n"+ m);
       m.list=m.list.makeStep();
      System.out.println("liste des poissons apres un pas:\n"+ m);
    }
    catch(NumberFormatException e) {
       System.out.println("donnee non entiere");
       System.exit(-1);
    }
    catch(PoissonInconnuException e) {
       System.out.println(e.getMessage());
    }
  }
}
```

3. (2 points) 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`.

```
public void run() {
    while (true) {
       try {
          list = list.makeStep();
          Thread.sleep(300);
       }
       catch(PoissonInconnuException e) {
          System.out.println(e.getMessage());
       }
       catch(InterruptedException e) {}
       System.out.println ("nb total de poissons : "+ list.size()+
                           ", nb de thons : " + list.nbThons());
    }
}
```

# B Problème Etudiants (13 points)

On veut écrire une classe `Etudiant` dont les instances décrivent un étudiant ayant un nom et une liste de notes entières (au maximum 5 notes) implantée par un tableau. On veut également gérer dans cette classe toutes les instances d'étudiants créées, et pour ce faire on utilisera une instance de la classe `Vector` (voir la documentation de cette classe au début du problème A)

*Rappel de cours* : toute instance de la classe `Exception` doit obligatoirement être attrapée ou signalée comme étant propagée par toute méthode susceptible de la lever.

1. (4 points) Écrire la classe `Etudiant` correspondant à la description ci-dessus avec un constructeur à un paramètre, le nom. La méthode `toString()` rend le nom de l'étudiant suivi de ses notes.

```
class Etudiant {
    static Vector vEtu=new Vector();;
    static int nbMaxNotes=5;
    String nom;
    int tabNotes[]=new int[nbMaxNotes];
    int nbNotes=0;

    Etudiant(String n) {
      nom=n;
      vEtu.add(this);
    }
    public String toString() {
      // rend le nom et les notes
      String s= "";
      for (int i=0; i<nbNotes; i++) {
        s += tabNotes[i] + " ";
      }
      return nom + " " + s ;
    }
    ..............
```

2. (3 points) Ajouter la méthode `void entrerNote(int note)` qui entre la note `note` dans la liste des notes de cet étudiant. Elle lèvera une exception `TabNotesPleinException` (à définir) dans le cas où le tableau de notes de cet étudiant serait plein. Cette exception sera capturée dans le `main`.

```
        void entrerNote(int note) throws TabNotesPleinException {
            if (nbNotes<nbMaxNotes) {
                tabNotes[nbNotes]=note;
                nbNotes++;
            }
            else {
              throw new TabNotesPleinException("le tableau de notes"+
                           de l'etudiant " + this.nom+ " est plein");
            }
        }
        Il faut évidemment aussi déclarer la classe :
        class TabNotesPleinException extends Exception {
            TabNotesPleinException(String s) {
              super(s);
            }
        }
```

En supposant que la classe qui contient le main s'appelle `TestEtudiants`, on veut passer sur la ligne de commande une liste d'étudiants avec leurs notes, par exemple :
`java TestEtudiants Anna 12 13 7 15 Tom Arthur 9 12 15 0 13 12 Karim 15 8 11 12 10 Melissa 12 6 18 10 12 6`
On supposera que chaque donnée est correcte (pas de mélange entre lettres et chiffres), et que la première donnée est un nom.
Ces données sont de deux types : chaîne de caractères et entier. On va utiliser le fait qu'un entier ne fait pas lever d'exception à la méthode `Integer.parseInt` alors qu'une chaîne de caractères lui fait lever l'exception `NumberFormatException`.
*Rappel* : 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.

3. (3 points) Ecrire le code du `main` qui récupère les données et affiche pour chacune "c'est une note" ou bien "c'est un nom" suivant le cas. On utilisera obligatoirement le mécanisme d'exception pour ce faire.

Voici une exécution possible :

```
>java TestEtudiants Anna 12 13 7 15 Tom Arthur 9 12 15 0 13 12

Anna c'est un nom,
12 c'est une note, 13 c'est une note, 7 c'est une note, 15 c'est
une note,
Tom c'est un nom,
Arthur c'est un nom,
9 c'est une note, 12 c'est une note, 15 c'est une note, 0 c'est
une note, 13 c'est une note, 12 c'est une note,
```

```
class TestEtudiants {
   public static void main(String args[]) {
     int note;

  // lecture des donnees sur ligne de commande :
     for (int i=0;i<args.length; i++) {
         try {
            note=Integer.parseInt(args[i]);
            System.out.print(args[i]+" c'est une note, ");
          }
          catch(NumberFormatException e) {
            System.out.println("\n" + args[i]+" c'est un nom, ");
         }
     }
   }
}
```

4. (3 points) Enrichir/modifier le code précédent pour qu'il traite les données de la façon suivante :
   - si c'est une chaîne de caractères, il crée une nouvelle instance d'étudiant portant ce nom.
   - si c'est une note, il ajoute cette note à la liste des notes de l'étudiant créé précédemment,

   puis affiche la liste des étudiants. On pensera à traiter les différentes exceptions levées (on rappelle qu'un étudiant a au maximum 5 notes).
   Voici une excécution possible :

```
>java TestEtudiants Anna 12 13 7 15 Tom Arthur 9 12 15 0 13 12 Karim 15 8 11 12 10
Melissa 12 6 18 10 12 6
le tableau de notes de l'etudiant Arthur est plein
le tableau de notes de l'etudiant Melissa est plein
les 5 etudiants :
[Anna 12 13 7 15 , Tom , Arthur 9 12 15 0 13 , Karim 15 8 11 12 10 , Melissa
12 6 18 10 12 ]
```

```
class TestEtudiants {
   public static void main(String args[]) {
     int note;
     Etudiant eCourant=null;

    // lecture et stockage des donnees de la ligne de commande :
     for (int i=0;i<args.length; i++) {
        try {
           note=Integer.parseInt(args[i]);
           eCourant.entrerNote(note);
         }
        catch(TabNotesPleinException e) {
          System.out.println(e.getMessage());
        }
        catch(NumberFormatException e) {
            eCourant = new Etudiant(args[i]);
        }
     }
     //affichage des etudiants :
     System.out.println("les "+ Etudiant.vEtu.size() +
                        " etudiants : \n" + Etudiant.vEtu);
  }
}
```