Grands Moulins

18/03/2023 - 16h00-18h00

Initiation à la Programmation (IP2)

Aucun document autorisé.
Dans vos réponses n'utilisez **aucune des librairies de java** qui sont en rapport avec les listes, vous devez écrire tout ce qui vous est utile.
Pensez à dire sans ambiguïté à quelle classe vos méthodes appartiennent : **ne comptez pas sur le correcteur pour deviner des choses**, c'est à vous d'être clair !

Ce sujet est composé de 3 exercices, ils sont indépendants. Le barème est indicatif, et vous donne une idée du temps à consacrer aux questions et leur difficulté. Si le barème évolue légèrement ce sera pour donner de l'importance aux exercices traités complètement et correctement : il vaut donc mieux s'appliquer à faire les choses bien plutôt que de se précipiter.

Exercice 1 (5 points) Les imbrications d'objets.

1. **(1 point)** On rappelle que la notation $[a, b]$ dénote l'intervalle des valeurs comprises entre a et b au sens large. Cette notation n'a de sens que lorsque a est plus petit ou égal à b. L'intervalle $[a, a]$ est valide : il ne contient que la valeur a. Lorsque deux intervalles $[a, b]$ et $[c, d]$ ont des valeurs en commun, tout l'intervalle correspondant à leur intersection est caractérisé par $[max(a, c), min(b, d)]$.

   Ecrivez dans une classe `Tools` une méthode statique `intersect`, publique qui prenne en argument des valeurs entières pour a, b, c, d et qui dise si oui ou non les deux intervalles $[a, b]$ et $[c, d]$ s'intersectent.[1]

2. Une classe pour les intervalles.

   (a) **(0.5 point)** Définissez une classe `Intervalle` destinée à représenter l'objet mathématique $[a, b]$ et écrivez un constructeur (**sans** vous préoccuper des anomalies éventuelles sur a et b),

   (b) **(0.5 point)** Mettez à disposition des utilisateurs de cette classe une méthode `boolean estVide()` qui permet de détecter les objets qu'on considérera être vides.

   (c) **(1 point)** Écrivez une méthode non statique qui permette de savoir si deux intervalles supposés non vides s'intersectent.

3. On défini une zone du plan comme étant un espace rectangulaire délimité par une position (x, y), désignant le point le plus en bas et à gauche, muni de deux longueurs horizontales et verticales partant de ce point et allant respectivement vers la droite et vers le haut.

   Pour savoir si deux zones sont en contact, on projette simplement les segments de leurs cotés horizontaux et verticaux sur les axes des x et des y. On vous affirme ici que s, sur ces deux axes, ces projections (qui sont des intervalles) s'intersectent alors c'est que les zones se touchent (et c'est équivalent).

   (a) **(1 point)** Définissez cette classe `Zone` **en utilisant des attributs de type `Intervalle`** et munissez là d'un constructeur naturel correspondant à la description faite dans l'introduction de cette question.

   (b) **(1 point)** Écrivez une méthode non statique `intersect` qui permette de savoir si deux zones sont en contact.

[1] Vous pouvez utiliser les méthodes Math.max et Math.min qui retournent respectivement les max et min entre 2 éléments.

Exercice 2 (7 points) On s'intéresse à une population d'individus représentés par la liste et les cellules suivantes :

```
public class Population {
  private Cellule first;
}
public class Cellule {
  private Personne p;
  private ZoneCouverte z;
  private Cellule suivant;
}
public class Personne {
  public final String name;
  private boolean malade;
  public boolean estMalade() {return malade;}
  public boolean contamine(Personne x) { ... }
  public static boolean contamination (Personne x, Personne y) {...}
}

```

L'idée générale de cet exercice est que si dans une population une personne malade est relativement proche d'une personne non malade alors elle l'infectera à coup sûr.

Nous n'écrirons pas de classe ZoneCouverte (mais elle pourrait par exemple ressembler à celle de l'exercice précédent). La seule chose utile à savoir ici est que cette classe est supposée disponible et qu'elle est munie d'une méthode non statique intersect permettant de savoir si oui ou non deux zones sont en contact : c'est cette condition qui déclenchera la transmission de la maladie entre les deux personnes auxquelles ont a associé ces zones dans leur cellule.

1. **(1 point)** Vous constatez que les attributs name et malade ont des signatures différentes. Nous voudrions que vous nous expliquiez clairement, en rédigeant quelques phrases et en donnant quelques arguments, pourquoi ces choix-là ont été fait.

2. **(1 point)** Dans la classe Personne on propose deux méthodes pour la contamination écrivez celle qui vous semble d'utilisation pertinente, et expliquez bien votre choix. (Si vous voulez faire d'autres changements : faites-les, pourvu que vous vous expliquiez)

3. **(1 point)** Si on voulait conserver ces deux méthodes (avec ces signatures) pourrait-on leur donner le même nom ? (Expliquez)

4. **(4 points)** On veut savoir comment va se développer la maladie et quelles sont les personnes qui seront malades ou non. Une méthode propagation() permettra d'obtenir l'état de santé final de la population. Pour le calculer vous aurez besoin d'écrire des méthodes auxiliaires.
   L'idée est simple : chaque personne sera présentée à toutes les autres pour un potentiel contact dans sa zone. Si une seule nouvelle contamination a lieu alors chaque personne sera à nouveau présentée à toutes les autres et ainsi de suite jusqu'à ce que la situation soit stable.

   Vous aurez probablement à écrire les méthodes suivantes (en précisant dans quelles classes) :

   - private boolean PresenteATous(Personne other, ZoneCouverte z)
   - ??? propagation()

   Si vous avez une stratégie qui vous convient mieux, commencez par l'expliquer en français avant d'écrire votre code afin de vous assurer qu'il soit bien compréhensible par le correcteur.

Exercice 3 (9 points)

On s'intéresse à une file de voiture qui circule sur une seule voie, toutes dans le même sens, elles seront toujours placées l'une derrière l'autre. Chaque voiture peut voir quelle est celle qui la précède et quelle est celle qui la suit ; elle connaît également la distance qui la sépare de la voiture qui est devant elle.

Par convention la voiture de tête verra une distance devant elle fixée à 0. Dans les autres cas, si une voiture voit celle de devant elle à une distance 0 cela signifie qu'elles sont très proches mais il n'y a pas d'accident et elles restent bien à la queue leu-leu dans cet ordre. Voici un affichage décrivant une situation avec des voitures dont les plaques sont "a", ..."e", qui sont très proches mais dans cet ordre : "a" devant "b" etc ...

```
<-- (0 a) <-- (0 b) <-- (0 c) <-- (0 d) <-- (0 e)
```

Notez que si a est associée à la valeur qu'elle voit devant elle de 0 c'est à cause de la convention qu'il n'y a personne devant la voiture de tête ; et que b, c, d et e voient celle qui la précède à distance 0 car b est très proche de a, que c est très proche de b etc.

Nous allons faire un peu bouger les voitures pour bien comprendre cette modélisation.
Si "a" avance de 3 unités, on obtient :

```
<-- (0 a) <-- (3 b) <-- (0 c) <-- (0 d) <-- (0 e)
```

En effet, la voiture de tête ne voit toujours personne devant elle, mais a pris ses distances avec b, et pour les autres rien ne change.
Si à présent "b" avance de 2 unités on obtient :

```
<-- (0 a) <-- (1 b) <-- (2 c) <-- (0 d) <-- (0 e)
```

On observe que "b" se rapproche de "a" et s'éloigne de "c".
Si elle avance encore un peu (d'un) elle rejoint "a" tout en restant derrière elle :

```
<-- (0 a) <-- (0 b) <-- (3 c) <-- (0 d) <-- (0 e)
```

Ici "c","d" et "e" sont quasiment au même niveau, car "e" voit "d" à distance 0, et "d" voit "c" à distance 0 et elles sont dans l'ordre "c" puis "d" puis "e".
Si "e" avance d'un pas : elle passera devant "c", s'en éloignera et s'approchera de b :

```
<-- (0 a) <-- (0 b) <-- (2 e) <-- (1 c) <-- (0 d)
```

Pour finir, si "e" avance encore de 3 unités elle passera en tête :

```
<-- (0 e) <-- (1 a) <-- (0 b) <-- (3 c) <-- (0 d)
```

Voici les classes qui permettent cette modélisation :

```
public class Voie {
  private Voiture first;
}
public class Voiture {
  private final String plaque;
  private int distVideDevant;
  public Voiture derriere;
  public Voiture devant;
  public Voiture (String s) {plaque =s;}
}
```

Dans la situation finale précédente, on a affiché une voie où first serait la voiture "e" ayant **derrière** elle "a" ayant **devant** elle "e" etc ... Remarquez qu'on a choisi de ne pas afficher tous les liens, mais seulement ceux qui permettent de comprendre la situation.

Dans la suite nous allons décomposer ces mouvements, et écrire quelques méthodes utilitaires.

1. **(1 point)** Que fait le constructeur par défaut de Voie ? Peut-on parler de constructeur par défaut pour Voiture ? (Rédigez votre réponse)

2. **(1 point)** Ecrivez la/les méthodes affiche() qui produirait **exactement** les même affichages que ceux que nous avons présentés.

3. **(1 point)** Ecrivez la méthode public void ajouteTete(String x) dans la classe Voie qui ajoute en premier élément de la liste une voiture de plaque x [2]

4. **(1 point)** Ecrivez la méthode private void updateDist() de la classe Voiture qui présuppose [3] que le véhicule courant restera bien à sa place dans la file sans doubler personne, et qui met à jour les distances concernées par un changement lorsque la voiture avance d'**un seul** pas.

5. **(2 points)** Ecrivez la méthode private void pass() qui s'adresse à une voiture et qui réorganise la séquence pour faire en sorte que la voiture concernée se retrouve ensuite devant toutes celles avec qui elle était à distance 0.

6. **(2 points)** Ecrivez la méthode public void avanceOne(String id) à la fois dans Voie et Voiture dont le rôle est de faire en sorte que le véhicule de plaque id avance d'une unité.

7. **(1 point facile)** Ecrivez la méthode public void avance (int d, String id) qui fait avancer la voiture id de d unités.

[2] nous supposerons sans avoir à le vérifier que les plaques utilisées sont toujours distinctes
[3] pas besoin de le vérifier donc