Université Paris-Diderot Paris
UFR d'informatique
L3 Licence d'informatique

Examen de Compléments de Programmation Orientée Objet

(première session 2015-2016)
Durée deux heures ; tous les documents sont interdits
Le barème est donné à titre indicatif : total 30
3 pages

Exercice 1 : Une *File* est une strcuture de données (FIFO : First In First Out) contenant deux méthodes : *mettre* et *prendre* : pour un objet `o`, `mettre(o)` insère `o` dans la file et `prendre()` retire l'objet le plus anciennement inséré dans la file. Par exemple, après « `mettre(a);mettre(b)` », `prendre()` retournera `a`, un nouveau `prendre()` retournera `b` et le contenu de la file sera le même qu'avant la séquence « `mettre(a);mettre(b)` ».

La file contiendra aussi une méthode `estPleine` (la file est pleine) et une méthode `estVide` (la file est vide) retrournant des booléens.

De plus `mettre(o)` retournera l'objet `o` dans le cas où l'opération a réussi (la file n'est pas pleine) et `null` sinon (la file est pleine). De même, `prendre()` retournera `null` si la file est vide.

1. (a) Ecrire une interface `File` correspondant à cette structure de données. L'interface (et les implémentations dans les questions suivantes) contiendra aussi une méthode `versTab` qui retourne un tableau contenant tous les éléments contenus dans la File. (1pt)

   (b) En utilisant comme classe *interne statique* la classe ci-dessous :

   ```
   class Noeud{
       Object val;
       Noeud suivant;
       Noeud(Object o){val=o;suivant=null;}
       Noeud(Object o, Noeud i){this(o);if(i!=null)i.suivant=this;}
   }
   ```

   définir une implémentation `FileList` de `File`. (On prendra soin à ce que `mettre` et `prendre` se fassent en temps constant. On pourra pour cela avoir deux variables `tete` et `queue` qui feront références respectivement à la tête de la file où les nouveaux éléments seront mis dans et à la queue par où les éléments de la liste seront pris.) (3pts)

   (c) Définir une implémentation `FileTab` de `File` en stockant les éléments de la File dans un tableau de `Object`. La taille de ce tableau sera initialisée par un constructeur. (On prendra soin à ce que `mettre` et `prendre` se fassent en temps constant. Pour cela on pourra avoir deux variables : `tete` qui pointe sur l'élément du tableau dans lequel le prochain `mettre` écrira et `queue` qui pointe sur l'élément du tableau correspondant à l'objet qui sera retiré par le prochain `prendre`. Les mises à jour de `tete` et `queue` se faisant modulo la taille du tableau.) (3pts)

   (d) Si maintenant on veut que la `File` soit générique (paramatrée par un type `T`) quelles modifications doit-on faire (préciser ces modifications pour l'interface et les deux implémentations proposées précédemment). (2pts)

2. Dans la suite on va utiliser les structures de données définies précédemment dans un contexte de concurrence (multi-thread).

   Un *producteur* est une thread qui utilise une instance de `File` et insère dans la `File` les entiers de 1 à 100. Un *consommateur* est une thread partageant cette instance de `File` qui extrait indéfiniment de la file les données et les écrit sur la sortie standard. Toutes les données des producteurs doivent être consommées pas les consommateurs.

(a) Ecrire une classe `Producteur` et une classe `Consommateur` pour réaliser les threads décitres ci-dessus. La référence sur la file utilisée par un producteur ou un consommateur sera initialisée par un constructeur. Pour produire (respectivement consommer) une valeur, le producteur (resp. le consommateur) fera une boucle while jusqu'à ce que la valeur soit mise dans la file (resp. prise de la file). (On rappelle que `mettre` et `prendre` retournent `null` en cas d'échec). (2pts)

(b) Ecrire un main qui instancie une file de type `FileTab` et lance une thread producteur et une thread consommateur partageant cette instance de file. Si vous exécutez ce main, peut-on garantir que la sortie standard contiendra dans l'ordre tous les entiers entre 1 et 100 ? tous les entiers entre 1 et 100 (non nécessairement dans le même ordre) ? Justifiez votre réponse. (3pts)

(c) En utilisant des verrous appropriés (`synchronized`) écrire une classe `Buffer` dérivée de `FileList` qui permet d'assurer que les accès à la file se font de façon séquentielle sans interférence entre les différentes thread partageant la file.

Procéder de même pour une classe `BufferBorne` dérivée de `FileTab`. (3pts)

(d) On suppose maintenant qu'il y a plusieurs producteurs et plusieurs consommateurs partageant la même file qui sera un `Buffer`. Quel est, du point de vue des performances l'inconvénient de la solution proposée dans la question précédente ?

Pour éviter ces inconvénients, modifier la classe `Buffer` (on pourra utiliser `wait` et `notifyAll`).

Même question concernant la classe `BufferBorne`. (4pts)

Exercice 2 : Rappelons que dans l'interface `Stream<T>` :

- `Stream<T> distinct()` : supprime les doublons du *stream*.
- `Stream<T> filter(Predicate<? super T> p)` : retourne le *stream* où l'on ne conserve que les éléments satisfaisant p.
- `void forEach(Consumer<? super T> action)` : applique l'action à chaque élément.
- `boolean anyMatch(Predicate<? super T> p)` : retourne vrai si et seulement si `p` est vrai pour au moins un élément du *stream*.
- `<R,A> R collect(Collector<? super T,A,R> collector)` : opère la réduction (mutable) définie par l'objet `collector`.
- `long count()` : retourne le nombre d'éléments dans le stream courant.
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)` : retourne le *stream* des éléments obtenus en appliquant la fonction `mapper` à chaque élément du stream courant.
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R» mapper)` : retourne le *stream* consistant en la concaténation des *streams* obtenus en appliquant la fonction `mapper` à chaque élément du stream courant.

Dans la classe `Collectors` :

- `static <T> Collector<T, ?, Set<T» toSet()` : retourne un objet définissant une réduction (mutable) d'un *stream* vers une collection de type `Set` (sans doublon).

1. Avec la classe `Habitant` :

```
class Habitant {
    public int age;
    // ...
    public int getAge() {
        return age;
    }
    //...
}
```

Expliquez ce qu'affiche le programme ci-dessous, puis écrivez le bloc `for` comme un *pipeline* d'opérations d'agrégation utilisant des lambda-expressions. (3pts)

```
int nbMineurs = 0;
```

```
for (Habitant h : population) {
    if (h.age < 18) nbMineurs++;
}
System.out.println(nbMineurs);
```

2. Avec la classe `Livre` :

```
class Livre{
    public Set<String> themes;
    //...
    public String auteur;
    //...
}
```

et les variables `bibli` et `themeDeBase` respectivement de type `Set<Livre>` et de type `String`, expliquez ce que fait le code suivant, puis convertissez-le en une ou des instruction(s) utilisant les lambda-expressions et les opérations d'agrégation au lieu des boucles `for`. (3pts)

```
Set<String> auteurs = new HashSet<>();
for (Livre l : bibli) {
      boolean concerneCeTheme = false;
      for (String t : l.themes) {
          if (t.equals(themeDeBase)) {
             concerneCeTheme = true;
             break;
          }
      }
      if (concerneCeTheme) auteurs.add(l.auteur);
}
for (String a : auteurs) System.out.println(a);
```

3. Parmi les *pipelines* suivants, lesquels garantissent-ils la sortie "`35214`" ? (3pts)
   (a) `Arrays.asList(3,5,2,1,4).stream().forEach(System.out::print);`
   (b) `Arrays.asList(3,5,2,1,4).parallelStream().forEach(System.out::print);`
   (c) `Arrays.asList(3,5,2,1,4).stream().parallel().forEach(System.out::print);`
   (d) `Arrays.asList(3,5,2,1,4).parallelStream().sequential().forEach(System.out::print);`
   (e) `Stream.of(3,5,2,1,4).forEach(System.out::print);`
   (f) `Arrays.asList(3,5,2,1,4).parallelStream().forEachOrdered(System.out::print);`