Cours de programmation Java (PROG2)

Magistère MIT, 1re année
Examen terminal, 19 avril 2013

Les notes personnelles rédigées en cours, les transparents du cours et les autres documents sont autorisés.

Les trois problèmes sont indépendants. Tenez compte des indications de barème pour répartir votre temps entre eux.

Dans vos réponses aux questions vous demandant d'écrire du code, vous pourrez prendre quelques libertés avec la syntaxe du langage. En revanche, vous apporterez un soin particulier à l'orthographe et à la présentation de votre copie.

Merci de bien vous identifier sur chacune des copies, et de les numéroter.

1 *Visitor* revisité (7,5 points)

Un programmeur facétieux nous a fourni une bibliothèque dont les interfaces sont parfois surprenantes. Par exemple, certaines méthodes renvoient des données organisées en liste dont l'interface suit.

```
public interface List<E> {
  public <F> F fold(ListVisitor<E,F> v);
}
public interface ListVisitor<E,F> {
  public F visitEmpty();
  public F visitCons(E elem, F accum);
}
```

Par ailleurs, les classes satisfaisant cette interface, et qui permettent de construire des listes, ne sont guère plus riches.

```
public final class EmptyList<E> implements List<E> {
  public EmptyList();
  public <F> F fold(ListVisitor<E,F> v);
}
public final class ConsList<E> implements List<E> {
  public ConsList(E head, List<E> tail);
  public <F> F fold(ListVisitor<E,F> v);
}
```

Enfin, la documentation dit que la méthode `fold` permet de visiter un à un et dans l'ordre tous les éléments de la liste, du plus ancien (la queue) au plus récent (la tête). Lors d'un appel à `fold`, si la liste est vide, c'est `visitEmpty` qui est appelé ; si la liste est faite d'un élément `e` et d'une suite `q`, c'est `visitCons(e,a)` qui est appelé où `a`

est le résultat de la visite de `q`. De plus, si une exception est levée par une des deux méthodes du `ListVisitor`, le comportement de `fold` n'est pas documenté.

Voici par exemple comment récupérer l'élément en tête de liste. Il suffit d'appeler la méthode `fold` avec l'instance suivante de `ListVisitor`.

```
static <E> ListVisitor<E,E> end() = new ListVisitor<E,E>() {
  public E visitEmpty() { return null; }
  public E visitCons(E elem, E accum) { return elem; }
};
```

1.1 Premiers visiteurs

a) Donnez une instance de `ListVisitor<E, Integer>` pour mesurer la longueur d'une liste d'éléments de type `E`.

b) Donnez une instance de `ListVisitor<E, String>` pour construire une représentation sous forme de chaine de caractères d'une liste d'éléments de type `E`.

c) Donnez une fonction `append` qui pour toute liste de type `List<E>` construit une instance de `ListVisitor<E, List<E>>` pour concaténer cette liste à la suite de toute autre liste. Par exemple, si `a` est la liste `(1, 2, 3)` et `b` la liste `(4, 5, 6)`, alors l'appel `a.fold(append(b))` doit renvoyer la liste `(1, 2, 3, 4, 5, 6)`. Voici une signature possible pour votre solution.

```
static <E> ListVisitor<E,List<E>> append(final List<E> l);
```

1.2 Visiteurs fonctionnels

Pour la suite de l'exercice, nous considérons l'interface suivante. Elle nous permet de représenter les fonctions (de domaine `E` et codomaine `F`) sous forme d'objets.

```
public interface Fun<E,F> {
  public F apply(E arg);
}
```

a) Écrivez une fonction `map` qui, étant donnée une fonction `f`, construit une instance de `ListVisitor` qui permet d'appliquer la fonction `f` à tous les éléments d'une liste (en construisant une nouvelle liste). Voici une signature possible pour votre solution.

```
static <E,F> ListVisitor<E,List<F>> map(final Fun<E,F> f);
```

Expliquez comment alors réaliser une copie (superficielle [1]) d'une liste.

b) Écrivez une fonction `filter` qui, employée avec `fold`, extrait d'une liste (de type `List<E>`) la liste des éléments qui satisfont un prédicat de type `Fun<E, Boolean>`.

1. La copie contient les mêmes éléments que la liste originale. Une copie *profonde* contiendrait une copie des éléments le la liste originale.

1.3 Visiteur à l'envers Il semble parfois plus judicieux de parcourir les éléments d'une liste depuis la tête jusqu'à la queue (c'est-à-dire dans l'ordre inverse à celui fourni par `fold`). On suppose alors que l'on a une fonction `foldl`, dont la signature suit, qui nous fournit ce parcours.

```
static <E,F> F foldl(List<E> l, ListVisitor<E,F> visitor);
```

a) En utilisant des fonctions déjà vues dans cet exercice, montrez comment récupérer le dernier élément d'une liste, puis comment renverser une liste (c'est-à-dire, étant donnée une liste, construire la liste qui contient exactement les mêmes éléments, mais dans l'ordre inverse).

b) Programmez `foldl`. Indice : continuations.

2 Threads (7,5 points)

2.1 Tri parallèle Proposez un algorithme de tri parallèle d'un tableau (`array`) d'entiers utilisant les `Thread`s java. Discuter la complexité de l'algorithme proposé.

2.2 Deadlocks Dans le code suivant, est-ce qu'il y a des scénarios qui peuvent conduire à un interblocage (*deadlock*) ? Si oui, donnez un exemple et proposez une solution pour les éviter.

```
final class Account {

  double balance;
  int id;
  Lock l = new ReentrantLock();

  void withdraw(double amount){ balance -= amount; }

  void deposit(double amount){ balance += amount; }

  static void transfer(Account from, Account to, double amount){
    from.l.lock();
    to.l.lock();
    from.withdraw(amount);
    to.deposit(amount);
    to.l.unlock();
    from.l.unlock();
  }
}
```

3 Gestion d'erreurs par exceptions (5 points)

3.1 Les exceptions du `Scanner` Pour lire des données dans un fichier on peut utiliser le code suivant.

```
public static void main(String[] arg) {
  Scanner sc = null;
  try {
    sc = new Scanner( new File("sequence_ordres") );
  } catch (java.io.FileNotFoundException e) {
    System.out.println(e);
  }

  // est-ce qu'il y a un nouvel élément à analyser ?
  while (sc.hasNextInt()) {
    // on récupère l'élément suivant
    System.out.println(sc.nextInt());
  }
}
```

On constate dans la Javadoc concernant la méthode `nextInt` de la classe `Scanner` qu'elle lance trois exceptions :

- `InputMismatchException` : si l'élément suivant ne correspond pas à un entier, ou dépasse sa capacité ;
- `NoSuchElementException` : si l'entrée est épuisée ;
- `IllegalStateException` : si le scanner a été fermé.

On y lit aussi que ces trois exceptions dérivent de `RuntimeException` qui elle-même dérive de `Exception`.

Pourquoi le programme ci-dessus était-il alors correct sans se préoccuper de ces trois exceptions ?

3.2 Un simple système d'achat en ligne On imagine le système d'achat en ligne suivant, avec vérification de l'âge légal au moment du paiement (voir figure 1 page suivante).

Que se passe-t-il si l'utilisateur écrit n'importe quoi, par exemple « bonjour » au moment où on lui demande son âge ?

3.3 Détection de l'âge d'un mineur On désire remonter le problème d'un achat par un utilisateur d'un âge inférieur à 18 (ans). Dans ce but :

a) Programmez une classe d'exception spécifique nommée `Exception18Ans`.

b) Programmez la méthode `verifierAge` pour signaler ce cas problématique au reste du programme plutôt que le simple affichage écran initial.

```java
public class SystemeAchat {
  final private Scanner s;

  public SystemeAchat() {
    // pour lire les caractères sur l'entrée standard (clavier)
    s = new Scanner(System.in);
  }

  public void verifierAge() {
    int age = s.nextInt();
    if (age < 18) {
      System.out.println("Age inférieur à 18 ans");
    }
  }

  // fait toutes les vérifications nécessaires
  public void verifications() {
    //...
    verifierAge();
    //...
  }

  public static void main(String[] arg) {
    SystemeAchat sys = new SystemeAchat();
    // achats sélectionnés par l'utilisateur...
    sys.verifications();
    // finalisation des achats...
  }
}
```

FIGURE 1 : Hypothétique système d'achat en ligne