Programmation et Langages de Programmation

Licence Informatique, mai 2001

Tout document autorisé, durée 4 heures

Justifier clairement et succinctement vos choix, et décrivez brièvement vos algorithmes lorsque vous le jugez nécessaire.

Toute réponse difficile à comprendre sera considérée comme fausse.

Vous pouvez rajouter du code (méthode, donnée ou classe) non demandé à chaque fois que vous le jugerez nécessaire. Ecrivez entièrement ce code, en particulier dans le cas d'exceptions, de classes abstraites ou d'interfaces.

Vous pouvez toujours considérer une classe ou une méthode demandée dans une question précédente comme disponible, même si vous n'avez pas traité cette question. Il vous est également conseillé de lire le sujet entièrement avant de commencer votre travail.

Si vous souhaitez utiliser une classe (ou une méthode particulière) de l'API Java dont vous ne vous souvenez pas du nom, donnez-lui un nom explicite et précisez-le clairement sur votre copie.

Ce sujet porte sur les arbres enracinés. Il est important de noter qu'une instance d'une classe d'arbre enraciné désigne aussi bien un noeud que l'arbre dont il est racine.

Partie 1 - Arbres Fonctionnels

On considère la classe `FArbre`, qui représente, dans un style de programmation fonctionnelle, des arbres enracinés (arborescences) où chaque noeud possède un nombre quelconque de fils.

```
public class FArbre {
    private Object data;
    private FArbre [] fils;

    /** Crée un arbre en spécifiant la donnée associée à la racine et
        ses fils. Pour un nombre de fils nul, le tableau des fils doit
        être de longueur 0, mais ne peut être null. */
    public FArbre(Object data, FArbre [] fils) {
        this.data = data;
```

```
        this.fils = new FArbre[fils.length];
        System.arraycopy(fils, 0, this.fils, 0, fils.length);
    }

    /** Crée un arbre limité à un seul noeud. */
    public FArbre(Object data) {
        this(data, new FArbre[0]);
    }

    /** Crée un itérateur sur les fils. */
    public Iterator fils() {
        /* La méthode de classe Arrays.asList(Object []) renvoie une
           instance de java.util.List dont les éléments sont ceux du
           tableau passé en paramètre. */
        return Arrays.asList(fils).iterator();
    }
    ...
}
```

Question 1 Rajouter les méthodes suivantes :

1. `public FArbre fils(int i)` qui renvoie le ième fils du noeud,
2. `public int nfils()` qui renvoie le nombre de fils du noeud,
3. `public Object data()` qui renvoie la donnée associée au noeud,
4. `public boolean estUneFeuille()` qui précise si un noeud est une feuille, c'est à dire sans fils.

Question 2 Rajouter les méthodes

1. `public int ordre()`
2. `public int hauteur()`

qui renvoient respectivement le nombre de noeuds d'un arbre A et sa hauteur en utilisant les formules de récurence suivantes :
$ordre(A) = 1 + \sum_{B \in fils(A)} ordre(B)$
$hauteur(A) = 0$ si A est une feuille, $1 + \max_{B \in fils(A)} hauteur(B)$ sinon.

Question 3 On considère l'interface suivante :

```
public interface Action {
    public void f(Object o);
}
```

Rajouter à `FArbre` la méthode `public void parcours(Action a);`
qui parcourt l'arbre récursivement en appliquant la méthode `a.f(Object)` d'abord à la donnée de la racine, puis à celles des noeuds de chacun des sous-arbres.
Utiliser `parcours` pour donner une nouvelle version de la méthode `ordre()`.

Partie 2 - Arbres Impératifs

On considère la classe `ArbreImp` qui représente, dans un style de programmation impérative, des arbres enracinés où chaque noeud possède un nombre quelconque de fils. Les premières méthodes de cette classe vous sont données ci-dessous.

```
public class ArbreImp {
    private Object data;
    private List fils;
    private ArbreImp pere;

    public ArbreImp(Object data) {
        this.data = data;
        fils = new ArrayList();
        pere = null;
    }

    public ArbreImp(Object data, ArbreImp [] fils) {
        this(data);
        for (int i = 0; i < fils.length; i++) {
            ajouterFils(fils[i]);
        }
    }

    public void ajouterFils(ArbreImp f) {
        if (f.pere != null) {
            throw new IllegalArgumentException(
               "Un arbre ne peut etre qu'une fois sous-arbre");
        }
        fils.add(f);
        f.pere = this;
    }

    public ArbreImp pere() {
        return pere;
```

```
    }
    ...
}
```

Question 1 Que faut-il rajouter pour qu'un client puisse écrire des algorithmes fonctionnant indifféremment avec des arbres impératifs ou fonctionnels à chaque fois que cela est raisonnable ?

Question 2 Compléter le code de la classe `ArbreImp` en respectant l'objectif de la question 1. La classe `ArbreImp` disposera en particulier d'une méthode `public void changerData(Object data)` qui modifie la valeur de la variable `data`.

Partie 3 - Arbres Syntaxiques

On souhaite représenter l'arbre syntaxique d'une expression en utilisant la classe `ArbreImp`. Ces expressions sont préfixées, et composées d'opérateurs sur les entiers, et de variables à valeur entière.

Pour représenter les opérateurs, on se munit d'une classe abstraite `Opérateur` dont le code est donné ci-dessous :

```
public abstract class Operateur {
    private static Map ops = new HashMap();
    private String nom;
    private int arite;

    public Operateur(String nom, int arite)
        throws OperateurExistantException
    {
        Operateur op = ops.get(nom);
        if (op != null) {
            throw new OperateurExistantException(op);
        }
        this.nom = nom;
        this.arite = arite;
        ops.put(nom, this);
    }

    public String nom() {
        return nom;
```

```
    }

    public static Operateur operateur(String nom)
        throws OperateurInexistantException
    {
        Operateur op = ops.get(nom);
        if (op == null) {
            throw new OperateurInexistantException(nom);
        }
        return op;
    }

    public int arite() {
        return arite;
    }

    public abstract int evaluer(int [] operandes);
}
```

Question 1 Pour représenter les variables, on utilise une classe concrète `Variable` qui hérite de `Opérateur`, en considérant qu'une variable est un opérateur d'arité 0 dont l'évaluation est une valeur entière associée. Une instance de `Variable` est créée à partir d'un nom et d'une valeur entière initiale. La classe `Variable` possède en plus des méthodes héritées de `Opérateur` la méthode
`public void changerValeur(int val)` qui modifie la valeur associée à la variable.

Donner le code de la classe `Variable`.

Question 2 On dispose d'une classe `ArbreSyntaxique` qui fabrique à partir d'une expression préfixée l'arbre syntaxique correspondant. Le code vous est donné ci-dessous.

```
public class ArbreSyntaxique extends ArbreImp {
    private int valeur;
    private static int index;

    /* Crée l'arbre syntaxique correspondant à l'expression
       commençant à lexemes[index]. Le booléen passé en paramètre
       indique s'il est nécessaire de remettre la variable de classe
       index à 0. */
```

```java
    private ArbreSyntaxique(String [] lexemes, boolean debut)
        throws ErreurSyntaxiqueException
    {
        /* L'opérateur portant le nom du lexeme d'indice index sera
           stocké dans la variable data héritée de ArbreImpl, alors
           que la valeur issue de l'évaluation de l'expression
           représentée par l'arbre sera stockée dans la variable valeur */
        super(null);
        if (debut) {
            index = 0;
        }
        try {
            Operateur op = Operateur.operateur(lexemes[index++]);
            int n = op.arite();
            ArbreSyntaxique [] fils = new ArbreSyntaxique[n];
            int [] operandes = new int[n];

            changerData(op);
            for (int j = 0; j < n; j++) {
                fils[j] = new ArbreSyntaxique(lexemes, false);
                ajouterFils(fils[j]);
                operandes[j] = fils[j].valeur;
            }
            valeur = op.evaluer(operandes);
        } catch (OperateurInexistantException e) {
            throw new ErreurSyntaxiqueException(lexemes, index);
        }
    }

    public ArbreSyntaxique(String [] lexemes)
        throws ErreurSyntaxiqueException
    {
        this(lexemes, true);
    }

    public int valeur() {
        return valeur;
    }
}
```

Rajouter une méthode `public static void main(String [] args)` qui fabrique l'arbre correspondant à l'expression passée en paramètre de la ligne de commande et qui affiche sur la sortie standard la valeur associée à sa racine. Les expressions passées en paramètre de la ligne de commande seront composées des opérateurs binaires sur les entiers `+` et `-` et de trois variables `a`, `b` et `c` respectivement initialisées à 1, 2 et 4. Par exemple, le résultat de

```
> java ArbreSyntaxique + a - - c b b
```

sera

```
> 1
```

Pour créer les opérateurs binaires, il est conseillé d'utiliser les classes internes anonymes.

Question 3 Rajouter à la classe `ArbreSyntaxique` une méthode `public void miseAJour()` qui met à jour récursivement les valeurs des noeuds situés sur le chemin entre l'instance mise à jour et la racine de l'arbre. Cette méthode est destinée à être appelée lors de la modification de la valeur d'une variable.

On souhaite que lorsqu'une variable est modifiée, les valeurs des noeuds de l'arbre syntaxique qui en dépendent soient mises à jour automatiquement. Pour cela, on va utiliser un modèle Observateur-Observable où les noeuds de l'arbre correspondant à la variable seront les observateurs et les variables les observables.

L'interface `java.util.Observer` est rappelée ci-dessous.

```
public interface Observer {
    void update(Observable o, Object arg);
}
```

Les méthodes de la classe `java.util.Observable` dont vous aurez besoin sont :

1. `public void addObserver(Observer o)`,
2. `public void notifyObservers(Object)`
3. `protected void setChanged()`

`public void addObserver(Observer o)` "branche" un observateur sur l'observable,
`public void notifyObservers(Object)` invoque la méthode `void update(Observable, Object)` de chacune des instances d'`Observer` branchées à condition que la méthode `protected void setChanged()` ait été préalablement invoquée, indiquant que l'observé a été modifié depuis la dernière notification.
Le paramètre de `notifyObservers` sert de deuxième paramètre pour l'invocation de `void update(Observable, Object)`.

Question 4 Expliquer pourquoi il est impossible d'avoir une classe `VariableObservable` qui hérite à la fois de `java.util.Observable` et de `Variable`.

En regardant l'accessibilité des méthodes de `java.util.Observable` à utiliser, expliquer pourquoi une classe `VariableObservable` qui hérite de `Variable` et qui délègue le traitement des observateurs (branchement, notification) à une instance de `java.util.Observable` ne peut fonctionner.

Pour réaliser la seule possibilité restante : héritage de `java.util.Observable` et délégation à une instance de `Variable`, nous avons besoin de modifier le code existant.

Question 5 Afin de permettre l'utilisation du modèle de décorateur en créant une classe `VariableObservable` qui décore la classe `Variable`, il est nécessaire de fournir des interfaces `IOperateur` et `IVariable` qui déclarent les méthodes d'instance des classes `Operateur` et `Variable`. Donner le code de ces deux interfaces et préciser les modifications à apporter à `Operateur` et `Variable` pour en tenir compte.

Question 6 Rajouter à la classe `Operateur` la méthode de classe `static void substituerOperateur(String nom, IOperateur op)` qui permet le remplacement de l'opérateur associé à un nom dans la table `Operateur.ops`.

Expliquer pourquoi cette méthode est indispensable. Quel moyen peut-on mettre en oeuvre pour en éviter un usage abusif ?

Question 7 Modifier le code de `ArbreSyntaxique` pour qu'il implémente l'interface `java.util.Observer` et se branche sur les instances de `VariableObservable` pour que les mises à jour soit faites automatiquement.

Question 8 On suppose que les valeurs des variables sont modifiées par des Threads, éventuellement plusieurs pour une même variable. Que faut-il modifier pour que les mises à jour se passent correctement ?