Programmation Objet Avancée, UE INF460

Master Informatique 1 - Master Miage 1 - Master BioInformatique 2
Université Bordeaux 1 - Olivier Baudon
Mercredi 16 décembre 2009, 14h
Aucun document autorisé, durée 3 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. Écrivez 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.

Vous pouvez à chaque fois que vous le jugez compréhensible utiliser des abréviations à la place des noms complets de classe et méthode ou encore '...' pour remplacer du code non modifié.

Les instructions `import` et `package` ne seront pas précisées.

Exercice - Itérateur

On rappelle l'interface `Iterator`

```
public interface Iterator<E> {
    /** returns true if the iteration has more elements. */
    public boolean hasNext();
    /** returns the next element of the iteration. Throws
       NoSuchElementException if the iteration has no more element. */
    public E next();
    /** Removes from the underlying collection the last element
       returned by the iterator. Throws UnsupportedOperationException
       if the remove  operation is not supported by this Iterator. */
    public void remove();
}
```

Question 1 Dans une classe `Iterateurs`, écrire une méthode de classe

`public static <E> Iterator<E> ajouterElement(E element, Iterator<E> it)`
qui retourne un itérateur $it2$ dont les éléments sont d'abord *element*, puis les éléments de it. La méthode `remove` d'$it2$ lèvera une exception `UnsupportedOperationException`.

Exemple : le code suivant

```
String[] ts = {"a", "b", "c"};
Iterator<String> it = Iterateurs.ajouterElement("x",
    java.util.Arrays.asList(ts).iterator());
while (it.hasNext())
    System.out.print(it.next() + " ");
System.out.println();
```

affichera sur la sortie standard

```
x a b c
```

Question 2 On rappelle que les classes `Double` et `Integer` héritent de la classe `Number`.

Le code suivant :

```
Integer [] ti = {1, 2, 3};
Number n = new Double(4.);
Iterator<Number> it2 = Iterateurs.<Number>ajouterElement(n, java.util.Arrays
.asList(ti).iterator());
```

entraîne l'erreur de compilation ci-dessous :

```
The parameterized method <Number>ajouterElement(Number, Iterator<Number>)
of type Iterateurs is not applicable for the arguments (Number, Iterator<Integer>)
```

Expliquez en une phrase pourquoi et modifier la déclaration de `Iterateurs.ajouterElement` pour éliminer cette erreur.

Exercice - Personne

On considère la classe `Personne` suivante :

```
public class Personne {
    private String nom;

    public Personne(String nom) {
        this.nom = nom;
    }

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

```
    }

    public boolean equals(Object o) {
        if (!(o instanceof Personne))
            return false;
        Personne p = (Personne) o;
        return p.nom.equals(nom);
    }
}
```

Question 1 Modifier cette classe pour que l'affichage sur la sortie standard d'une instance de `Personne` affiche son nom.

Par exemple, le résultat des instructions :

```
Personne dupont = new Personne("Dupont");
System.out.println(dupont);
```

sera l'affichage sur la sortie standard de

```
Dupont
```

Question 2 On considère la classe `Personne2` qui étend la classe `Personne` en rajoutant un prénom.

```
public class Personne2 extends Personne {
    private String prenom;

    public Personne2(String prenom, String nom) {
        super(nom);
        this.prenom = prenom;
    }

    public String prenom() {
        return prenom;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Personne2))
            return false;
        Personne2 p = (Personne2) o;
        return super.equals(p) && p.prenom().equals(prenom);
    }
}
```

Expliquer pourquoi la méthode `equals` de `Personne2` n'est pas correcte.

Question 3 On propose de remplacer la méthode `equals(Object)` de `Personne2` par le code suivant :

```
public boolean equals(Object o) {
    if (!(o instanceof Personne))
        return false;
    else if (!(o instanceof Personne2))
        return o.equals(this);
    else {
        Personne2 p = (Personne2) o;
        return super.equals(p) && p.prenom().equals(prenom);
    }
}
```

Pourquoi cette nouvelle solution n'est-elle pas non plus acceptable ?

Question 4

Proposer une solution pour l'implémentation de `Personne2` qui soit correcte pour la notion d'égalité et sans engendrer de duplication de code.

Problème - Jeu

On souhaite étudier un jeu à l'aide d'une simulation.

Ce jeu comporte deux joueurs : "Rouge" et "Bleu". Ces joueurs déposent des pions sur des cases. Puis ils doivent à tour de rôle déplacer un de leur pion sur une case voisine. Un joueur a perdu quand au moins un de ses pions ne peut plus se déplacer.

On considère la classe suivante `Jeu`. La méthode `jouerAleatoire()` permet de tester des configurations en faisant jouer les joueurs de façon aléatoire.

```
public class Jeu {
    public static int ROUGE = 0;
    public static int BLEU = 1;

    /*
     * Positions respectives des pions rouges et bleus Par exemple,
     * positions[ROUGE][i] = j si le ieme pion rouge est sur la case j
     */
    private int[][] positions;
    /*
     * liaisons[i][j] est vrai si l'on peut deplacer un pion de la case i a la
     * case j.
     */
    private boolean[][] liaisons;
    /*
     * Occupation des cases par les pions. Les pions rouges sont representes par
     * des entiers pairs, les bleus par des entiers impairs. occupations[j] =
     * 2*i si la case j est occupee par le ieme pion de rouge, 2*i+1 si la case
```

```
 * j est occupee par le ieme pion bleu. INOCCUPE signifie aucun pion
 * present.
 */
private int[] occupations;
public static final int INOCCUPE = -1;

/*
 * Generateur aleatoire. La methode random.nextInt(int n) renvoie
 * aleatoirement un entier positif compris entre 0 et n-1.
 */
private Random random = new Random();

public Jeu(boolean[][] liaisons, int[] rouges, int[] bleus) {
    this.liaisons = liaisons;
    this.positions = new int[2][];
    this.positions[ROUGE] = rouges;
    this.positions[BLEU] = bleus;
    int n = liaisons.length;
    occupations = new int[n];
    for (int i = 0; i < n; i++)
        occupations[i] = INOCCUPE;
    for (int i = 0; i < rouges.length; i++)
        occupations[rouges[i]] = 2 * i;
    for (int i = 0; i < bleus.length; i++)
        occupations[bleus[i]] = 2 * i + 1;
}

/**
 * Deplacement d'un pion.
 */
public void deplacer(int couleur, int pion, int destination) {
    int position = positions[couleur][pion];
    positions[couleur][pion] = destination;
    occupations[position] = INOCCUPE;
    occupations[destination] = 2 * pion + couleur;
}

/**
 * Teste si un joueur a perdu.
 */
public boolean perdu(int couleur) {
    for (int i = 0; i < positions[couleur].length; i++) {
        if (accessibles(positions[couleur][i]).size() == 0)
            return true;
    }
```

```
    return false;
}

/**
 * Renvoie la liste des positions accessibles a partir d'une position
 * donnee.
 */
public List<Integer> accessibles(int position) {
    List<Integer> accessibles = new ArrayList<Integer>();
    for (int i = 0; i < occupations.length; i++) {
        if (liaisons[position][i] && occupations[i] == INOCCUPE)
            accessibles.add(i);
    }
    return accessibles;
}

/**
 * Teste une partie ou les joueurs jouent aleatoirement.
 */
public void jouerAleatoire() {
    int nCoups = 0;
    for (;;) { // boucle infinie
        nCoups++;
        for (int couleur = 0; couleur < 2; couleur++) {
            /*
             * On choisit d'abord aleatoirement le pion a deplacer parmi les pions
             * deplacables, puis toujours aleatoirement la nouvelle position parmi
             * celles accessibles.
             */
            int pion = random.nextInt(positions[couleur].length);
            List<Integer> accessibles = accessibles(positions[couleur][pion]);
            deplacer(couleur, pion, accessibles.get(random
                    .nextInt(accessibles.size())));
            if (perdu((couleur + 1) % 2)) {
                System.out.println((couleur == ROUGE ? "Bleu" : "Rouge")
                        + " a perdu en " + nCoups + " coups !");
                return;
            }
        }
    }
}

public static void main(String[] args) {
    Jeu jeu = new Jeu(new boolean[][] { { false, true, false },
            { true, false, true }, { false, true, false } },
```

```
                new int[] { 0 }, new int[] { 2 });
        jeu.jouerAleatoire();
    }
}
```

Question 1 Quel sera le résultat de l'exécution de la classe `Jeu` avec la méthode `main` ici présente ?

Question 2 Rajouter une méthode `position(int couleur, int pion)` qui renvoie la position du pion d'indice `pion` et de couleur `couleur`.

Question 3 On considère le code suivant :

```
int [] rouge = new int[] {0};
int [] bleu = new int[] {2};
Jeu jeu = new Jeu(new boolean[][] { { false, true, false },
{ true, false, true }, { false, true, false } }, rouge, bleu);
// INSTRUCTION CACHEE N'UTILISANT PAS LA METHODE DEPLACER
System.out.println("position du pion 0 de rouge: " + jeu.position(Jeu.ROUGE, 0));
```

dont le résultat affiché est :

```
position du pion 0 de rouge: 1
```

Donner le code de l'instruction cachée. Corriger ce défaut d'encapsulation.

Question 4 Dans la méthode `deplacer(int couleur, int pion, int destination)`, il est possible de déplacer un pion alors que la case contenant le pion et la case destination ne sont pas reliées, ou encore alors que la case destination est déjà occupée.

Modifier la méthode `deplacer` pour qu'elle lève une exception `DeplacementImpossibleException` dans ces deux cas. Donner le code de la classe `DeplacementImpossibleException`.

Question 5 De façon à pouvoir facilement modifier les stratégies utilisées, il est décidé de refaire le code en utilisant deux interfaces `Joueur` et `Deplacement` décrites ci-dessous :

```
public interface Joueur {
    public Deplacement jouer();
}

public interface Deplacement {
    public int pion();
    public int destination();
}
```

Ecrire une classe `DeplacementImpl` qui implémente `Deplacement`. Les données nécessaires seront fournies à la construction.

Ecrire une classe `JoueurAleatoire` qui implémente `Joueur` et fournit un joueur au comportement aléatoire comme dans la version actuelle de `Jeu`. Le joueur aléatoire sera créé dans la classe `Jeu` qui lui fournira à la construction toutes les données dont il a besoin.

Modifier la classe `Jeu` en conséquence.