

Programmation Objet Avancée, UE INF460

Corrigé

Master Informatique 1 - Master Miage 1 - Master
BioInformatique 2

Université Bordeaux 1 - Olivier Baudon

Mercredi 16 décembre 2009, 14h

Exercice - Itérateur

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`.

```
public class Iterateurs {
    public static <E> Iterator<E> ajouterElement(E element, Iterator<E> it) {
        final E e = element;
        final Iterator<E> suite = it;
        return new Iterator<E>() {
            private boolean premier = true;

            public boolean hasNext() {
                return premier || suite.hasNext();
            }

            public E next() {
                if (premier) {
                    premier = false;
                    return e;
                } else {
                    return suite.next();
                }
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Question 2 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.

Le type `Number` est substitué au paramètre de type `E`. On attend donc un `Number` pour la variable `element` et un `Iterator<Number>` pour la variable `it`. Or si les instances de `Double` et `Integer` sont des instances de `Number`, une instance de `Iterator<Integer>` n'est pas une instance de `Iterator<Number>`. Il faut donc modifier le prototype de la méthode `ajouterElement` de la façon suivante :

```
public static <E> Iterator<E> ajouterElement(E element, Iterator<? extends
E> it)
```

Exercice - Personne

On considère la classe `Personne`

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

Il faut rajouter à la classe `Personne` la méthode `toString` suivante :

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

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

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

Cette méthode entraîne le fait que la méthode `equals` n'est plus symétrique. Par exemple, le code suivant :

```
Personne p1 = new Personne("Dupont");
Personne2 p2 = new Personne2("Jean", "Dupont");
System.out.println(p1.equals(p2) + " " + p2.equals(p1));
```

affiche

```
true false
```

alors que les deux valeurs devraient être égales.

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 ?

Cette nouvelle méthode entraîne le fait que la méthode `equals` n'est plus transitive. Par exemple, le code suivant :

```
Personne p1 = new Personne("Dupont");
Personne2 p2 = new Personne2("Jean", "Dupont");
Personne2 p3 = new Personne2("André", "Dupont");
System.out.println(p2.equals(p1) + " " + p1.equals(p3) + " => " + p2.equals(p3));
```

affiche

```
true true => false
```

alors que le troisième booléen devrait également valoir true

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.

La meilleure solution consiste à réutiliser le code de `Personne` dans `Personne2` en utilisant non pas l'héritage, mais la délégation.

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

    public Personne2(String prenom, String nom) {
        delegate = new Personne(nom);
        this.prenom = prenom;
    }
}
```

```

    public String nom() {
        return delegate.nom();
    }

    public String prenom() {
        return prenom;
    }

    public String toString() {
        return prenom + " " + nom();
    }

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

```

Problème - Jeu

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

Le résultat affiché sera Bleu a perdu en 1 coups !

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

```

public int position(int couleur, int pion) {
    return positions[couleur][pion];
}

```

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.

L'instruction cachée est

`rouge[0] = 1;`

Pour corriger ce défaut d'encapsulation, il faut recopier les tableaux passés en paramètres à la construction :

```
public Jeu(boolean[][] liaisons, int[] rouges, int[] bleus) {
    this.liaisons = liaisons.clone();
    this.positions = new int[2][];
    this.positions[ROUGE] = rouges.clone();
    this.positions[BLEU] = bleus.clone();
    ...
}
```

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`.

```
public void deplacer(int couleur, int pion, int destination)
    throws DeplacementImpossibleException {
    int position = positions[couleur][pion];
    if (!liaisons[position][destination] || occupations[destination] != INOCCUPE)
        throw new DeplacementImpossibleException(position, destination);
    positions[couleur][pion] = destination;
    occupations[position] = INOCCUPE;
    occupations[destination] = 2 * pion + couleur;
}
```

```
public class DeplacementImpossibleException extends Exception {
    private int depart, arrivee;

    public DeplacementImpossibleException(int depart, int arrivee) {
        this.depart = depart;
        this.arrivee = arrivee;
    }

    public int positionDepart() {
```

```

        return depart;
    }

    public int positionArrivee() {
        return arrivee;
    }

    public String getMessage() {
        return "Deplacement impossible de " + depart + " vers " + arrivee;
    }
}

```

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`.

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

```

public class DeplacementImpl implements Deplacement {
    private int pion;
    private int destination;

    public DeplacementImpl(int pion, int destination) {
        this.pion = pion;
        this.destination = destination;
    }

    public int destination() {
        return destination;
    }

    public int pion() {
        return pion;
    }
}

```

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.

```

public class JoueurAleatoire implements Joueur {
    private Random random = new Random();
    private int nPions;
    private Jeu jeu;
}

```

```

private int couleur;

public JoueurAleatoire(int couleur, int nPions, Jeu jeu) {
    this.couleur = couleur;
    this.nPions = nPions;
    this.jeu = jeu;
}

public Deplacement jouer() {
    int pion = random.nextInt(nPions);
    List<Integer> accessibles = jeu
        .accessibles(jeu.position(couleur, pion));
    return new DeplacementImpl(pion, accessibles.get(random
        .nextInt(accessibles.size())));
}
}

```

Modifier la classe Jeu en conséquence.

```

public class Jeu {
    ...
    private Joueur[] joueurs;

    public Jeu(boolean[][] liaisons, int[] rouges, int[] bleus) {
        ...
        this.joueurs = new Joueur[2];
        for (int i = 0; i < 2; i++)
            joueurs[i] = new JoueurAleatoire(i, positions[i].length, this);
        ...
    }

    ...

    /**
     * Teste une partie ou les joueurs jouent aleatoirement.
     */
    public void jouerAleatoire() throws DeplacementImpossibleException {
        int nCoups = 0;
        for (;;) { // boucle infinie
            nCoups++;
            for (int couleur = 0; couleur < 2; couleur++) {
                Deplacement d = joueurs[couleur].jouer();
                deplacer(couleur, d.pion(), d.destination());
                if (perdu((couleur + 1) % 2)) {
                    System.out.println((couleur == ROUGE ? "Bleu" : "Rouge"))

```

```

        + " a perdu en " + nCoups + " coups !");
        return;
    }
}

public static void main(String[] args)
    throws DeplacementImpossibleException {
    Jeu jeu = new Jeu(new boolean[][] { { false, true, false },
                                          { true, false, true }, { false, true, false }
                                          },
                      new int[] { 0 }, new int[] { 2 });
    jeu.jouerAleatoire();
}
}

```