université de BORDEAUX

4TYE707U
Programmation orientée objet
Master MIAGE

Examen du 13 décembre 2016, durée 3h
Corrigé

Exercice - Boîte

Question 1 Compléter le code.

```
public class Boite {
    private Object contenu;

    public Boite(Object contenu) {
        this.contenu = contenu;
    }

    public Object contenu() {
        return contenu;
    }
}
```

Question 2 Modifier le code pour que le type de l'objet contenu dans la boîte soit générique.

```
public class Boite<E> {
    private E contenu;

    public Boite(E contenu) {
        this.contenu = contenu;
    }

    public E contenu() {
        return contenu;
    }
}
```

Question 3 Modifier le code de la question 2 pour que si l'on exécute l'instruction

```
System.out.println(b)
```

où `b` désigne une instance de `Boite` contenant un objet `o`, le résultat soit l'affichage sur la sortie standard du message suivant :

```
Boîte contenant <chaîne décrivant l'objet contenu>
```

Il faut redéfinir la méthode `String toString()`.

```
public class Boite<E> {
    ...
    public String toString() {
        return "Boîte contenant " + contenu;
    }
}
```

Question 4 Modifier le code pour que deux boîtes soient égales si les objets qu'elles contiennent sont égaux (au sens de la méthode `equals`). Quelle autre méthode héritée de la classe `Object` est-il dans ce cas souhaitable de modifier ? Proposer également un code pour cette méthode.

Il faut redéfinir la méthode `boolean equals(Object o)`, ainsi que la méthode `int hashCode()` car deux objets égaux doivent toujours renvoyer la même valeur lors de l'appel de `hashCode()`.

```
public class Boite<E> {
    ...
    public boolean equals(Object o) {
        if (!(o instanceof Boite))
            return false;
        return ((Boite<?>) o).contenu.equals(contenu);
    }

    public int hashCode() {
        return contenu.hashCode();
    }
}
```

Exercice - Itération

Question 1 Écrire dans une classe `Iterations` une méthode de classe
`public static <E> Iterator<E> iterationContinue(E valeur)`.
L'itérateur obtenu renverra indéfiniment la valeur `valeur` passée en paramètre. Les méthodes `remove()` et `forEachRemaining()` devront lever l'exception `UnsupportedOperationException` (*attention, ce n'est pas le comportement par défaut de `forEachRemaining()`*).

```
public class Iterations {
    public static <E> Iterator<E> iterationContinue(E valeur) {
        return new Iterator<E>() {

            public boolean hasNext() {
                return true;
            }

            public E next() {
                return valeur;
            }

            public void forEachRemaining(Consumer<? super E> action) {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Question 2 On souhaite maintenant pouvoir manipuler plusieurs valeurs dans l'itération. La nouvelle déclaration de la méthode `iterationContinue` sera donc :
`public static <E> Iterator<E> iterationContinue(E... valeurs)`.
L'itérateur obtenu retournera les valeurs passées en paramètre une par une et recommencera à la première une fois toutes les valeurs retournées.

```
public class Iterations {
    public static <E> Iterator<E> iterationContinue(E... valeurs) {
        E[] valeurs2 = valeurs.clone();

        return new Iterator<E>() {
            int i = 0;

            public boolean hasNext() {
                return true;
            }

            public E next() {
                E v = valeurs2[i];
                i = (i+1) % valeurs2.length;
                return v;
            }

            public void forEachRemaining(Consumer<? super E> action) {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Exercice - Observable

On considère l'interface `Valeurs` et les deux classes `ValeursImpl` et `Histogramme`.

Question 1 Montrer que dans la classe `ValeursImpl`, il est possible de modifier le résultat retourné par la méthode `valeur(int i)` sans utiliser `changerValeur(int i, double v)` et donc sans que la variable `somme` soit mise à jour. Corriger ce défaut d'encapsulation.

Lors de l'instanciation de ValeursImpl, on stocke la référence (adresse) du tableau `valeurs` passé en paramètre du constructeur. Si le "client" modifie ensuite le contenu de ce tableau, la méthode `valeur(int i)` renverra le nouveau contenu. Par exemple :

```
int [] t = {1, 2, 3};
Valeurs v = new ValeursImpl(t);
System.out.print(v.valeur(0) + " ");
t[0] = 4;
System.out.println(v.valeur(0));
```

affichera

```
1 4
```

Il faut donc stocker une copie du tableau et non seulement sa référence, en utilisant par exemple la méthode `clone()`.

```
public class ValeursImpl implements Valeurs {
    private double[] valeurs;
    private double somme;

    public ValeursImpl(double[] valeurs) {
        this.valeurs = valeurs.clone();
        ... // reste du code inchangé
    }
    ...
}
```

Question 2 Modifier le constructeur `ValeursImpl(double[] valeurs)` pour qu'il lève une exception `ValeurNegativeException` si une des valeurs passées dans le tableau `valeurs` est négative.
Donner également le code de la classe `ValeurNegativeException`.

```
public class ValeursImpl implements Valeurs {
    private double[] valeurs;
    private double somme;

    public ValeursImpl(double[] valeurs) throws ValeurNegativeException {
        this.valeurs = valeurs.clone();
        somme = 0;
        for (int i = 0; i < valeurs.length; i++) {
            if (valeurs[i] < 0)
                throw new ValeurNegativeException(i, valeurs[i]);
            somme += valeurs[i];
        }
    }
    ... // reste du code inchangé
}

public class ValeurNegativeException extends Exception {
    private int i;
    private double valeur;

    public ValeurNegativeException(int i, double valeur) {
        this.i = i;
        this.valeur = valeur;
    }

    public int index() {
        return i;
    }

    public double valeur() {
        return valeur;
    }

    public String getMessage() {
        return "Valeur " + valeur + " négative à l'indice " + i;
    }
}
```

Question 3 On souhaite également qu'il soit impossible de passer une valeur négative à la méthode `changerValeur(int i, double v)`. Ecrire les modifications à apporter.

Il faut bien sûr modifier la méthode `changerValeur(int i, double v)` dans la classe `ValeurImpl`, mais aussi sa définition dans l'interface `Valeur` pour que celle-ci déclare la possible levée de l'exception.

```
public interface Valeurs {
    ... // début du code inchangé
    public void changerValeur(int i, double v) throws ValeurNegativeException;
}

public class ValeursImpl implements Valeurs {
    ... // début du code inchangé
    public void changerValeur(int i, double v) throws ValeurNegativeException {
        if (v < 0)
            throw new ValeurNegativeException(i, v);
        somme += v - valeurs[i];
        valeurs[i] = v;
    }
    ... // reste du code inchangé
}
```

On souhaite que les instances d'`Histogramme` soient automatiquement mises à jour. Pour cela, on va utiliser l'interface `java.util.Observer` et la classe `java.util.Observable`.

Question 3 Ecrire une classe `ValeursObservables` qui rend une instance de `Valeurs` observable. La classe `ValeursObservables` devra donc étendre la classe `Observable` et implémenter l'interface `Valeurs` en utilisant une instance de `Valeurs` déjà existante.

```
public class ValeursObservables extends Observable implements Valeurs {

    private Valeurs valeurs;

    public ValeursObservables(Valeurs valeurs) {
        this.valeurs = valeurs;
    }

    public int taille() {
        return valeurs.taille();
    }

    public double somme() {
        return valeurs.somme();
    }

    public double valeur(int i) {
        return valeurs.valeur(i);
    }

    public void changerValeur(int i, double v) throws ValeurNegativeException {
        valeurs.changerValeur(i, v);
        setChanged();
        notifyObservers(i);
    }
}
```

Question 4 Ecrire une classe `HistogrammeObservateur` qui permet de disposer d'histogrammes observant l'instance de `Valeurs` qu'ils affichent. En d'autre termes, en utilisant le code de la question 3,

tout appel de la méthode `changerValeur(int i, double v)` sur l'instance de `Valeurs` affichée par l'histogramme devra entraîner une mise à jour de celui-ci.

```
public class HistogrammeObservateur extends Histogramme implements Observer {

    public HistogrammeObservateur(ValeursObservables val) {
        super(val);
        val.addObserver(this);
    }

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