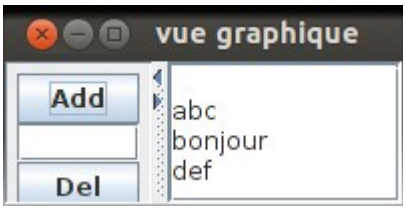
Licence Science-Technologie-Santé - mention Informatique - 3e année

module Programmation Objet 2

Examen – janvier 2013 – durée : 2h

Documents autorisés : les polycopiés du cours.

Partie A : Gestion d'une dictionnaire *(environ 12 points)*



Notre application, écrite selon le pattern MVC, n'est pas tout à fait finie.

Question A1 :

Ecrire la classe Dictionnaire.java qui constitue notre modèle :

- Elle hérite de Observable.
- Elle possède un attribut qui contient une liste des mots.
- Elle comporte un constructeur sans paramètre qui crée une liste vide.
- Elle comporte une méthode « add » pour ajouter un mot.
- Elle comporte une méthode « remove » pour supprimer un mot.

Voici le code de la classe DictionnaireGestion1.java :

```
public class DictionnaireGestion1 {
  public static void main(String[] args) {
    final Dictionnaire dict = new Dictionnaire();
       SwingUtilities.invokeLater(new Runnable() {
      public void run() {
          ControleurGraphik controlGra = new ControleurGraphik(dict);
          VueGraphik vueGra = new VueGraphik(dict);
          JFrame frame = new JFrame("vue graphique");
          JPanel panel = new JPanel();
          panel.add(controlGra);
          // The splitPane is split horizontally – the two components panel and vueGra appear side
       by side
          JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, panel, vueGra);
          // The user can collapse (and then expand) either of the components with a single click
          splitPane.setOneTouchExpandable(true);
          // Make 50% of the space go to left component panel
          splitPane.setDividerLocation(0.5);
          frame.getContentPane().add(splitPane);
          frame.setSize(200,100);
          frame.setVisible(true);
```

```
      }
   });
  }
}
```

Et voici le code de la classe ControleurGraphik.java :

```
public class ControleurGraphik extends JPanel
{
  /** le dictionnaire a controler */
  private Dictionnaire dict;
  /** pour saisir un mot */
  private JTextField motField = null;
  /** constructeur */
  public ControleurGraphik(Dictionnaire d) {
    super(new BorderLayout());
    dict = d;
    motField = new JTextField("");
    JButton addButton = new JButton("Add");
    addButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          dict.add(motField.getText());
          motField.setText("");
    }} );
    JButton delButton = new JButton("Del");
    delButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          dict.remove(motField.getText());
          motField.setText("");
    }} );
    this.add(motField, BorderLayout.CENTER);
    this.add(addButton, BorderLayout.NORTH);
    this.add(delButton, BorderLayout.SOUTH);
  }
}
```

Question A2 :

Ecrire la classe VueGraphik.java qui affiche la liste des mots de la dictionnaire.

Partie B : Mise en Cache *(environ 8 points)*

La mise en cache permet d'optimiser les temps d'exécution. Par exemple : l'accès à des fichiers, la génération de fichier HTML via PHP, le calcul de fonctions …
Voici un fichier d'interface modélisant les taches à résultat réutilisable :

```
public interface CalculReutilisable<T,R> {
  /* calcul re-utilisable posséde
     une méthode calculer, couteuse en temps d'exécution,
     qui travaille sur des données de type T,
     fournit des résultats de type R.
  */
  R calculer(T arguments);
}
```

Quand le calcul a été fait pour la valeur 1515, il suffit de mémoriser le résultat calculé pour le ressortir s'il est demandé une nouvelle fois.
Voici une première implémentation CalculerEtCacher1.java :

```
1  import java.util.*;
2  public class CalculerEtCacher1<T,R> implements CalculReutilisable<T,R> {
3    // @GuardedBy("this")
4    /* cache permet de stocker les résultats déjà calculés
5       chaque résultat est "indicé" par la donnée du calcul */
6    private final Map<T,R> cache = new HashMap<T,R>();
7    private CalculReutilisable<T,R> maTache = null;
8    public CalculerEtCacher1(CalculReutilisable<T,R> tache) {
9      this.maTache = tache;
10    }
11    public synchronized R calculer(T arguments) {
12      R resultatPrecedent = cache.get(arguments);
13      if (resultatPrecedent != null)
14        return resultatPrecedent;
15      R resultat = maTache.calculer(arguments); // rappel : calcul très long !
16      cache.put(arguments, resultat);
17      return resultat;
18    }
19  }
```

Remarque : une map de Java est une sorte d'array associatif PhP

Voici un banc de test : les données sont fournies au clavier par l'utilisateur ; un pool de 2 threads exécutent au fer et à mesure les calculs; les résultats sont affichés en console.

```
1  import java.util.*;
2  public class Test0 {
3    public static void main (String [] args)  {
4      new Test0();
5    }
6    public Test0() {
7      CalculerEtCacher1<Integer, Integer> calculerEtCacher =
8          new CalculerEtCacher1<Integer, Integer>(new CalculNiemeDecimaleDePi());
9      Scanner scanIn = new Scanner(System.in);
10      while (scanIn.hasNextInt()) {
11        int nombre = scanIn.nextInt();
12        AFaire aFaire =new AFaire(calculerEtCacher, nombre);
13        Thread thread = new Thread(aFaire);
14        thread.start();
15      }
16    }
17    class AFaire implements Runnable{
18      private int nombre;
19      private CalculerEtCacher1<Integer, Integer> calculerEtCacher;
20      public AFaire(CalculerEtCacher1<Integer, Integer> c, int n) {
21        calculerEtCacher = c; nombre = n;
22      }
23      public void run() {
```

```
24        int resultat = calculerEtCacher.calculer(nombre);
25        System.out.println("calcul("+nombre+") = "+resultat);
26      }
27    }
28  }
```

Voici la classe de calcul réutilisable implémentant l'interface :

```
public class CalculNiemeDecimaleDePi implements CalculReutilisable<Integer, Integer> {
  public Integer calculer(Integer x) {
    /* calcul bidon mais ... long ! */
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
    if (y < 0)
      y = - y;
    return y%10;
  }
}
```

Les questions sont indépendantes :

1. La solution programmée dans `CalculerEtCacher1` est une catastrophe. Dans le programme de test, quand les données sont toutes différentes, le temps d'exécution est le même qu'avec une exécution séquentielle par le thread main. Par contre, dès que des données ont déjà été calculées, la technique du cache diminue le temps d'exécution.
   Quel est le problème? *(1 à 2 lignes d'explication)*

2. Proposer une meilleure solution `CalculerEtCacher2` en changeant un peu son code

3. Le premier programme de test générait plein de thread de façon très inefficace. En voici une amélioration avec un pool de 2 threads "exécutant" et un mécanisme de producteur-consommateur (implémenté par une ArrayBlockingQueue) : le clavier produit les données, et les threads exécutants les consomment.
   Le code des « exécutants » est à écrire:

```
1  import java.util.*;
2  import java.util.concurrent.*;
3  public class Test1 {
4    public static void main (String [] args)  {
5      new Test1();
6    }
7    public Test1() {
8      CalculerEtCacher2<Integer, Integer> calculerEtCacher =
9          new CalculerEtCacher2<Integer, Integer>(new CalculNiemeDecimaleDePi());
10     ArrayBlockingQueue<Integer> fileDeNombres = new ArrayBlockingQueue<Integer>(20);
11     Executant exe1 = new Executant( fileDeNombres, calculerEtCacher);
12     exe1.start();
13     Executant exe2 = new Executant( fileDeNombres, calculerEtCacher);
14     exe2.start();
15     Scanner scanIn = new Scanner(System.in);
16     int nombre = 0;
17     try {
18       while (scanIn.hasNextInt()) {
19         nombre = scanIn.nextInt();
20         fileDeNombres.put(nombre);
21       }
22     } catch (InterruptedException ie) {}
23   }
24   class Executant extends Thread {
   …...
39   }
40 }
```