# Algorithmique avancée
# TD n° 3

## 1 : étude et implémentation du Robot manipulateur de palettes.

Des palettes numérotées de 0 à n sont empilées en désordre sur un emplacement (la source). Un robot manipulateur doit, en s'aidant d'un emplacement auxiliaire, disposer les palettes en ordre ( le plus petit numéro en bas, le plus grand au sommet) sur un troisième emplacement (la destination). Il ne peut déplacer qu'une palette à la fois (cf cours).

L'algorithme s'inspire du tri par sélection du minimum analysé dans le TD1 avec 2 piles. Le robot essaye d'abord de récupérer la palette de plus petit numéro. Pour cela :
- il empile dans l'emplacement auxiliaire toutes les palettes situées au dessus de la palette recherchée.
- puis il met la palette de plus petit numéro à sa place.

Il recherche ensuite la plus petite restante pour la mettre en place, et ainsi de suite …

### 1.1 Implémentation de l'algorithme

On a implémenté au TD2 une classe PileInt. Vous allez d'abord implémenter et tester l'algorithme sans interface graphique, puis ajouter l'interface graphique.

**Q1**. Quels sont les traitements que doit effectuer la classe implémentant les piles de palettes pour exécuter l'algorithme ?

**Q2**. Comme on n'a pas de visualisation, on veut que chaque pile de palettes écrive à chaque opération qu'elle fait son nom et l'opération sur System.out. Comment faire cela ?

**Q3**. Implémentez la classe PileIntTrace qui effectue les traitements prévus dans Q1 et dans Q2 et testez la. Un constructeur crée une pile vide, un autre crée une pile avec les entiers de 0 à n-1 dans un ordre aléatoire. On donne pour cela quelques lignes de code qui initialisent un tableau de taille nb avec les nombres de 0 à nb-1 placés au hasard :

```
int t[] = new int[nb];
      for (int i=0;i<nb;i++)  t[i]=i;  // remplissage dans l'ordre
      for (int i=0;i<5*nb;i++)   // mélange
      { int r1= (int) (nb * Math.random()); // on tire au sort 2 cases à permuter
        int r2= (int) (nb * Math.random());
        int aux = t[r1];   t[r1]=t[r2];   t[r2]=aux; // on les permute
      }  // end for
```

Implémentation du robot : on crée une classe RobotTrace. Cette classe a en variable d'instance un tableau pPal[] de trois PileIntTrace : pPal[0] est la source, pPal[1] est la pile auxiliaire, pPal[2] est la destination. La méthode manipuler exécute l'algorithme. On donne la classe TestRobotTrace qui utilise ce robot :

```
public class TestRobotTrace {
    public static void main (String args[])          {
        PileIntTrace source = new PileIntTrace ("source",12);
        PileIntTrace destination = new PileIntTrace ("destination");
        PileIntTrace temporaire = new PileIntTrace ("temporaire");
        RobotTrace robot = new RobotTrace ();
        robot.manipuler(source,destination,temporaire);
```

```
    }// end main
  }// end class
```

Codage de l'algorithme de la méthode manipuler() : pour savoir dans quelle pile chercher la prochaine palette (la pile numéro 0 ou la pile numéro 1 de pPal[]), le robot s'aide d'un tableau position[] possédant autant de cases qu'il y a de palettes.
Au début, toutes les palettes sont dans la pile 0 : il initialise donc toutes les cases du tableau à 0. Lorsqu'il transfère une palette marquée p dans la pile auxiliaire (la n° 1), la case correspondante pos[p] du tableau est mise à 1. Plus généralement, un transfert d'une palette p dans la pile de numéro n s'accompagne de la mise à n de la case pos[p]
Le reste de la méthode suit l'algorithme du cours.
**Q4**. Implémenter la classe RobotTrace comportant les constructeurs utilisés dans TestRobotTrace et la méthode manipuler().

### 1.2 Implémentation du modèle observateur / observé : les observés

Du coté observé, on fait l'implémentation en deux temps : une classe ObservablePileInt utilisable pour observer n'importe quelle pile d'entiers, et une classe PilePalette qui implémente ce qui est spécifique au robot

#### 1.2.1 pile d'entiers observable

On crée donc une sous-classe ObservablePileInt de Observable qui encapsule une PileInt (cf cours). Comme on veut accéder à la pile encapsulée de façon non standard pour l'afficher, on va en fait accéder à une inner classe protégée de ObservablePileInt qui sous-classe PileInt et dans laquelle on ajoute les méthodes non standard. La pile observable a aussi un nom. Le schéma est ci-dessous :

```
import java.util.* ;
public class ObservablePileInt extends Observable   {
        protected InnerPileInt p;
        protected String nom ;
        public ObservablePileInt()      {
                /* à compléter */
        }

        public void empiler(int i)      {
                /* à compléter */
        }
        /* autres méthodes */

        protected class InnerPileInt extends PileInt{
                protected InnerPileInt() {
                        super() ;
                } // fin du constructeur

                public int getSommet(){
                        return sommet ;
                } // fin de getSommet

                /* autres méthodes */
        } // fin de InnerPileInt

}  // fin de la classe ObservablePileInt
```

**Q5**. Compléter la classe ObservablePileInt en y écrivant les méthodes standard de Pileint, et deux méthodes non standard : public Vector getAsVector() renvoie une copie de la pile (dans un Vector, on ne peut mettre que des objets, par exemple des Integer), et public int getNbElements() renvoie le nombre d'éléments dans la pile. Quand cette classe lance un notifyObservers(), l'observateur doit pouvoir savoir à la réception si le update() signale un empilement ou un dépilement.

### 1.2.2 PilePalettes

On crée une sous-classe PilePalettes de ObservablePileInt. Dans PilePalette, on met les constructeurs qui permettent d'initialiser la pile en y copiant un tableau ou en y plaçant au hasard les nombres de 0 à n-1. Pour le reste PilePalette est identique à ObservablePieInt.
**Q6**. Écrire la classe PilePalettes.

## 1.3 Implémentation du modèle observateur / observé : les observateurs

On va implémenter deux sortes d'observateurs : avec affichage sous forme texte (dans un JTexteArea), et avec affichage sous forme graphique (un tableau de JLabel). Chaque observateur est attaché à un JPanel. Tous les observateurs dérivent de PanelVisu.

### 1.3.1 la classe racine des observateurs

Pour pouvoir exécuter l'algorithme en pas à pas, on utilise une classe de contrôle appelée PanelNext (classe fournie) : chaque instance affiche un bouton ; quand on appelle la méthode next() de cette instance, l'exécution est suspendue jusqu'à ce qu'on ait cliqué sur le bouton.
**Q7**. Écrire la classe abstraite PanelVisu qui implémente Observer. Cette classe a deux variables membres : un PanelNext et un Observable. Elle a un seul constructeur public PanelVisu (Observable obs,PanelNext pn). La méthode update() est abstraite.

### 1.3.2 L'observateur PanelVisuTexte

Il a un JtextArea en variable membre. Le constructeur installe ce JtextArea dans le Panel, la méthode update() attend le bouton seulement quand c'est un dépilement. Dans tous les cas, elle termine en écrivant une trace dans le JTextArea (au choix : en reprenant la trace de Q2, ou en utilisant le toString() de la pile)
**Q8**. Écrire PanelVisuTexte

### 1.3.3 L'observateur PanelVisuTab

Il construit l'affichage de la liste en plaçant des Jlabels dans un GridLayout dont le nombre de positions est *tailleGraph*. Pour empiler vers le haut, le fond de la pile (indice 0 dans la pile) est placé en fin dans la grille (indice tailleGraph – 1). Comme le code est un peu long à mettre au point pour le temps dont nous disposons, le code du constructeur est fourni :

```
public class PanelVisuTab extends PanelVisu{
   private JLabel[] palIm ;
   private int tailleGraph, sommet ;
   private static String vide = "          " ; /* determine la largeur de  l'affichage */

   public PanelVisuTab(ObservablePileInt obs, PanelNext pn, int tg) {
         super(obs,pn) ;
         tailleGraph = tg ;
         this.setLayout(new GridLayout(tg+1,1,3,3));//tg+1 : voir placement
         sommet = 0 ;
         palIm = new JLabel[tailleGraph] ;
         Font gf = (new JLabel(vide)).getFont().deriveFont(Font.BOLD,30) ;
         /* Initialisation des cellules vides  */
         for(int i = 0 ;i<tailleGraph  ;i++ ) {
            palIm[i] = new JLabel(vide, SwingConstants.CENTER) ;
            palIm[i].setFont(gf);
```

```
        }//fin du for
        /* Initialisation de l'affichage avec l'etat de la pile en commencant par le fond */
        Enumeration en = obs.getAsVector().elements() ;
        while (en.hasMoreElements()) {
            palIm[tailleGraph -1 - sommet].setText(en.nextElement().toString()) ;
            sommet++ ;
        }//fin du while
        /* Placement des cellules dans la grille */
        this.add(new JLabel(vide)) ; // pour l'esthetique : un espace au dessus
        for(int i = 0 ;i<tailleGraph  ;i++ ) {
            this.add(palIm[i]) ;
        }//fin du for
    } // fin du constructeur
```

**Q9**. Écrire la méthode update(Observable obs, Object arg) de PanelVisuTab. La tester en utilisant le fichier TestFrameRobotTab.java fourni.

Questions facultatives :
**Q10.** Écrire une classe FenetreVisu dans laquelle on peut placer un PanelVisu ou un PanelNext. Tester une version du robot où le bouton et les piles sont dans 4 fenêtres séparées.

**Q11**. Écrire un programme où le robot trie successivement deux piles distinctes sur deux destinations distinctes, mais en utilisant la même pile "temporaire" et le même bouton.

**Q12**. Écrire un programme dans lequel le robot boucle : dès qu'il a fini, il re-tire au sort une pile sur "source" et la trie.