Résumé : Le but de ce TP est de manipuler les piles et les files.

1 Labyrinthe

Nous allons implémenter un algorithme d'exploration de labyrinthe. Dans ce TP, nous considérons uniquement des labyrinthes carrés, composés de cellules qui sont soit franchissables (un passage), soit infranchissables (un mur). Chaque cellule est identifiée par un couple dentiers (x,y) où x est le numéro de colonne (les colonnes sont numérotées de gauche à droite à partir de 0), et y est le numéro de ligne (les lignes sont numérotées de haut en bas à partir de 0). Les cellules du bord sont infranchissables sauf celles situées en haut à gauche (coordonnées (0,1)) et en bas à droite. Ceux sont respectivement les cellules d'entrée et de sortie du labyrinthe. Les cellules franchissables explorées seront marquées dune petite pierre rouge. Une pierre rose identifiera les cellules déjà visitées sur lesquelles on retourne pour visiter leurs voisins non encore explorés. L'idée est que l'on marque les cellules visitées pendant le parcours afin de garder en mémoire les chemins déjà visités. Pour que vous commenciez directement par implémenter les piles, un programme permettant de créer et dessiner un labyrinthe vous est donné.
Copiez la classe `Labyrinthe.java` (que vous trouverez dans dans les ressources sur le site web) dans le répertoire des sources de votre projet Eclipse `tp3`. Il est inutile de lire ce fichier ; pour l'utiliser, il vous suffit de savoir qu'il contient une classe `Labyrinthe` définissant les méthodes suivantes :

```
// le constructeur cree une fenetre et affiche le labyrinthe dedans
Labyrinthe()
// est-ce que (x,y) est un mur ?
boolean estMur(int x, int y)
// la taille du labyrinthe, a la fois nb lignes et nb colonnes
int n()
// teste si la cellule (x,y) comporte une marque
boolean estMarque(int x, int y)
// pose une marque
void poserMarque(int x, int y)
// pose une marque retour
void poserMarqueRetour(int x, int y)
```

Q 1. Cellule
Pour parcourir le labyrinthe, vous utiliserez une pile de cellules. Une cellule comprend les 2 coordonnées x et y. Écrivez une classe `Cellule` avec les deux attributs x et y ainsi qu'un constructeur qui permet de les initialiser.
Q 2. Parcours
Maintenant, nous allons explorer le labyrinthe à proprement parler. Nous allons utiliser une pile de cellules pour cela, au moyen de la classe java `Stack`. La pile représente le chemin vers l'arrivée, et permet de conserver en mémoire ce chemin (s'il existe !). On ajoute une cellule dans la pile si elle est une case voisine de la `Cellule` courante, qu'elle n'est pas un mur et qu'elle nest pas marquée (si elle lest cela signifie quon a déjà testé le chemin passant par cette cellule). Dès que l'on insère une `Cellule` dans la pile, on la marque, afin de se souvenir qu'on est passé par là. On retire une cellule différente de l'arrivée de la pile lorsqu'on est sûr qu'elle ne mènera pas à la sortie, c'est-à-dire si tous ses voisins sont soit marqués, soit des murs. Écrire une classe `Parcours` avec une méthode main qui implémente l'algorithme suivant :

```
Créer un labyrinthe l
Créer une pile p
Ajouter la cellule de départ dans p
Marquer la cellule de départ // pour éviter de ressortir par l'entrée
Tant que p n'est pas vide {
    soit c la cellule du debut de la pile
    la marquer comme retour
    si c est la case d'arrivée {
        Afficher "Sortie trouvée"
    }
    sinon {
        pour chacune des 4 cases voisines de c
        si une n'est ni un mur ni une case marquée{
           la marquer et l'empiler
        }
        si tous les voisins sont marqués, on dépile
    }
}
```

```
Afficher "Il n'y a pas de chemin reliant l'entrée à la sortie" si il n'y pas de chemin
```

L'affichage se fait de manière instantanée, car nos machines sont trop rapides. Vous pouvez ralentir la boucle en y insérant l'instruction suivante.

```
try {
  Thread.sleep(10) ;
}
catch(InterruptedException e){}
```

Bonus : vous pouvez modifier le labyrinthe dans le fichier `Labyrinthe.java` pour voir le comportement de votre programme quand la sortie n'est pas atteignable, en bouchant la sortie par un mur.

Q 3. Autre Pile

Écrivez une classe `Pile` qui implémente une pile avec un tableau. Elle contiendra des objets de type `Cellule`. Retestez la classe `Parcours` en utilisant votre classe `Pile`. Elle devrait se comporter exactement de la même manière, sans que vous ayez changé quoi que ce soit dans Parcours.

Q 4. File

Vous pourriez remplacer la Pile par une File. Le changement de comportement du parcours est alors intéressant à observer. Modifiez `Parcours` pour qu'une file soit utilisée à la place d'une pile. Utilisez les méthodes de l'interface `Queue` : `offer` pour ajouter en queue de liste et `poll` qui supprime et retourne l'objet en tête de liste. Vous avez le choix de la classe qui implémente `Queue` dans la bibliothèque Java.

Q 5. Autre File

S'il vous reste encore du temps, vous pouvez cette fois-ci implémenter une file à partir de tableau (cf. TD) et l'utiliser à la place de l'interface `Queue`. Vérifiez que le comportement est toujours similaire.