TP 2
Tableaux

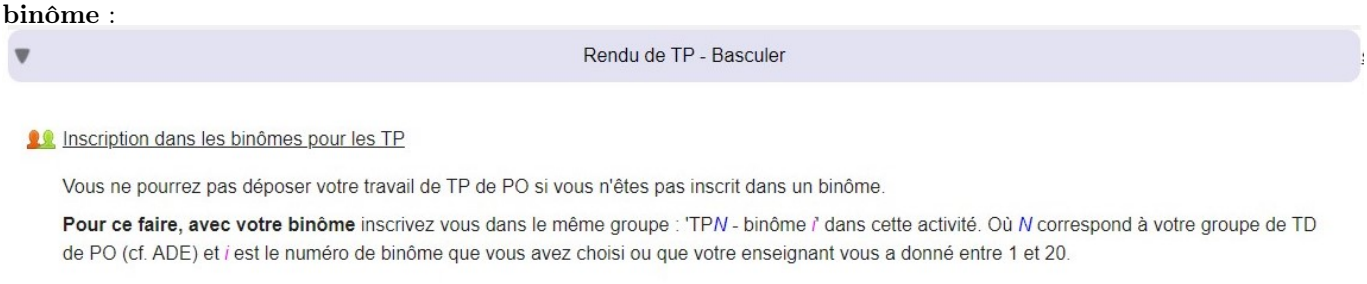
1 Contenu et objectifs des TPs

1.1 Objectif général :

L'objectif de ce TP est de manipuler les tableaux en JAVA. Pour ce faire vous allez développer des méthodes permettant de compléter une grille de Sudoku.

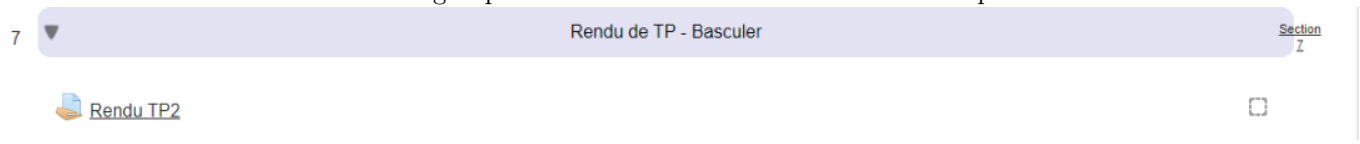
1.2 Rendu attendu :

Ce TP est un TP évalué. La remise du TP s'effectuera sur Moodle. **Vous devez commencer par enregistrer votre binôme** :



Choisissez tous les deux un groupe encore libre parmi ceux préfixés par votre groupe de TP, ex :

Les groupes sont **strictement limités à deux membres** par groupe. Une fois le groupe créé, vous pourrez rendre votre devoir avec votre identifiant de groupe. Le rendu s'effectue sur l'activité correspondante :



Date de remise du projet : **29/10/2021 à 23h59 max**.

1.3 Le jeu du Sudoku

Pour rappel ou information, une grille de Sudoku (*cf.* figure 1.3) se définit comme suit (d'après Wikipédia) : « La grille de jeu est un carré de neuf cases de côté, subdivisé en autant de carrés identiques, appelés régions (voir figure). La règle du jeu est simple : chaque ligne, colonne et région ne doit contenir qu'une seule fois tous les chiffres de un à neuf. Formulé autrement, chacun de ces ensembles doit contenir tous les chiffres de un à neuf. »

Problème

	3							6
			7		2	3		
1		4		3	8			
3				2		8	1	
9	1	8				2	6	5
	6	2		5				7
			1	4		7		8
		1	2		9			
8							2	

Solution

2	3	7	5	9	4	1	8	6
5	8	6	7	1	2	3	4	9
1	9	4	6	3	8	5	7	2
3	7	5	9	2	6	8	1	4
9	1	8	4	7	3	2	6	5
4	6	2	8	5	1	9	3	7
6	2	3	1	4	5	7	9	8
7	4	1	2	8	9	6	5	3
8	5	9	3	6	7	4	2	1

FIGURE 1 – Exemple de grille de Sudoku

On veut donc réaliser un programme fournissant un peu d'aide dans la découverte d'une solution.

2 Modélisation JAVA

Avant de commencer ce TP, on créera un répertoire de travail et on y copiera le fichier `Sudoku.java` disponible sur Moodle. Vous trouverez dans ce fichier, la structure de base que nous avons choisie d'utiliser, *i.e.* (c'est à dire) la modélisation choisie pour la grille de Sudoku, modélisation que nous vous expliquons ci-après.

Pour modéliser un plateau de Sudoku, on utilisera un tableau d'entiers à 2 dimensions, dont les éléments peuvent valoir 0 (la case n'a pas encore de valeur fixée), 1, . . . , 9.

```
static final int n = 3 ; // taille des régions int[][]
unPlateau = new int [n*n][n*n] ;
```

Dans le programme `Sudoku.java`, on dispose :

— D'une fonction d'initialisation d'une grille à partir d'une chaîne de caractères :

```
static int [][] aPartirDe (String s) {
/*
* Prérequis : s est une chaîne contenant au moins
* n^4 chiffres décimaux
* Résultat : un plateau de sudoku initialisé avec les n^4
* premiers chiffres décimaux de s (les chiffres sont considérés
* comme rangés par lignes).
*/
```

— D'une fonction de visualisation d'une grille :

```
static String enClair (int [][] m) {
/*
* Résultat : une chaîne dont l'affichage permet de visualiser m
* en tant que plateau de sudoku
*/
```

Ainsi, la séquence ci-dessous :

```
String grille1 =
"000 402 000 \n" +
"030 000 090 \n" +
"902 000 105 \n" +
" \n" +
"000 010 000 \n" +
"500 603 002 \n" +
"020 000 070 \n" +
" \n" +
"408 901 507 \n" +
"090 000 040 \n" +
"003 705 900 " ;
System.out.println(enClair(aPartirDe(grille1)));
```

provoquera l'affichage de :

```
0 0 0 4 0 2 0 0 0
0 3 0 0 0 0 0 9 0
9 0 2 0 0 0 1 0 5

0 0 0 0 1 0 0 0 0
5 0 0 6 0 3 0 0 2
0 2 0 0 0 0 0 7 0

4 0 8 9 0 1 5 0 7
0 9 0 0 0 0 0 4 0
0 0 3 7 0 5 9 0 0
```

3 Vérification sur les lignes, colonnes et régions

Pour placer une valeur dans une cellule de la grille, il faut d'abord vérifier que cette valeur n'est présente ni dans la ligne, ni dans la colonne, ni dans la région de la cellule. Pour faire cela, rédiger et tester les 3 fonctions de vérification suivantes :

Exercice 1. Vérification sur une ligne :

```
static boolean presentLigne (int [][] m, int v, int i) {
/*
* Prérequis :
* - v est compris entre 1 et n^2
* - i est compris entre 0 et n^2-1
* Résultat : indique si v est présent dans la ligne i de m
*/
    return true ; // A MODIFIER
} // presentLigne
```

Exercice 2. Vérification sur une colonne :

```
static boolean presentColonne (int [][] m, int v, int j) {
/*
* Prérequis :
* - v est compris entre 1 et n^2
* - j est compris entre 0 et n^2-1
* Résultat : indique si v est présent dans la colonne i de m
*/
    return true ; // A MODIFIER
} // presentColonne
```

Exercice 3. Vérification sur une région :

```
static boolean presentRegion (int [][] m, int v, int i, int j) {
/*
* Prérequis :
* - v est compris entre 1 et n^2
* - i et j sont compris entre 0 et n^2-1
* Résultat : indique si v est présent dans la région contenant
* la case <i, j>
* */
    return true ; // A MODIFIER
} // presentRegion
```

4 Étude d'une place

Étant donné un plateau `m`, on cherche à déterminer l'ensemble des valeurs qui pourraient être affectées à une case `<i, j>`. Pour représenter un tel ensemble on utilisera un tableau `r` de booléens, l'idée étant que si `r[v]` vaut `true` cela signifie que `v` est une valeur possible pour `m[i][j]`.

Exemple :

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
r	?	false	true	true	false	false	true	false	false	false

Dans cet exemple, l'ensemble des valeurs décrites par `r` est `{2, 3, 6}` ; `r[0]` n'est pas significatif.

Exercice 4. Rédiger et tester la fonction :

```
static boolean [] lesPossiblesEn (int [][] m, int i, int j) {
    /*
    * Prérequis :
    * - i et j sont compris entre 0 et n^2-1
    * - m[i][j] vaut 0
    * Résultat : un tableau r de longueur n^2+1 tel que,
    * dans la tranche r[1..n^2], r[v] indique
    * si v peut être placé dans m en <i, j> selon les règles du
    sudoku
    * */
    return null ; // A MODIFIER
}
```

Exercice 5. Lorsque l'ensemble des valeurs affectables à une case est réduit à un élément il est clair qu'on a trouvé l'unique valeur qui puisse convenir pour cette case. Aussi, pour vérifier cette unicité, rédiger et tester la fonction suivante :

```
static int toutSeul (int [][] m, int i, int j) {
    /*
    * Prérequis :
    * - i et j sont compris entre 0 et n^2-1
    * - m[i][j] vaut 0
    * Résultat :
    * - v si la seule valeur possible dans m en <i, j> est v
    * selon les règles du sudoku
    * - -1 dans les autres cas
    * */
    return -1 ; // A MODIFIER
}
```

5 Simplification d'une grille

Exercice 6. Trouver une valeur unique pour une cellule ne suffit pas, il faut maintenant essayer de remplir toutes les cases n'ayant pas de valeur intiale. Pour ce faire, rédiger et tester la fonction suivante :

```
static void essais (String grille) {
    /*
    * Prérequis : grille représente une grille de sudoku
    * (les chiffres sont considérés comme rangés par lignes)
    *
    * Effet :
    * 1) affiche en clair la grille
    * 2) affecte, tant que faire se peut, toutes les cases pour
    * lesquelles il n'y a qu'une seule possibilité
    * 3) affiche en clair le résultat final
    */ int[][] m =
    aPartirDe(grille);
    System.out.println("Problème\n\n"+enClair(m));
    // A COMPLETER
    System.out.println("Il se peut qu'on ait avancé\n\n"+enClair(m));
}
```

Remarque : On conseille d'afficher, à chaque fois qu'une case est modifiée, les coordonnées de la case et la valeur affectée.

6 Un peu de stylistique

On peut vouloir s'intéresser à des plateaux de taille différente, par exemple sur un plateau de taille 4x4, en jouant les chiffres de 1 à 4 (les régions font 2×2) ou un plateau 16×16 en augmentant les nombres possibles (les régions font 4×4).

Exercice 7. Déterminer combien de lignes de votre programme doivent être modifiées pour traiter de tels problèmes.

Vous écrirez votre réponse dans un commentaire à la fin du fichier.