R1.01.P2 – Traces d'un programme et Opérations simples sur les tableaux – TD de la semaine 45 et TP des semaines 45 et 46

Objectifs du TD/TP

- Traces de l'exécution d'un programme
- Algorithmique simple sur des tableaux d'entiers

Présentation du TD

Lorsqu'une exécution de code (peu importe le langage) se passe mal, il y a principalement 2 problèmes :

- Soit il y a un bug (plantage) d'exécution et l'interpréteur (ou l'exécution du binaire) s'arrête brusquement avec, pour certains langages, très peu d'explications pour justifier cet arrêt.
- Soit le programme est sémantiquement incorrecte : il ne fait pas ce qu'il faut (le résultat obtenu n'est pas le bon), il boucle à l'infini etc.

Une solution pour comprendre l'exécution d'un programme, c'est de le tracer. Tracer une exécution c'est examiner **pour chaque instruction** le contenu des données. Bien sûr, il n'est pas possible d'examiner 1000 lignes de code. Par contre, tracer une exécution sur une portion de code (celle qui pose problème) est indispensable.

L'outil que nous utilisons dans ce TD est le tableau des traces (présenté en cours). Un tableau des traces (qui vous est fourni et que vous devrez remplir) se compose de 5 colonnes :

- *Num. lgn code* : le numéro de la ligne dans l'éditeur correspondant à l'instruction que vous devez détailler.
- *Instruction* : l'instruction qui se trouve au numéro de ligne ci-dessus et que vous devez détailler.
- *Remarque* : une remarque éventuelle à ajouter concernant cette instruction.
- *Contenu des variables <u>après</u> exécution de l'instruction* : chacune des 5 colonnes correspond à une variable (5 variables maximum). Comme titre de la colonne, on donne le nom de la variable et pour chaque ligne d'instruction on donne pour chaque variable un contenu. Par convention, une variable qui n'est pas encore déclarée contiendra **« inexistant »** et une variable qui est déclarée mais qui n'est pas encore initialisée contiendra **« indéterminé »**. Attention, une variable de type « tableau » contient une adresse vers la première case du tableau (notation : *adr1 –> (1)* où (1) désigne une structure de type tableau).
- *Contenu des tableaux si nécessaire* : si certaines variables contiennent des adresses vers des tableaux (*adr1 –> (1)* par exemple) alors dans cette case on dessinera chaque tableau et le contenu de chaque case du tableau.

Exercice 1

Soit le code Java ci-après écrit dans la méthode *principal()*.

A. En supposant une exécution complète de la méthode *principal()*, donnez à l'aide du tableau des traces (document fourni) pour chaque instruction l'état (contenu) de toutes vos variables. A titre indicatif, il y a environ 18 lignes d'instruction à expliciter.

B. Quel est au final le contenu des tableaux pointés par *tab2* et *tab3* ?

```
3.   void principal() {
4.
5.      int[] tab1, tab3;
6.      int[] tab2 = {23, 14, -9};
7.      int i, taille;
8.
9.      taille = tab2.length;
10.     tab1 = new int[taille];
11.
12.     i = taille - 1;
13.
14.     while ( i >= 0 ) {
15.
16.        tab1[taille-i-1] = tab2[i];
17.        i--;
18.     }
19.
20.     tab3 = tab1;
21.     tab1 = null;
22.  }
```

Exercice 2

Soit le code Java ci-après écrit dans les méthodes *principal()* et *doSomething(...)*.

A. En supposant une exécution complète des méthodes *principal()* et *doSomething(...)*, donnez à l'aide du tableau des traces (document fourni) pour chaque instruction l'état (contenu) de toutes vos variables. A titre indicatif, il y a environ 16 lignes d'instruction à expliciter.

B. Quel est au final le contenu de la variable *valX* ?

```
3.   void principal() {
4.
5.      int[] tab1 = {-2, 12, 8};
6.      int valX;
7.
8.      valX = doSomething ( tab1 );
9.
10.
11.  }
12.
13.  int doSomething ( int[] theT ) {
14.
15.     int tmp;
16.     int i = 0;
17.
18.     while ( i < (theT.length-1) ) {
19.
20.        if ( theT[i+1] < theT[i] ) {
21.
22.           tmp = theT[i+1];
23.           theT[i+1] = theT[i];
24.           theT[i] = tmp;
25.        }
26.
27.        i++;
28.     }
29.
30.     return theT[theT.length-1];
31.  }
```

Présentation du TP

Ce TP ce prépare lors de la 2ème séance de TD.

Ce TP se code en Java sur 2 X 1h30. Le rendu se fera impérativement sur Moodle la semaine 46, le **samedi 16/11 à 23h55 au + tard**. Les modalités de ce rendu sont expliquées en détail sur l'espace des rendus créé à cet effet : voir *https://moodle.univ-ubs.fr/course/view.php?id=7518*.

Attention, un **malus** sera appliqué sur la note de TP : *-2* points pour tout retard de rendu (même d'une minute !). Le nommage du fichier sera impérativement *Nom_Prenom.zip* (PAS de fichier *.rar*).

Méthodes à développer

La classe *SimplesTableau* contient 14 méthodes d'opérations simples sur un tableau d'entiers. **Aucune variable globale** ne sera utilisée (excepté éventuellement des constantes). Chaque méthode codée doit être testée par une autre méthode qui se chargera d'appeler la méthode développée pour la tester **dans tous les cas de figures** (cas normal, cas limites, cas d'erreurs). Ainsi, si une méthode *truc(...)* est développée alors on créera une méthode de test *void testTruc()* qui mettra à l'épreuve la méthode *truc(...)* dans tous les cas de figures. **L'automatisation des tests** se fera nécessairement par appel de la méthode ***void testCasTruc(…)*** qui prend en paramètres les données du test et le résultat attendu.

En R1.01.P1, le test des méthodes se limitait aux cas normaux.

En R1.01.P2, on rajoute 2 autres types de cas pour le test des méthodes :

- les cas limites : cas où l'on teste la méthode (l'algorithme) sur des données particulières pour vérifier que ces données ne « plantent » pas votre algorithme (donc pas d'erreurs générées). Exemple : un algorithme de tri doit se comporter normalement sur un tableau à 1 case.
- les cas d'erreurs : cas où la méthode (l'algorithme) refusera de s'exécuter car les données sur lesquelles il doit travailler ne sont pas valides. Exemple : un algorithme de tri ne peut pas travailler sur un tableau qui n'existe pas (adresse *null*). La gestion d'un cas d'erreur dans la méthode se fera obligatoirement de la manière suivante :

```
// Affichage de l'explication courte de l'erreur précédé du nom de la méthode dans laquelle se
// produit l'erreur

System.err.println ( "Nom de la méthode" + "Explication courte de l'erreur" ) ;
```

Bien entendu, comme en R1.01.P1, on essayera d'automatiser les tests au maximum.

L'ordre de développement des méthodes est important et est le suivant :

1. *verifTab*
2. *afficherTab*
3. *tirerAleatoire*
4. *remplirAleatoire*
5. *egalite*
6. *copier*
7. *leMax*
8. *inverse*
9. *echange*

10. *melange*
11. *decalerGche*
12. *supprimerUneValeur*
13. *inclusion*
14. *remplirToutesDiff*

Le détail du rôle de chaque méthode, de même que le détail des paramètres d'entrée et de sortie est présenté dans le document en annexe intitulé « *javaDoc* de la classe *SimplesTableau* ».

Partie TP

Mise en place de l'environnement de développement sous Linux (ou Windows)

Sous linux (windows) et dans votre arborescence personnelle, créer les répertoires *Info/Prog/R1.01.P2/TPs/TP_SimplesTableau* (le répertoire *R1.01.P2* doit apparaître au même niveau que le répertoire *R1.01.P1*). Nous aurons donc un répertoire par *TP* à rendre (trois au total).

En dessous du répertoire *R1.01.P2/TPs/TP_SimplesTableau*, créer 4 sous-répertoires : */ws*, */src*, */class*, */javaDoc*, dont le rôle est :

- pour */ws* : répertoire de travail, utilisé pour exécuter les commandes de compilation et d'exécution,
- pour */class* : de ranger les fichiers compilés (*.class*),
- pour */src* : de ranger les fichiers sources (*.java*),
- pour */javaDoc* : de ranger les pages *html* extraites de la documentation embarquée.

Pour permettre aux outils *Java* de retrouver les fichiers compilés, la variable d'environnement *CLASSPATH* devra être définie et initialisée une fois pour toute. Pour cela, sous **linux** (sous windows, voir les explications dans le fichier *javaCLASSPATH* sur Moodle *R1.01.P2*) :

- éditer le fichier *.bashrc* pour déclarer la variable *CLASSPATH* et l'initialiser :
  - *cd* (vous êtes à la racine de votre home)
  - *geany .bashrc* (point bashrc !)
  - rajouter à la suite dans ce fichier la ligne de texte : *export CLASSPATH=../class*
- retourner en *...R1.01.P2/TPs/TP_SimplesTableau/ws*
- forcer la création de la variable en tapant *source ~/.bashrc*
- pour vérifier que la variable *CLASSPATH* est correctement initialisée, taper : *echo $CLASSPATH* (en réponse, l'affichage à l'écran doit être *../class*)

Première compilation et exécution

Comme en *R1.01.P1*, les classes *Start* et *SimpleInput* doivent se trouver dans le répertoire */class* (voir le lien de téléchargement *StartEtSimpleInput* sur *Moodle R1.01.P2*).

La première étape consiste à créer le fichier *SimplesTableau.java* : dans */ws*, lancer l'éditeur de code de votre choix (*geany*, *atom*, *visual studio code*, *emacs* par exemple). Ensuite, cliquez sur « Nouveau » dans l'éditeur pour créer le fichier *SimplesTableau.java* qui devra impérativement être enregistré dans le répertoire *../src*.

Le squelette du fichier *java* à développer doit **impérativement** respecter la structure suivante :

```
/**
  * JavaDoc : rôle de votre classe (programme), à compléter…
  */
class SimplesTableau {


   /**
     * JavaDoc : point d'entrée de l'exécution
     */
   void principal() {

      // appel des cas de test un par un

      testVerifTab();

      etc.

   }

   /**
     * JavaDoc : rôle de la méthode
     */
   boolean verifTab( int[] leTab, int nbElem ) {

      // code complet de la méthode, à écrire…

      // cette méthode sera testée par « testVerifTab() »

      // à compléter…

   }


   /**
     * JavaDoc : test de la méthode verifTab
     */
   void testVerifTab() {

      // déroulement complet du test de la méthode « verifTab(…) »

      // à compléter…

      // appel de la méthode void testCasVerifTab ( … ) autant de fois que nécessaire

   }

}
```

Pour un premier test de compilation et d'exécution, écrivez simplement les méthodes *verifTab(…)* et *testVerifTab()*. On se rappellera qu'en *Java*, la création d'un tableau d'entiers initialise toutes ses cases à zéro.

Pour compiler : rester dans */ws* et taper *javac –d ../class ../src/SimplesTableau.java*

Pour exécuter : rester dans */ws* et taper *java Start SimplesTableau*

Suite et fin

Pour chaque méthode développée, il sera nécessairement associé une méthode qui la teste. Ainsi, le code de la méthode *principal()* est un code d'enchaînement des cas de test :

```
void principal() {

   // appel des cas de test un par un
   testVerifTab();
   testAfficherTab();
   testTirerAleatoire();
   testRemplirAleatoire();
   testEgalite();
   testCopier();
   testLeMax();
   testInverse();
   testEchange();
   testMelange();
   testDecalerGche();
   testSupprimerUneValeur();
   testInclusion();
   testRemplirToutesDiff();

}
```

Chaque méthode de test doit mettre à l'épreuve la méthode à tester dans 3 familles de cas différents : les cas normaux, les cas limites et les cas d'erreurs. Au minimum, nous devons trouver la famille des cas normaux.

Pour ne pas alourdir inutilement le code de ces méthodes de test, nous vous demandons de tester le cas d'erreur **trivial** (*i.e.* le tableau n'existe pas et/ou le nombre d'éléments, *nbElem*, n'est pas compatible avec la taille) pour une méthode ou l'autre **mais pas systématiquement**. Il sera plus intéressant de tester les cas d'erreurs non triviaux.

La JavaDoc

Pour terminer, il vous est également demandé d'écrire la *javaDoc* de chacune des méthodes et de la générer. Pour générer cette *javaDoc*, dans */ws* tapez la commande :

javadoc –encoding UTF8 –private –d ../javaDoc ../src/SimplesTableau.java

Corrigez les warnings si il y en a et vérifiez l'existence des pages *html* générées dans */javaDoc*.

Il est possible d'utiliser les caractères < (+ petit) et > (+ grand) dans cette *javaDoc*. Par contre, le générateur des pages *html* peut éventuellement se mettre en erreur (ça dépendra de la version de *java* que vous utilisez) car il interprète ces symboles comme des balises *html*. Pour corriger le problème si il apparaît, remplacer < par ***<*** et > par ***>***

ANNEXE : JavaDoc de la classe SimplesTableau

Class SimplesTableau

java.lang.Object
    SimplesTableau

```
public class SimplesTableau
extends java.lang.Object
```

Cette classe effectue des opérations élémentaires sur un ou plusieurs tableaux d'entiers. La taille d'un tableau est par définition le nombre TOTAL de cases = tab.length. Un tableau d'entiers créé possède nbElem éléments qui est nécessairement inférieur ou égal à la taille du tableau : nbElem <= tab.length (= taille).

Author:

J-F. Kamp - octobre 2022

Constructor Summary

Constructors

Constructor	Description
SimplesTableau()	

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
(package private) void	afficherTab(int[] leTab, int nbElem)	Affiche le contenu des nbElem cases d'un tableau une par une.
(package private) int[]	copier(int[] tabToCopy, int nbElem)	Renvoie la copie exacte (clone) du tableau passé en paramètre.
(package private) void	decalerGche(int[] leTab, int nbElem, int ind)	Décale de une case de la droite vers la gauche toutes les cases d'un tableau à partir d'un indice "ind" et jusque nbElem-1 ([ind]<-[ind+1]<-[ind+2]<-...<-[nbElem-2]<-[nbElem-1]).
(package private) void	echange(int[] leTab, int nbElem, int ind1, int ind2)	Echange les contenus des cases du tableau passé en paramètre, cases identifiées par les indices ind1 et ind2.
(package private) boolean	egalite(int[] tab1, int[] tab2, int nbElem1, int nbElem2)	Renvoie vrai si les 2 tableaux passés en paramètre sont exactement les mêmes en nombre d'éléments et en contenu (case par case).
(package private) boolean	inclusion(int[] tab1, int[] tab2, int nbElem1, int nbElem2)	Renvoie vrai ssi le tableau tab1 est inclus dans tab2.
(package private) int[]	inverse(int[] leTab, int nbElem)	Renvoie un nouveau tableau qui est l'inverse de celui passé en paramètre.
(package private) int	leMax(int[] leTab, int nbElem)	Renvoie le maximum parmi les éléments du tableau.
(package private) int[]	melange(int[] leTab, int nbElem)	Retourne un nouveau tableau qui a la même taille et les mêmes occurrences d'élements que le tableau passé en paramètre mais ces éléments sont répartis selon des indices aléatoires (0 <= indice <= nbElem-1).
(package private) void	principal()	Le point d'entrée du programme.

(package private) void	**remplirAleatoire** (int[] leTab, int nbElem, int min, int max)	A partir d'un tableau créé, remplir aléatoirement ce tableau de nbElem valeurs comprises entre min et max.
(package private) void	**remplirToutesDiff** (int[] leTab, int nbElem)	A partir d'un tableau déjà créé, remplir le tableau de nbElem valeurs saisies par l'utilisateur.
(package private) int	**supprimerUneValeur** (int[] leTab, int nbElem, int valeur)	Supprime du tableau la première case rencontrée dont le contenu est égale à "valeur".
(package private) int	**tirerAleatoire**(int min, int max)	Renvoie un entier aléatoire compris entre min et max (min <= valeur <= max).
(package private) boolean	**verifTab**(int[] leTab, int nbElem)	Factorisation : appelé chaque fois qu'une vérification doit se faire sur la validité d'un tableau.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SimplesTableau

```
public SimplesTableau()
```

Method Detail

principal

```
void principal()
```

Le point d'entrée du programme.

afficherTab

```
void afficherTab(int[] leTab, int nbElem)
```

Affiche le contenu des nbElem cases d'un tableau une par une. Tenir compte des cas d'erreurs concernant leTab et nbElem (appel de verifTab).

Parameters:

`leTab` - le tableau à afficher

`nbElem` - le nombre d'entiers que contient le tableau

egalite

```
boolean egalite(int[] tab1, int[] tab2, int nbElem1, int nbElem2)
```

Renvoie vrai si les 2 tableaux passés en paramètre sont exactement les mêmes en nombre d'éléments et en contenu (case par case). Tenir compte des cas d'erreurs concernant tab1, nbElem1 et tab2, nbElem2 (appel de verifTab 2 fois) et afficher un éventuel message d'erreur (et dans ce cas renvoyer faux).

Parameters:

`tab1` - le 1er tableau à comparer

`tab2` - le 2ème tableau à comparer

`nbElem1` - le nombre d'entiers présents dans le 1er tableau

`nbElem2` - le nombre d'entiers présents dans le 2ème tableau

Returns:
true si égalité parfaite sinon false

remplirAleatoire

```
void remplirAleatoire(int[] leTab, int nbElem, int min, int max)
```

A partir d'un tableau créé, remplir aléatoirement ce tableau de nbElem valeurs comprises entre min et max. Tenir compte des cas d'erreurs concernant leTab et nbElem (appel de verifTab). Vérifier que min <= max, sinon afficher également une erreur.
Utiliser obligatoirement la méthode "int tirerAleatoire (int min, int max)".

Parameters:
`leTab` - le tableau à remplir de valeurs tirées aléatoirement

`nbElem` - le nombre d'entiers que contiendra le tableau

`min` - la valeur de l'entier minimum

`max` - la valeur de l'entier maximum

tirerAleatoire

```
int tirerAleatoire(int min, int max)
```

Renvoie un entier aléatoire compris entre min et max (min <= valeur <= max). Vérifier que min <= max et min >= 0, sinon afficher une erreur (et dans ce cas retourner -1).

Parameters:
`min` - la valeur de l'entier minimum

`max` - la valeur de l'entier maximum

Returns:
l'entier aléatoire

copier

```
int[] copier(int[] tabToCopy, int nbElem)
```

Renvoie la copie exacte (clone) du tableau passé en paramètre.

Parameters:
`tabToCopy` - le tableau à copier

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:
le nouveau tableau qui est la copie du tableau passé en paramètre

leMax

```
int leMax(int[] leTab, int nbElem)
```

Renvoie le maximum parmi les éléments du tableau.

Parameters:
`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:
le maximum des éléments du tableau

inverse

`int[] inverse(int[] leTab, int nbElem)`

Renvoie un nouveau tableau qui est l'inverse de celui passé en paramètre. Son jème élément est égal au (nbElem+1-j) élément du tableau initial (dans l'explication, j=1 signifie premier élément du tableau).

Parameters:

`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:

le nouveau tableau qui est l'inverse de leTab sur la plage (0...nbElem-1)

echange

`void echange(int[] leTab, int nbElem, int ind1, int ind2)`

Echange les contenus des cases du tableau passé en paramètre, cases identifiées par les indices ind1 et ind2. Vérifier que les indices ind1 et ind2 sont bien compris entre zéro et (nbElem-1), sinon afficher un message d'erreur.

Parameters:

`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

`ind1` - numéro de la première case à échanger

`ind2` - numéro de la deuxième case à échanger

melange

`int[] melange(int[] leTab, int nbElem)`

Retourne un nouveau tableau qui a la même taille et les mêmes occurrences d'élements que le tableau passé en paramètre mais ces éléments sont répartis selon des indices aléatoires (0 <= indice <= nbElem-1). Une technique simple consiste à utiliser les méthodes "echange" et "tirerAleatoire" pour effectuer le mélange.

Parameters:

`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:

le nouveau tableau qui a le même contenu que le tableau initial mais mélangé

decalerGche

`void decalerGche(int[] leTab, int nbElem, int ind)`

Décale de une case de la droite vers la gauche toutes les cases d'un tableau à partir d'un indice "ind" et jusque nbElem-1 ([ind]<-[ind+1]<-[ind+2]<-...<-[nbElem-2]<-[nbElem-1]). Vérifier que ind est compris entre 0 et (nbElem-2) sinon afficher une erreur.

Parameters:

`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

`ind` - l'indice à partir duquel commence le décalage à gauche

supprimerUneValeur

`int supprimerUneValeur(int[] leTab, int nbElem, int valeur)`

Supprime du tableau la première case rencontrée dont le contenu est égale à "valeur". La case du tableau est supprimée par décalage à gauche des cases du tableau. **L'appel de la méthode "decalerGche" est obligatoire**. A l'issue de la suppression (si elle existe) le nombre d'éléments dans le tableau est décrémenté et retourné.

Parameters:

`leTab` - le tableau

`nbElem` - le nombre d'entiers présents dans le tableau

`valeur` - le contenu de la première case à supprimer

Returns:

le nombre d'éléments dans le tableau (éventuellement inchangé)

inclusion

```
boolean inclusion(int[] tab1, int[] tab2, int nbElem1, int nbElem2)
```

Renvoie vrai ssi le tableau tab1 est inclus dans tab2. Autrement dit, si tous les éléments de tab1 se retrouvent intégralement dans tab2 (y compris les doublons) mais pas nécessairement dans le même ordre. L'utilisation de méthodes déjà écrites est autorisé.

Parameters:

`tab1` - le premier tableau

`tab2` - le deuxième tableau

`nbElem1` - le nombre d'entiers présents dans le tableau1

`nbElem2` - le nombre d'entiers présents dans le tableau2

Returns:

vrai ssi tableau1 est inclus dans tableau2

remplirToutesDiff

```
void remplirToutesDiff(int[] leTab, int nbElem)
```

A partir d'un tableau déjà créé, remplir le tableau de nbElem valeurs saisies par l'utilisateur. Au fur et à mesure de la saisie, si la nouvelle valeur saisie existe déjà dans le tableau alors ne pas l'insérer et demander de ressaisir. Tenir compte des cas d'erreurs concernant leTab et nbElem (appel de verifTab).

Parameters:

`leTab` - le tableau à remplir d'éléments tous différents

`nbElem` - le nombre d'entiers que contiendra le tableau

verifTab

```
boolean verifTab(int[] leTab, int nbElem)
```

Factorisation : appelé chaque fois qu'une vérification doit se faire sur la validité d'un tableau. Le tableau n'est pas valide si celui-ci est inexistant ou si le nombre d'éléments qu'il contiendra est incompatible avec sa taille (leTab.length).

Parameters:

`leTab` - le tableau dont on doit vérifier l'existence (leTab différent de null)

`nbElem` - le nombre d'éléments que contiendra le tableau, doit vérifier la condition 0 < nbElem <= leTab.length

Returns:

vrai ssi le tableau est valide