Résumé Dans ce TP, vous implémenterez les tris vus en TD sous Eclipse et vous vérifierez leur bon fonctionnement grâce au framework de test JUnit.

1 Environnement de développement Java

1.1 Création dun projet Java sous Eclipse

1 Créez un répertoire `sdd` à la racine de votre répertoire personnel (homedir) et lancez Eclipse. Vérifiez les points suivants : paramétrez Eclipse de manière à ce qu'il soit considéré comme espace de travail (workspace) sous Eclipse (sinon c'est le répertoire workspace qui sera utilisé par défaut). C'est dans ce répertoire que vous développerez vos travaux pratiques.
2 Créez un nouveau projet Java `tp1`. Cela aura pour effet de créer un répertoire `tp1` dans votre répertoire `sdd`. Vérifiez que les sous-répertoires `src` et `bin` ont bien été créés dans `tp1`. Le premier accueillera les fichiers sources (.java) et le second, les fichiers de bytecode Java (.class) correspondants.
3 Ajoutez un nouveau **répertoire de sources** nommé `tests` au même niveau darborescence que `src`. Ce dernier contiendra vos fichiers sources de test.
4 Copiez la classe `Sort` fournie sur le site dans votre répertoire `src`.

1.2 Configuration de tests unitaires

1 Dans l'explorateur de paquets, faites un clic droit sur la classe `Sort`.
2 Dans le menu contextuel, cliquez sur New JUnit Test Case.
3 Choisissez le répertoire de sources `tests` et nommez la classe `SortTest`. Vérifiez que la classe sous test est bien `Sort`. Cliquez sur *next* ne générez pas les signatures des méthodes de tests (elles vous sont fournies) et créez la classe.
4 Eclipse va proposer d'ajouter la bibliothèque de JUnit dans le *buildpath* du projet, acceptez.
5 Remplacer le contenu de la classe SortTest fraîchement créée par celui de classe SortTest fournie dans les ressources du TP sur Moodle.

2 Implémentation pilotée par les tests

Les tests ont été écrits pour vous. Vous allez devoir implémenter les méthodes de la classe Sort qui contiennent les algorithmes de tris vu en TD ainsi que des méthodes auxiliaires. Après chaque écriture de méthode, vous vérifierez que le test associé considère votre implémentation valide en faisant un clic droit sur la classe de test puis en choisissant *Run As - JUnit test* dans le menu contextuel.

Question 1 :
Écrivez une méthode `static void printArray(int [] tab)` qui affiche le contenu d'un tableau d'entiers. Cette méthode n'est soumise à aucun test mais vous servira si vous souhaitez vérifier les contenus de vos tableaux après tris. A l'avenir vous pourrez utiliser la méthode `toString` de la bibliothèque `Arrays`.

Question 2 :
Écrivez une méthode `static int [] generateRdmIntArray(int n, int min, int max)` qui retourne un tableau d'entiers de taille n générés aléatoirement dans l'intervalle entre min et max. Pour rappel la méthode `Math.random()` retourne un `double` dans l'intervalle [0.0,1.0[.

Question 3 :
Implémentez les méthodes de tri suivantes vues en TD :

```
static void bubbleSort(int [] tab)
static void selectSort(int [] tab)
static void insertSort(int [] tab)
```

3 Utilisations de compteurs

Nous souhaitons maintenant compter le nombre de comparaisons et de permutations effectuées par les algorithmes de tris. Pour ce faire, nous utiliserons une classe `Counter` qui contiendra deux attributs déclarés privés : `comp` pour compter le nombre de comparaisons effectuées, et `perm` pour le nombre de permutations d'éléments.

Question 6 :
Écrivez une classe `Counter`. Ajoutez des méthodes pour incrémenter chacun des deux attributs, de 1 puis de n (méthodes incComp(), incComp(int n), incPerm() et incPerm(int i))

Question 7 :
Redéfinissez la méthode `toString` de manière à afficher "(nombre de comparaisons, nombre de permutations)" et ajoutez une méthode `reset` qui remet les attributs à 0.

Question 8 : Utilisation de la classe Counter
Réécrivez vos méthodes de tri de façon à accepter un argument supplémentaire à vos fonctions :
```
static void bubbleSort(int[] tab, Counter c)
static void selectSort(int[] tab, Counter c)
static void insertSort(int [] tab, Counter c)
```
Lors de l'appel à ces fonctions, le compteur c sera incrémenté en conséquence lors des comparaisons entre éléments et déplacements d'éléments.

Question 9 :
A l'aide de vos compteurs, comparer le nombre de comparaisons et de permutations des différents algorithmes de tris implémentés. Comparez ces résultats avec les nombres théoriques vus en TD.

4 Tri d'objets

Il n'y a pas de relation d'ordre naturelle entre deux objets, comme il peut y en avoir une entre les entiers. C'est le cas des chaînes de caractères vues en TD pour lesquelles l'implémentation de la méthode `compareTo` de l'interface `Comparable` permet cette comparaison. Vous savez sans doute que les collections Java comme les listes implémentent une méthode `sort` qui permet trier de manière optimisée ses éléments. Comme elle ne peut "deviner" la relation d'ordre entre deux objets, elle se base sur le résultat de la méthode `compareTo`. La méthode `sort` de la bibliothèque `Arrays` permet également de trier des tableaux d'objets.

Question 10 :
Créez une classe `Friend` avec deux attributs `name` et `age` et son constructeur. Cette classe doit implémenter l'interface `Comparable`.

Questions 11 :
Créez plusieurs instances de la classe `Friend` dans une méthode `main` avec différents noms et âges. Implémentez la méthode `compareTo` de manière à ce que la méthode `Arrays.sort()` de la bibliothèque Java trie votre tableau d'objets de type Friend dans l'ordre croissant des âges.

Pour rappel, si l'objet sur lequel la méthode `compareTo` est appelée est considéré comme inférieur à l'objet en argument, la méthode renvoie un entier négatif, 0 s'ils sont égaux et un entier positif s'il est plus grand.

5 Pour aller plus loin...

Questions 12 :

Écrivez la méthode public `public int partition(int [] tab, int debut, int fin, int pivot)` qui partitionne le tableau `tab` entre les indices `debut` et `fin`. Les valeurs de la partition du début de tableau sont les valeurs inférieures à la valeur pivot et les valeurs de l'autre partition sont supérieures ou égales à la valeur `pivot`. La méthode retourne l'indice de la dernière valeur de la partition de début (celles des valeurs strictement inférieures à `pivot`). Attention, on ne demande pas que les partitions soient elles-mêmes triées, on ne cherche ici qu'à partitionner le tableau.

Questions 13 :
Le tri rapide ou Quicksort est un des algorithmes de tri les plus performants, il a été inventé par **Tony Hoare** en 1961. Documentez-vous et réfléchissez à l'implémentation d'un tri rapide à partir de la méthode `partition`.