TP Java n°1

- N'oubliez les principes indiqués en cours : l'essentiel n'est pas de coder, mais de **bien** coder - et cela passe par une phase de réflexion. Indentez correctement votre code (2 ou 4 espaces, pas de tabulations)… Pour cela, configurez correctement votre éditeur de texte.
- Bien que ce ne soit pas précisé dans les sujets des exercices, les solutions doivent mettre en œuvre les spécificités du langage (pas de C/C++/Python/Go traduit en Java) et votre code doit être décomposé en sous-programmes (donc en méthodes…).
- Pour ce TP, **n'utilisez que les concepts et les types vus dans les cours précédents**. Notamment, n'utilisez pas les classes collections de Java. Chaque exercice doit être implémenté par une classe contenant une fonction `main` et chaque sous-programme sera implémenté par une méthode **statique**.
- *Pour tous les exercices de ce TP, supposez dans un premier temps que l'utilisateur saisit bien un entier (il vous restera simplement à tester qu'il appartient à l'intervalle voulu). Ensuite, ajoutez la gestion des exceptions si nécessaire.*
- Implémentez chaque exercice comme une classe distincte, appartenant à un paquetage `tp1` .

Exercice 1

Un *diviseur propre* d'un nombre entier positif *N* est un diviseur de *N* < *N*. Les diviseurs propres de 6, par exemple, sont 1, 2 et 3. On peut prouver que les diviseurs propres de *N* sont inférieurs ou égaux à $N/2$.

Un nombre entier positif est dit *parfait* s'il est égal à la somme de ses diviseurs propres. 6 est donc un nombre parfait.

Écrire un programme qui demande un nombre entier strictement supérieur à 2 à l'utilisateur et qui affiche si ce nombre est parfait ou non.

- Pour cet exercice et les suivants, on aura besoin de calculer la somme des diviseurs propres d'un nombre. Commencez donc par écrire une fonction qui effectue ce calcul et qui sera appelée dans le programme principal.
- On peut aussi s'arrêter à $\sqrt{N}$… Si N = 28, par exemple, on recherchera les diviseurs entre 2 et 5… (au lieu de 2 et 14). On trouvera donc 2 (et il faut alors ajouter aussi 28/2 = 14) et 4 (on ajoute aussi 28/4 = 7). La somme finale est donc 1 + 2 + 14 + 4 + 7 = 28, ce qui indique que 28 est parfait… On note que ce calcul est bien plus rapide qu'en s'arrêtant à N/2 car on fait beaucoup moins de boucles.

Exercice 2

Écrire un programme prenant une limite strictement supérieure à 2 en paramètre et affichant la liste des nombres parfaits inférieurs ou égaux à cette limite (le programme doit vérifier la validité de son appel).

SHELL

```
$ java Parfaits 1000
6 28 496
```

Exercice 3

Écrire un programme qui affiche tous les couples de nombres amis inférieurs à une borne saisie au clavier. Deux nombres sont amis si la somme des diviseurs propres de l'un est égale à l'autre et réciproquement.

SHELL

```
$ java Amis
Entrez une limite > 2 : 10000
(220,284) (1184,1210) (2620,2924) (5020,5564) (6232,6368)
```

Exercice 4

Écrire un programme qui demande un nombre à l'utilisateur et qui affiche si ce nombre est plus petit ou plus grand qu'un nombre secret choisi par le programme.

Le programme gère trois niveaux : facile, moyen et difficile correspondant à un nombre de tentatives maximum (plus c'est difficile, moins on a de tentatives).

Après chaque jeu, l'utilisateur a le choix de rejouer (en changeant éventuellement de niveau) ou de quitter (si on rejoue, il faut évidemment que le programme utilise un autre nombre secret…). Vous utiliserez une méthode (qu'il vous appartient de trouver dans la documentation de l'API) pour que le programme choisisse un nombre secret aléatoire compris entre 1 et une limite fixée par une constante du programme (*indice : aléatoire se dit random en anglais*).

Si l'utilisateur ne trouve pas le secret dans le nombre de coups imparti, le programme affiche le nombre qu'il fallait trouver. Si l'utilisateur trouve le secret, le programme affiche le nombre de tentatives de l'utilisateur.

SHELL

```
$ java Secret
Choisir un niveau (0 : simple, 1 : moyen, 2 : difficile) 0
Entrez un nombre compris entre 1 et 100 : 50
C'est moins...
Entrez un nombre compris entre 1 et 100 : 20
C'est moins...
Entrez un nombre compris entre 1 et 100 : 10
C'est moins...
Entrez un nombre compris entre 1 et 100 : 5
C'est moins...
Entrez un nombre compris entre 1 et 100 : 2
C'est plus...
Entrez un nombre compris entre 1 et 100 : 3
Bravo, vous avez trouvé en 6 coups
Une autre partie (o/n) : o
Choisir un niveau (0 : simple, 1 : moyen, 2 : difficile) 2
Entrez un nombre compris entre 1 et 100 : 99
C'est moins...
Entrez un nombre compris entre 1 et 100 : 98
C'est moins...
Entrez un nombre compris entre 1 et 100 : 97
C'est moins...
Entrez un nombre compris entre 1 et 100 : 96
C'est moins...
Entrez un nombre compris entre 1 et 100 : 95
C'est moins...
Il fallait trouver 42
Une autre partie (o/n) : n
Au revoir...
```

Exercice 5

La méthode de Héron est décrite sur cette page (https://fr.wikipedia.org/wiki/Méthode_de_Héron#principe) et permet de calculer la racine carrée d'un nombre par approximations successives.

Écrire le programme **Heron.java** qui devra avoir la forme suivante :

JAVA

```
public class Heron {
    /** Renvoie la racine carrée de nbre à la précision indiquée */
    static double heron(int nbre, double precision) {
      (...)
    }

    /** Renvoie la racine carrée de nbre à la précision 0.001 */
    static double heron(int nbre) {
        return heron(nbre, 0.001);
    }
    public static void main(String[] args) {
        // assert heron(2, 0.001) == Math.sqrt(2);           // Doit échouer !!!
        assert (Math.abs(heron(2, 0.01) - Math.sqrt(2)) <= 0.01);
    assert heron(1) == 1 && heron(0) == 0 ;
    }
}
```

Pour que les assertions soient vérifiées par Java, vous devez exécuter le programme avec l'option **-ea** (*enable assertions*).