TP 4

L. Naert – T. Ferragut – E. Lemonnier

2 octobre 2024

Objectifs du TP

- Utiliser des méthodes
- Écrire des méthodes
- Écrire des méthodes de tests

On considerera dans ce TP que les tests de la valeurs des paramètres sont fait avant l'appel de la méthode et non dans la méthode elle-même. Aucun test d'erreurs n'est à réaliser dans les méthodes.

Exercice 1 (reprise du TD)

En mathématique, le nombre de combinaisons de k éléments parmi n s'écrit C_n^k et est défini pour $0 \leq k \leq n$ par :

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Avec $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times ... \times n$ et $0! = 1$.

1. Écrire les deux méthodes suivantes :

```
/**
 * calcul de la factorielle du paramètre
 * @param n valeur de la factoriel à calculer, n>=0
 * @return factorielle de n
 */
int factorielle (int n)

/**
 * calcul de la combinaison k parmi n
 * @param n cardinalité de l'ensemble
 * @param k nombre d'éléments dans n avec 0<=k<=n
 * @return nombre de combinaisons de k parmi n
 */
int combinaison (int n, int k)
```

2. Tester la méthode `factorielle()` en utilisant la méthode `testFactorielle()` :

```
/**
 * Teste la méthode factorielle()
 */
void testFactorielle () {
    System.out.println ();
    System.out.println ("*** testFactorielle()");

    testCasFactorielle (5, 120);
    testCasFactorielle (0, 1);
    testCasFactorielle (1, 1);
    testCasFactorielle (2, 2);
}

/**
 * teste un appel de factorielle
 * @param n valeur de la factorielle à calculer
 * @param result resultat attendu
 */
void testCasFactorielle (int n, int result) {
    // Affichage
    System.out.print ("factorielle (" + n + ") \t= " + result + "\t : ");

    // Appel
    int resExec = factorielle(n);

    // Verification
    if (resExec == result){
        System.out.println ("OK");
    } else {
        System.err.println ("ERREUR");
    }
}
```

3. Sur le modèle de `testFactorielle()`, écrire la méthode `testCombinaison()` et `testCasCombinaison()`

   Rendre le code du programme et les tests d'exécution.

4. **Bonus :** calculer `combinaison(25,24)`. Comment expliquer le résultat obtenu ? (mettre l'explication sur le compte-rendu)
5. **Bonus :** programmer, tester une méthode `combinaison` qui permette de passer outre le problème mis en évidence dans la question précédente.

   Rendre le code modifié et les tests.

Exercice 2 (*)

1. Écrire une méthode `estDiviseur(int p, int q)` qui rend vrai si q divise p, faux sinon.

   Par exemple, 2 est diviseur de 10 car 2*5 vaut 10, par contre 3 n'est pas diviseur de 10.

   ```
   /**
    * teste la divisibilité de deux entiers
    * @param p entier positif à tester pour la divisibilité
    * @param q diviseur strictement positif
    * @return vrai ssi q divise p
    */
   boolean estDiviseur (int p, int q)
   ```

2. Sur le modèle de l'exercice 1, écrire la méthode `testEstDiviseur()` et `testCasEstDiviseur()` qui testent la méthode `estDiviseur()`.

   Rendre le code des méthodes `estDiviseur()`, `testEstDiviseur()` et `testCasEstDiviseur()` et de l'exécution des tests.

Exercice 3 (**)

Un nombre parfait est un nombre entier tel que la somme de ses diviseurs (autres que lui-même) est égale à ce nombre. Par exemple, 6 est un nombre parfait car $6 = 1 + 2 + 3$ et que 1, 2 et 3 sont les diviseurs de 6. De même, 28 est aussi un nombre parfait car $28 = 1 + 2 + 4 + 7 + 14$. Les trois premiers nombres parfaits sont 6, 28, 496.

1. En utilisant la méthode `estDiviseur()`, écrire la méthode `estParfait()` :

   ```
   /**
    * teste si un nombre est parfait
    * @param a entier positif
    * @return vrai ssi a est un nombre parfait
    */
   boolean estParfait (int a)
   ```

2. Écrire les méthodes `testEstParfait()` et `testCasEstParfait()` qui testent la méthode `estParfait()`.

   Rendre le code de `estParfait()`, de `testEstParfait()`, de `testCasEstParfait()` et l'éxécution.

3. Écrire une méthode `QuatreNbParfaits()` qui affiche les quatre premiers nombres parfaits.

   ```
   /**
    * Affiche les quatre premiers nombres parfaits
    */
   void quatreNbParfaits(){
   ```

   Rendre le code de `QuatreNbParfaits()`, et le résultat de l'exécution.

Exercice 4 (**)

1. Écrire une méthode `testEstCroissant()` et une méthode `testCasEstCroissant()` qui testent la méthode :

```
/**
 * teste si les valeurs d'un tableau sont triées par ordre croissant
 * @param t tableau d'entiers
 * @return vrai ssi les valeurs du tableau sont en ordre croissant
 */
boolean estCroissant (int[] t)
```

2. Écrire le code de la méthode `estCroissant()`

   Rendre le code de `estCroissant()`, de `testEstCroissant()`, de `testCasEstCroissant()` et le résultat de l'exécution du test.

Exercice 5 (**)

1. Écrire une fonction qui cherche combien de fois un caractère est présent dans une chaîne de caractères (`String`). Le caractère à chercher et la chaîne seront passés en paramètres.

```
/**
* cherche combien de fois un caractère est présent dans une chaîne de caractères
* @param chaine Chaine de caractère
* @param car Caractere a rechercher
* @return nombre d'occurences de car dans chaine
*/
int nbOcc (String chaine, char c)
```

2. Écrire une méthode `testNbOcc()` et une méthode `testCasNbOcc()` qui testent la méthode `nbOcc()`.

   Rendre le code de `nbOcc()`, de `testNbOcc()`, de `testCasNbOcc()` et le résultat de l'exécution du test.