



Résumily | Université de Boumerdes

Document

Solution Série n°2 2022/2023

Module

Programmation Orientée Objet

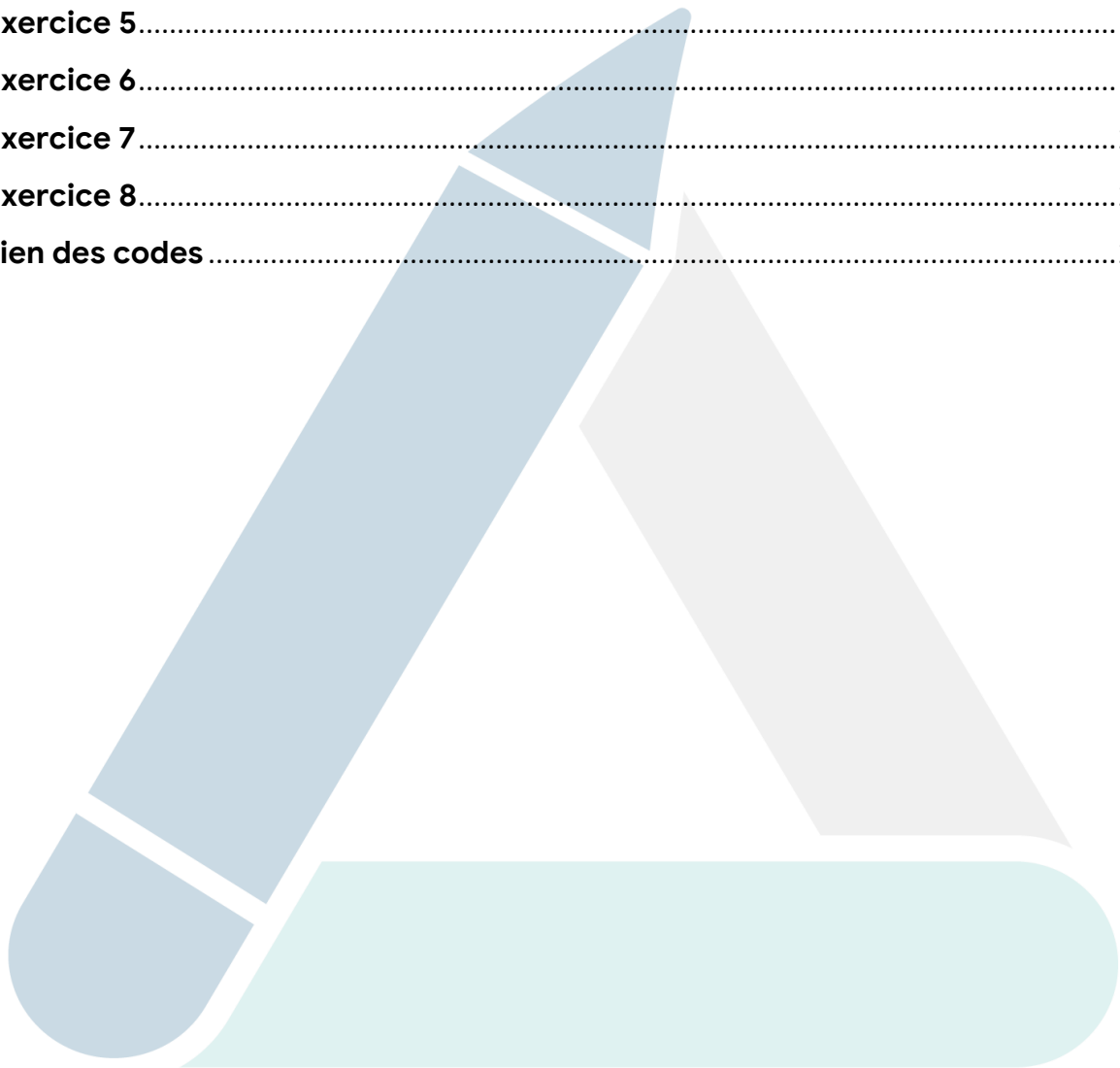
Réalisé par

Résumily



SOMMAIRE

Introduction	3
Exercice 1	4
Exercice 2	8
Exercice 3	12
Exercice 4	15
Exercice 5	17
Exercice 6	19
Exercice 7	21
Exercice 8	23
Lien des codes	26



INTRODUCTION

Bienvenue dans ce document qui contient la solution détaillée de la série 2 du module de Programmation Orientée Objet sur les concepts de base sur Java de l'année universitaire 2022/2023. Ce document a été créé par l'équipe **Résumily** afin de vous aider à consolider votre compréhension des concepts de base de la programmation orientée objet en Java et de vous fournir des solutions claires et détaillées pour les exercices proposés dans cette série.

Nous espérons que ce document vous aidera à renforcer vos compétences en programmation orientée objet et nous vous souhaitons une excellente lecture !

[Énoncé de la Série n°2](#)



Cliquez sur l'icône

EXERCICE 1

- Analyse théorique du code :
 - ✓ La variable **n** est initialisée à 0 et la variable **k** est initialisée à 1.
 - ✓ La boucle **do-while** s'exécute au moins une fois car **k** est initialisé à 1.
 - ✓ Si **n** est pair, le message "n est pair" est affiché, **n** est augmenté de 3 et la boucle continue.
 - ✓ Si **n** est un multiple de 3, le message "n est multiple de 3" est affiché, **n** est augmenté de 5 et la boucle continue.
 - ✓ Si **n** est un multiple de 5, le message "n est multiple de 5" est affiché, suivi d'un saut de ligne, et la boucle s'arrête avec l'instruction **break**.
 - ✓ Si aucun des tests précédents n'est vrai, **n** est simplement augmenté de 1 et la boucle continue.
 - ✓ À la fin de la boucle, la valeur finale de **n** est affichée.

Testons maintenant ce code sur Eclipse pour voir si notre analyse est correcte.

Voici le code :

```
public class exo01_code1 {  
  
    public static void main(String[] args) {  
  
        int n = 0, k = 1;  
        do {  
            if (n % 2 == 0) {  
                System.out.println(n + " est pair");  
                n += 3;  
                continue;  
            }  
            if (n % 3 == 0) {  
                System.out.println(n + " est multiple de 3");  
                n += 5;  
            }  
            if (n % 5 == 0) {  
                System.out.println(n + " est multiple de 5\n");  
                break;  
            }  
            n += 1;  
        } while (k == 1);  
        System.out.println("La valeur finale de n à la sortie de la boucle  
est " + n);  
    }  
}
```

Le résultat attendu est :

0 est pair

3 est multiple de 3

9 est multiple de 3

15 est multiple de 3

20 est multiple de 5

La valeur finale de n est : 20

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$ java exo01_code1.java
0 est pair
3 est multiple de 3
9 est multiple de 3
15 est multiple de 3
20 est multiple de 5

La valeur finale de n à la sortie de la boucle est 20
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$
```

Le résultat obtenu correspond bien à notre analyse théorique, donc notre réponse est correcte.

- Si on remplace k par 0 dans le code fourni on aura comme résultat :

0 est pair

La valeur finale de n à la sortie de la boucle est 3

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$ java exo01_code1.java
0 est pair
La valeur finale de n à la sortie de la boucle est 3
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$
```

➤ En remplaçant la boucle **do-while** par **while** :

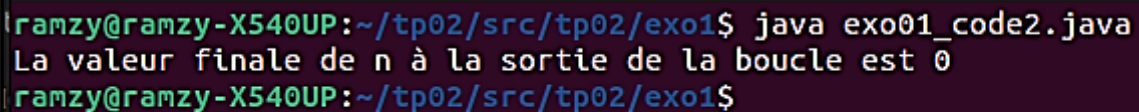
Si nous initialisons la valeur de **k** à 0, la boucle ne sera jamais exécutée car la condition de la boucle **while** sera fausse dès le départ. Le code affichera simplement "La valeur finale de n à la sortie de la boucle est 0".

Voici le code modifié :

```
public class exo01_code2 {  
  
    public static void main(String[] args) {  
        int n = 0, k = 0;  
        while (k == 1) {  
            if (n % 2 == 0) {  
                System.out.println(n + " est pair");  
                n += 3;  
                continue;  
            }  
            if (n % 3 == 0) {  
                System.out.println(n + " est multiple de 3");  
                n += 5;  
            }  
            if (n % 5 == 0) {  
                System.out.println(n + " est multiple de 5\n");  
                break;  
            }  
            n += 1;  
        }  
        System.out.println("La valeur finale de n à la sortie de la boucle  
est " + n);  
    }  
}
```

Le résultat obtenu est simplement :

La valeur finale de n à la sortie de la boucle est 0



```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$ java exo01_code2.java  
La valeur finale de n à la sortie de la boucle est 0  
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo1$
```

Par conséquent, la boucle **do-while** s'exécute au moins une fois, tandis que la boucle **while** ne s'exécutera pas du tout si la condition est initialement fausse. La différence entre les deux boucles est donc que la boucle **do-while** s'exécute au

moins une fois, tandis que la boucle **while** peut ne pas s'exécuter du tout si la condition est initialement fausse.

Tips and tricks :

- Voici quelques astuces générales sur l'utilisation de **do-while** et **while** :
- La boucle **while** est utile lorsque vous voulez exécuter une boucle tant qu'une condition est vraie. Si la condition n'est jamais vraie, la boucle ne sera jamais exécutée.
- La boucle **do-while** est similaire à la boucle **while**, mais elle garantit que le bloc de code à l'intérieur de la boucle est exécuté au moins une fois, peu importe la valeur de la condition.
- Il est important de s'assurer que la condition de la boucle est mise à jour correctement dans le bloc de code à l'intérieur de la boucle, sinon vous risquez de créer une boucle infinie.
- L'utilisation des instructions **continue** et **break** peut aider à contrôler l'exécution de la boucle. **Continue** permet de sauter une itération de la boucle et d'aller directement à la suivante, tandis que **break** permet de sortir de la boucle.

EXERCICE 2

- Voici le programme Java pour déterminer la nième valeur de la suite de Fibonacci de manière itérative :

```
import java.util.Scanner;

public class exo02_iteratif {
    public static void main(String[] args) {

        int n, n1 = 1, n2 = 1, n3 = 0;
        Scanner sc = new Scanner(System.in);

        do {
            System.out.print("Entrez un nombre entier supérieur à 2 : ");
            n = sc.nextInt();
        } while (n <= 2);

        for (int i = 2; i <= n; i++) {
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
        }

        System.out.println("La " + n + "ème valeur de la suite de
        Fibonacci est : " + n3);
    }
}
```

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo2$ java exo02_iteratif.java
Entrez un nombre entier supérieur à 2 : 8
La 8ème valeur de la suite de Fibonacci est : 34
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo2$
```

Le programme commence par demander à l'utilisateur d'entrer un nombre entier supérieur à 2. Si l'utilisateur entre un nombre inférieur ou égal à 2, le programme redemandera à l'utilisateur de saisir une nouvelle valeur.

Ensuite, le programme utilise une boucle **for** pour calculer la nième valeur de la suite de Fibonacci. Le premier et le deuxième terme de la suite sont initialisés à 1, et

le troisième terme est calculé en additionnant les deux termes précédents. Les termes suivants sont calculés de la même manière, jusqu'à ce que la *n*ème valeur soit atteinte.

- Voici maintenant le programme Java pour déterminer la *n*ème valeur de la suite de Fibonacci de manière récursive :

```
import java.util.Scanner;

public class exo02_recuratif {
    public static void main(String[] args) {

        int n;
        Scanner sc = new Scanner(System.in);

        do {
            System.out.print("Entrez un nombre entier supérieur à 2 : ");
            n = sc.nextInt();
        } while (n <= 2);

        int n3 = fib(n);

        System.out.println("La " + n + "ème valeur de la suite de
        Fibonacci est : " + n3);
    }

    public static int fib(int n) {
        if (n < 2) {
            return 1;
        } else {
            return fib(n-1) + fib(n-2);
        }
    }
}
```

Le programme commence également par demander à l'utilisateur d'entrer un nombre entier supérieur à 2, et redemande une nouvelle valeur si l'utilisateur entre un nombre inférieur ou égal à 2.

Ensuite, le programme utilise une méthode récursive pour calculer la *n*ème valeur de la suite de Fibonacci. La méthode prend en entrée le nombre *n* et renvoie le *n*ème terme de la suite de Fibonacci. Si *n* est inférieur ou égal à 2, la méthode renvoie 1. Sinon, elle renvoie la somme des deux termes précédents de la suite de Fibonacci (calculés en appelant récursivement la méthode ***fib***).

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo2$ java exo02_recuratif.java
Entrez un nombre entier supérieur à 2 : 5
La 5ème valeur de la suite de Fibonacci est : 8
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo2$
```

Notez que la méthode récursive peut être moins efficace que la méthode itérative pour de grandes valeurs de n , car elle utilise beaucoup de mémoire et effectue des appels de méthode récursive.

Tips and tricks :

- Avant de commencer à écrire une fonction, définissez clairement ce que la fonction doit faire et ce que ses entrées et sorties doivent être.
 - N'oubliez pas de déclarer le type de retour de la fonction avant son nom. Si la fonction ne retourne rien, utilisez le type "void".
 - Les noms de fonction devraient être des verbes ou des phrases verbales qui décrivent clairement ce que la fonction fait. Par exemple, dans les deux exemples de code donnés, le nom "fib" pour la fonction qui calcule les valeurs de la suite de Fibonacci est un choix judicieux car il décrit clairement l'action de la fonction.
 - Les fonctions peuvent prendre des paramètres. Déclarez-les entre les parenthèses juste après le nom de la fonction. Si plusieurs paramètres sont nécessaires, séparez-les par des virgules.
 - Les fonctions doivent être conçues pour être réutilisables et génériques autant que possible, pour minimiser la duplication de code.
 - Les fonctions récursives peuvent être utiles pour des problèmes mathématiques tels que le calcul de la suite de Fibonacci, mais elles peuvent également être gourmandes en ressources si elles ne sont pas correctement optimisées. Dans ces cas-là, une approche itérative peut être plus efficace.
- La récursivité :

- ✓ Comprendre le concept de récursivité : une fonction récursive est une fonction qui s'appelle elle-même. Il est important de bien comprendre comment cela fonctionne et comment les appels récursifs sont effectués pour éviter les boucles infinies.
- ✓ Avoir une condition d'arrêt : il est important d'avoir une condition d'arrêt dans une fonction récursive, c'est-à-dire une condition qui arrête l'appel récursif et renvoie une valeur.
- ✓ Éviter les appels récursifs inutiles : il est important d'éviter les appels récursifs inutiles, car cela peut entraîner des performances médiocres ou même des erreurs de dépassement de pile. Il faut donc s'assurer que chaque appel récursif a un but et contribue à la résolution du problème.
- ✓ Utiliser la récursivité pour résoudre des problèmes récursifs : la récursivité est utile pour résoudre des problèmes qui ont une structure récursive. Par exemple, les problèmes de tri, les problèmes de recherche et les problèmes de parcours de graphes sont souvent résolus de manière récursive.

EXERCICE 3

```
import java.util.Arrays;
import java.util.Scanner;

public class exo03 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Demande la taille des tableaux à l'utilisateur
        int taille;
        do {
            System.out.print("Entrez la taille des tableaux : ");
            taille = sc.nextInt();
        } while (taille <= 0);

        // Alloue de la mémoire pour les trois tableaux
        int[] t1 = new int[taille];
        int[] t2 = new int[taille];
        int[] t3 = new int[taille];

        // Remplit t1 avec les valeurs saisies par l'utilisateur
        for (int i = 0; i < taille; i++) {
            System.out.print("Entrez la valeur " + (i+1) + " : ");
            t1[i] = sc.nextInt();
        }

        // Calcule les carrés des nombres dans t1 et les place dans t2
        for (int i = 0; i < taille; i++) {
            t2[i] = t1[i] * t1[i];
        }

        // Affiche les carrés dans t2 en utilisant la boucle for-each
        System.out.print("Les carrés des valeurs dans t1 sont : ");
        for (int val : t2) {
            System.out.print(val + " ");
        }
        System.out.println();

        // Remplit t3 avec des valeurs aléatoires entre 5 et 99
        for (int i = 0; i < taille; i++) {
            t3[i] = (int) (Math.random() * 95) + 5;
        }

        // Trie les éléments de t3 dans l'ordre croissant
        Arrays.sort(t3);

        // Affiche les valeurs dans t3 en utilisant la boucle for-each
        System.out.print("Les valeurs dans t3 triées dans l'ordre
croissant sont : ");
        for (int val : t3) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

```
// Affiche la valeur maximale de t3 en utilisant la méthode max
de la classe Arrays
int max = Arrays.stream(t3).max().getAsInt();
System.out.println("La valeur maximale de t3 est : " + max);

// Affiche la valeur minimale de t3 en utilisant la méthode min
de la classe Arrays
int min = Arrays.stream(t3).min().getAsInt();
System.out.println("La valeur minimale de t3 est : " + min);

sc.close(); // Fermer le scanner et libérer les ressources
associées à ce scanner.

}
}
```

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo3$ java exo03.java
Entrez la taille des tableaux : 5
Entrez la valeur 1 : 3
Entrez la valeur 2 : 2
Entrez la valeur 3 : 7
Entrez la valeur 4 : 1
Entrez la valeur 5 : 4
Les carrés des valeurs dans t1 sont : 9 4 49 1 16
Les valeurs dans t3 triées dans l'ordre croissant sont : 20 48 58 64 95
La valeur maximale de t3 est : 95
La valeur minimale de t3 est : 20
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo3$
```

Notez que le programme utilise la classe `Arrays` pour trier les éléments de `t3` et trouver les valeurs maximale et minimale de `t3`. Ces méthodes sont très pratiques et évitent de devoir écrire notre propre code pour ces opérations.

Tips and tricks :

- **Arrays** est une classe Java qui fournit des méthodes pour travailler avec des tableaux. Certaines méthodes utiles incluent `sort` pour trier un tableau dans l'ordre croissant, `max` et `min` pour trouver la valeur maximale et minimale d'un tableau, et `toString` pour obtenir une chaîne de caractères représentant le tableau. Assurez-vous de connaître les différentes méthodes disponibles dans la classe **Arrays** et comment les utiliser.

- La méthode **Math.random** retourne un nombre aléatoire compris entre 0 et 1. Vous pouvez utiliser cette méthode pour générer des valeurs aléatoires dans votre code. Par exemple, pour générer un nombre aléatoire compris entre 5 et 99, vous pouvez multiplier le résultat de **Math.random** par 95 (la différence entre 99 et 5) et ajouter 5 (la valeur minimale).
- L'allocation dynamique vous permet de créer des tableaux dont la taille est déterminée à l'exécution plutôt qu'à la compilation. En Java, vous pouvez utiliser l'opérateur **new** pour allouer de la mémoire pour un tableau. Par exemple, `int[] t1 = new int[taille];` crée un tableau d'entiers de taille `taille`. Assurez-vous de valider la taille saisie par l'utilisateur pour éviter les erreurs de mémoire.
- Pour parcourir un tableau, vous pouvez utiliser une boucle **for** classique ou une boucle **for-each**. La boucle **for-each** est utile lorsque vous n'avez pas besoin de connaître l'indice de chaque élément du tableau. Par exemple, `for (int val : t2)` parcourt chaque élément de `t2` et stocke sa valeur dans la variable `val`.

EXERCICE 4

```
public class exo04 {
    public static void main(String[] args) {
        int[] occurrences = new int[11];

        for (int i = 0; i < 70000; i++) {
            int dice1 = (int) (Math.random() * 6) + 1;
            int dice2 = (int) (Math.random() * 6) + 1;
            int sum = dice1 + dice2 - 2;

            occurrences[sum]++;
        }

        for (int i = 0; i < occurrences.length; i++) {
            int sum = i + 2;
            System.out.println("Somme " + sum + " : " + occurrences[i] +
                " occurrences");
        }

        int maxOccurrences = 0;
        int maxSum = 0;
        for (int i = 0; i < occurrences.length; i++) {
            int sum = i + 2;
            if (occurrences[i] > maxOccurrences) {
                maxOccurrences = occurrences[i];
                maxSum = sum;
            }
        }

        System.out.println("La somme la plus fréquente est " + maxSum +
            " avec " + maxOccurrences + " occurrences.");
    }
}
```

Ce code utilise un tableau d'entiers occurrences pour stocker les occurrences de chaque somme possible. Il utilise ensuite une boucle for pour simuler 70000 lancers de deux dés et mettre à jour le tableau d'occurrences en fonction de chaque lancer.

Ensuite, le code utilise une deuxième boucle for pour afficher les résultats du nombre d'occurrences de chaque somme possible. Le code conserve également la somme la plus fréquente et son nombre d'occurrences dans les variables **mostFrequentSum** et **maxOccurrence**.

Enfin, le code affiche la somme la plus fréquente ainsi que son nombre d'occurrences.

Le chiffre obtenu pour la somme la plus fréquente dépendra de la simulation. En général, la somme la plus fréquente devrait être autour de 7, car il y a plus de combinaisons possibles pour obtenir cette somme que pour les autres sommes.

Cependant, il est possible que le résultat soit différent en raison de la nature aléatoire de la simulation.

Notez que la méthode **Math.random()** renvoie un nombre aléatoire compris entre 0 et 1. Pour simuler le lancer d'un dé à 6 faces, on peut multiplier ce nombre par 6 et ajouter 1 (pour obtenir une valeur entre 1 et 6).

Lorsque le code est compilé et exécuté, il affiche :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo4$ java exo04.java
Somme 2 : 1935 occurrences
Somme 3 : 3981 occurrences
Somme 4 : 5804 occurrences
Somme 5 : 7741 occurrences
Somme 6 : 9838 occurrences
Somme 7 : 11614 occurrences
Somme 8 : 9590 occurrences
Somme 9 : 7713 occurrences
Somme 10 : 6042 occurrences
Somme 11 : 3797 occurrences
Somme 12 : 1945 occurrences
La somme la plus fréquente est 7 avec 11614 occurrences.
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo4$
```

Tips and tricks :

- Utilisation d'un tableau pour stocker les occurrences : Le tableau est une structure de données très pratique pour stocker des occurrences, car il permet de stocker facilement et efficacement des valeurs en utilisant un indice. Dans ce code, le tableau `occurrences` est utilisé pour stocker le nombre d'occurrences de chaque somme de deux dés.
- Utilisation de **Math.random()** pour générer des nombres aléatoires : La méthode **Math.random()** est une méthode pratique pour générer des nombres aléatoires. Elle renvoie un **double** compris entre 0 (inclus) et 1 (exclus). Pour générer un nombre aléatoire compris entre **a** et **b**, on peut utiliser la formule : **(int) (Math.random() * (b-a+1)) + a**.
- Trouver la somme la plus fréquente : Pour trouver la somme la plus fréquente, on peut parcourir le tableau `occurrences` en recherchant la plus grande valeur. On peut utiliser une variable **maxOccurrences** pour stocker le nombre maximum d'occurrences et une variable **maxSum** pour stocker la somme correspondante. On peut utiliser une boucle **for** pour itérer sur le tableau et comparer chaque valeur avec **maxOccurrences**. Si une valeur est supérieure à **maxOccurrences**, on met à jour **maxOccurrences** et **maxSum**.

EXERCICE 5

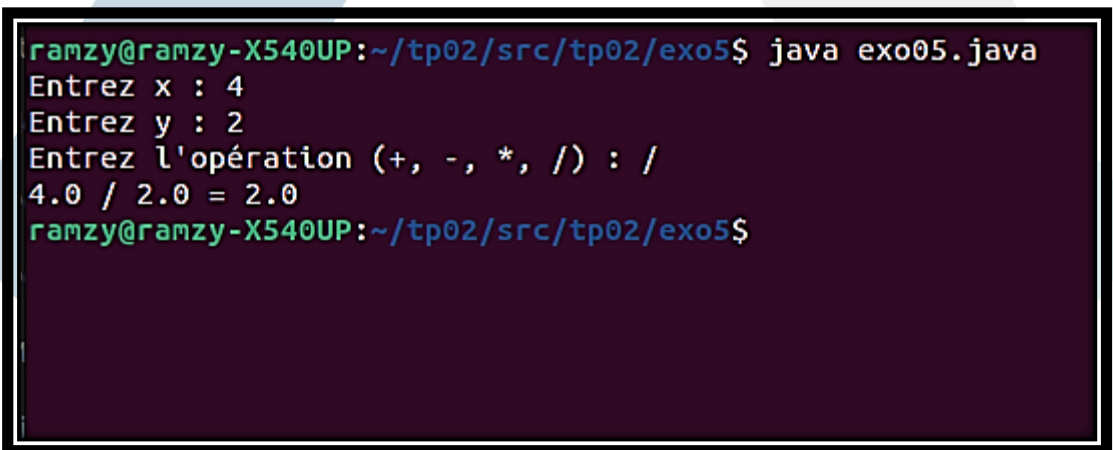
```
import java.util.Scanner;

public class exo05 {

    public static double operation(double x, double y, char op) {
        switch (op) {
            case '+':
                return x + y;
            case '-':
                return x - y;
            case '*':
                return x * y;
            case '/':
                return x / y;
            default:
                return x + y;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Entrez x : ");
        double x = sc.nextDouble();
        System.out.print("Entrez y : ");
        double y = sc.nextDouble();
        System.out.print("Entrez l'opération (+, -, *, /) : ");
        char op = sc.next().charAt(0);
        double result = operation(x, y, op);
        System.out.println(x + " " + op + " " + y + " = " + result);
        sc.close();
    }
}
```

Output :



```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo5$ java exo05.java
Entrez x : 4
Entrez y : 2
Entrez l'opération (+, -, *, /) : /
4.0 / 2.0 = 2.0
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo5$
```

La méthode **operation** prend en entrée deux valeurs réelles **x** et **y** ainsi qu'un caractère **op**. Elle renvoie le résultat correspondant à l'opération indiquée par **op**, qui peut être +, -, * ou /. Si **op** est un autre caractère, la méthode effectue une addition par défaut.

Le programme principal (méthode main) appelle la méthode **operation** quatre fois avec des valeurs connues de **x**, **y** et des opérateurs différents, et affiche les résultats.

Tips and tricks :

- La structure de contrôle **switch** est utilisée pour évaluer différentes valeurs d'une variable et exécuter le code correspondant au cas correspondant.
- Utilisation de la méthode `sc.next().charAt(0);` pour lire un caractère depuis l'entrée utilisateur :

L'objet **Scanner** permet de lire l'entrée utilisateur depuis la console. Pour lire un caractère, on utilise la `sc.next().charAt(0);` qui permet de lire le premier caractère de la prochaine chaîne de caractères saisie par l'utilisateur. Dans l'exemple donné, cette méthode est utilisée pour lire le caractère d'opération saisi par l'utilisateur (dans la variable '**op**'). Le caractère saisi est ensuite utilisé dans la structure de contrôle **switch** pour déterminer quelle opération arithmétique doit être effectuée.

Par exemple, si l'utilisateur entre le caractère "+", la méthode `sc.next().charAt(0);` retournera le caractère '+'.

EXERCICE 6

```
import java.util.Scanner;

public class exo06 {

    public static boolean premierTest(int n) {
        if (n <= 1) {
            return false;
        }

        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
                return false;
            }
        }

        return true;
    }

    public static void chercherNbPremier(int n) {
        for (int i = 2; i <= n; i++) {
            if (premierTest(i)) {
                System.out.print(i + " ");
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Entrer une valeur pour n: ");
        int n = sc.nextInt();
        if (premierTest(n)==true) System.out.println("Le nombre est premier");
        else System.out.println("Le nombre n'est pas premier");
        System.out.println("La liste des nombres premiers jusqu'à "+n+ " : ");

        long start = System.currentTimeMillis();
        chercherNbPremier(n);
        long end = System.currentTimeMillis();

        double timeElapsed = (end - start) / 1000.0;
        System.out.printf("Temps d'execution: %.5f secondes\n",
            timeElapsed);
    }
}
```

Le code fournit une implémentation de deux méthodes permettant de travailler avec des nombres premiers.

La première méthode, « **boolean premierTest (int n)** », prend en entrée un nombre entier **n** et renvoie un booléen **true** si **n** est un nombre premier et **false** sinon. Elle vérifie si **n** est divisible par tous les nombres entre 2 et **n/2**, si oui, elle renvoie **false**, sinon elle renvoie **true**.

La deuxième méthode, « **void chercherNbPremier(int n)** », prend en entrée un nombre entier **n** et affiche à l'écran tous les nombres premiers compris entre 2 et **n** en utilisant la fonction **premierTest**. Elle utilise une boucle for pour parcourir tous les entiers entre 2 et **n**, et appelle la fonction **premierTest** pour chaque entier. Si **premierTest** renvoie **true**, l'entier est affiché à l'écran.

Dans le programme principal, la méthode **chercherNbPremier** est appelée avec **n=100** pour afficher tous les nombres premiers entre 2 et 100. Le temps d'exécution de la méthode **chercherNbPremier** est également calculé en utilisant la fonction **System.currentTimeMillis()** pour mesurer le temps écoulé avant et après l'exécution de la méthode. Ce temps est affiché à l'écran en secondes avec une précision de 5 chiffres après la virgule.

Enfin, le code a été optimisé pour améliorer les performances. Pour cela, la méthode **premierTest** ne teste que les diviseurs impairs plutôt que tous les diviseurs entre 2 et $n/2$, car si un nombre **n** est divisible par un nombre pair, alors il est aussi divisible par 2 et **n** ne peut donc pas être un nombre premier. Cette optimisation permet de réduire le nombre de tests effectués et donc d'améliorer les performances du programme.

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo6$ java exo06.java
Entrer une valeur pour n: 100
Le nombre n'est pas premier
La liste des nombres premiers jusqu'à 100 :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Temps d'exécution: 0.00700 secondes
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo6$
```

Tips and tricks :

- La méthode **System.currentTimeMillis()** est utilisée pour mesurer le temps d'exécution d'une partie du code. Voici quelques astuces pour son utilisation :
 1. Appeler la méthode **System.currentTimeMillis()** avant et après le bloc de code à mesurer.
 2. Soustraire le temps de fin du temps de début pour obtenir le temps écoulé en millisecondes.
 3. Diviser le temps écoulé par 1000.0 pour obtenir le temps écoulé en secondes.
 4. Utilisez la méthode **printf** pour afficher le temps écoulé avec un nombre précis de décimales.

EXERCICE 7

```
import java.util.Scanner;

public class exo07 {

    // Question 1
    public static void affseq(int n) {
        if (n == 0) {
            System.out.print(n);
            return;
        }
        affseq(n - 1);
        System.out.print(", " + n);
    }

    // Question 2
    public static void affseqInv(int n) {
        if (n == 0) {
            System.out.print(n);
            return;
        }
        System.out.print(n + ",");
        affseqInv(n - 1);
    }

    // Question 3
    public static int somme_un_a(int n) {
        if (n == 1) {
            return 1;
        }
        return n + somme_un_a(n - 1);
    }

    // Question 4
    public static int facto(int n) {
        if (n == 0 || n == 1) {
            return 1;
        }
        return n * facto(n - 1);
    }

    // Fonction main pour tester les différentes fonctions
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Entrez un entier n : ");
        int n = sc.nextInt();

        System.out.println("Question 1 :");
        affseq(n);

        System.out.println("\nQuestion 2 :");
        affseqInv(n);

        System.out.println("\nQuestion 3 :");
        int sum = somme_un_a(n);
        System.out.println("Somme de 1 à "+ n + " : " + sum);
    }
}
```

```
System.out.println("\nQuestion 4 :");
System.out.print("Entrez un entier n : ");
n = sc.nextInt();
int fact = facto(n);
System.out.println("Factorielle de "+ n + " : " + fact);
}
}
```

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo7$ java exo07.java
Entrez un entier n : 10
Question 1 :
0,1,2,3,4,5,6,7,8,9,10
Question 2 :
10,9,8,7,6,5,4,3,2,1,0
Question 3 :
Somme de 1 à 10 : 55
Question 4 :
Entrez un entier n : 5
Factorielle de 5 : 120
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo7$
```

Tips and tricks :

- La fonction **affseq()** affiche une séquence de nombres de 0 à n en utilisant la récursivité descendante. L'idée principale est que la fonction appelle elle-même avec un paramètre réduit à chaque appel jusqu'à atteindre la condition d'arrêt, qui est lorsque n est égal à 0. À ce stade, la fonction affiche simplement 0 et retourne.
- La fonction **affseqInv()** affiche une séquence de nombres de n à 0 en utilisant la récursivité. L'astuce ici consiste à placer l'affichage du nombre avant l'appel récursif. De cette façon, les nombres sont affichés dans l'ordre décroissant.
- La fonction **somme_un_a()** calcule la somme des nombres de 1 à n en utilisant la récursivité ascendante. L'idée principale est que la fonction appelle elle-même avec un paramètre qui est augmenté à chaque appel jusqu'à atteindre la condition d'arrêt, qui est lorsque n est égal à 1. À ce stade, la fonction retourne simplement 1.
- La fonction **facto()** calcule la factorielle d'un nombre en utilisant la récursivité descendante. L'astuce ici consiste à appeler la fonction elle-même avec un paramètre réduit à chaque appel jusqu'à atteindre la condition d'arrêt, qui est lorsque n est égal à 0 ou 1. À ce stade, la fonction retourne simplement 1.

EXERCICE 8

```
import java.util.Scanner;

public class exo08 {

    /**
     * Méthode de construction du triangle supposant la structure
     mémoire existante
     * @param triangle Tableau des coefficients
     */
    public static void construitTriangle(int[][] triangle) {
        for (int i = 0; i < triangle.length; i++) {
            triangle[i][0] = 1;
            triangle[i][i] = 1;
            for (int j = 1; j < i; j++) {
                triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j];
            }
        }
    }

    /**
     * Méthode d'initialisation de la hauteur du triangle uniquement (nb
     de lignes)
     * @param nbNiveaux Indique le nombre de lignes utilisées
     * @return La matrice de coefficients initialisée en hauteur
     uniquement
     */
    public static int[][] creer_HauteurTriangle(int nbNiveaux) {
        if (nbNiveaux <= 0) {
            System.out.println("Nombre de niveaux non valide,
initialisation à 6 niveaux.");
            nbNiveaux = 6;
        }
        int[][] triangle = new int[nbNiveaux][];
        for (int i = 0; i < nbNiveaux; i++) {
            triangle[i] = new int[i+1];
        }
        return triangle;
    }

    /**
     * Méthode pour afficher le triangle de Pascal
     * @param triangle Tableau des coefficients
     */
    public static void afficheTriangle(int[][] triangle) {
        for (int i = 0; i < triangle.length; i++) {
            for (int j = 0; j < triangle[i].length; j++) {
                System.out.print(triangle[i][j] + " ");
            }
            System.out.println();
        }
    }

    /**
     * Fonction principale
     * @param args Les arguments de la ligne de commande (non utilisés
     ici)
     */
}
```

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Entrez le nombre de niveaux : ");
    int nbNiveaux = input.nextInt();
    int[][] triangle = creer_HauteurTriangle(nbNiveaux);
    construitTriangle(triangle);
    afficheTriangle(triangle);
}
```

Output :

```
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo8$ java exo08.java
Entrez le nombre de niveaux : 8
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
ramzy@ramzy-X540UP:~/tp02/src/tp02/exo8$
```

La méthode **construitTriangle** remplit le tableau 2D triangle avec les coefficients de Pascal en utilisant les règles décrites dans l'énoncé. La méthode **creer_HauteurTriangle** initialise la structure de données 2D pour stocker les coefficients, en créant d'abord un tableau 2D de taille **nbNiveaux** et en initialisant les tableaux relatifs à chaque niveau avec la taille appropriée. La méthode **afficheTriangle** affiche simplement le contenu de triangle à l'écran.

La méthode main demande à l'utilisateur d'entrer le nombre de niveaux pour le triangle de Pascal, initialise la structure de données avec la méthode **creer_HauteurTriangle**, remplit le tableau avec les coefficients de Pascal en appelant **construitTriangle**, puis affiche le résultat avec **afficheTriangle**.

Tips and tricks :

- Pour déclarer une matrice en Java, vous devez utiliser la syntaxe suivante :

```
type[][] nomMatrice = new type[nbLignes][nbColonnes];
```

Par exemple, pour déclarer une matrice d'entiers de 3 lignes et 4 colonnes :

```
int[][] matrice = new int[3][4];
```


- Vous pouvez également créer une matrice avec un nombre variable de colonnes pour chaque ligne. Pour cela, vous devez déclarer la matrice en utilisant un tableau à une dimension, puis initialiser chaque élément avec un tableau à une dimension de taille variable.

```
type[][] nomMatrice = new type[nbLignes][];  
for (int i = 0; i < nbLignes; i++) {  
    nomMatrice[i] = new type[nbColonnes];  
}
```

- Pour accéder aux éléments d'une matrice, vous pouvez utiliser les indices de ligne et de colonne. L'indice de la première ligne et de la première colonne est 0. Pour accéder à un élément spécifique, utilisez la syntaxe suivante :

```
nomMatrice[ligne][colonne]
```

Pour passer une matrice en tant qu'argument à une méthode, vous devez spécifier le type de la matrice suivi de deux paires de crochets. Par exemple, pour une méthode qui prend en entrée une matrice d'entiers de taille variable :

```
public static void maMethode(int[][] matrice) {  
    // Code  
}
```

For each :

- Pour afficher une matrice à l'aide d'une boucle for each, vous pouvez utiliser la syntaxe suivante :

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
  
for (int[] row : matrix) {  
    for (int value : row) {  
        System.out.print(value + " ");  
    }  
    System.out.println();  
}
```

LIEN DES CODES

Vous trouverez tous les codes (.java) sur notre **Github**

github.com/Resumily/poo-tp2



Cliquez sur l'icône

Bonne révision !
-L'équipe de Résumily-



Liens / Résumily

Cliquez sur les icônes