# MOOC Init Prog Java
# Exercices semaine 6

---

### Exercice 18 (fonctions, passage des paramètres)

Soit le programme suivant :

```
class ConcatIncorrecte
{
    public static void main(String[] args)
        {
            String s = "China Blue";
            System.out.println(s);
            concatener(s, " Express");
            System.out.println(s);

        }


    public static void concatener(String s, String s2)
        {
            s +=s2;
        }
}
```

1. Expliquez pourquoi la méthode `concatener` ne parvient pas à modifier la chaîne `s` du `main` (en y concaténant `" Express"`)
2. Corriger le codage de la méthode `concatener` et son utilisation dans le `main` de sorte à ce que l'exécution du programme affiche :

   ```
   China Blue
   China Blue Express
   ```

   au lieu de :

   ```
   China Blue
   China Blue
   ```

---

## Exercice 19 : Multiplication matricielle revisitée (Fonctions)

Le programme `MulMat.java` que vous avez développé dans l'exercice 13 contient de nombreuses portions de code dupliquées: il est mal modularisé!
On souhaiterait améliorer ce programme au moyen de méthodes auxiliaires. Dans le programme `MulMatMod.java`, la méthode `main` a déjà été réécrite avec des appels à des méthodes auxiliaires utiles. Il vous est demandé de compléter le code de `MulMatMod.java`. Vous pouvez transférer les instructions nécessaires depuis le programme `MulMat.java`en les copiant-collant. Exécutez les deux programmes pour vérifier qu'ils ont le même comportement.
Code à compléter :

```
class MulMatMod {

        /* Notez que le Scanner à utiliser pour les lectures est déclaré ici
         * globalement pour ne pas être recréé à chaque appel de méthode
         * où il est utile.
         * Toute méthode du programme peut alors utiliser la variable scanner
         * sans la re-déclarer.
         * Il n'y a plus besoin d'avoir recours à scanner.close();
         */

        private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        double[][] mat1 = lireMatrice();
        double[][] mat2 = lireMatrice();
        if (mat1[0].length != mat2.length) {
            System.out.println("Multiplication de matrices impossible !");
        } else {
            double [][] prod = multiplierMatrice(mat1, mat2);
            System.out.println("Résultat :");
            afficherMatrice(prod);
        }
    }
}
```

---

## Exercice 20 : nombres amicaux (fonctions)

En mathématiques, deux nombres entiers sont dits amicaux si :

1. la somme des diviseurs de l'un, `m`, coïncide avec la somme des diviseurs de l'autre;
2. et la somme des deux nombres vaut `m`.

Par exemple 220 et 284 sont amicaux car :

*somme des diviseurs de 220 = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 + 220 = 504*
*somme des diviseurs de 284 = 1 + 2 + 4 + 71 + 142 + 284 = 504*
*220 + 284 = 504.*

Le but de cet exercice est d'écrire une méthode `afficherAmicaux` qui prend en entrée un tableau d'entiers et retourne toutes les paires de nombres amicaux qu'il contient.

Chaque paire ne sera affichée qu'une fois.

Proposez une implémentation possible (en Java) pour la méthode `afficherAmicaux`.

Vous pouvez utiliser l'exemple auivant en guise de programme principal :

```
public static void main(String[] args) {
int[] nombres = {1210, 45, 27, 220, 54, 284, 9890, 120, 1184};
System.out.println("Les paires de nombres amicaux sont : ");
afficherAmicaux(nombres);
}
```

L'affichage qui en résulte devrait ressembler à ceci :

```
Les paires de nombres amicaux sont :
1210 1184
220 284
```

Modularisez votre code au moyen de méthodes auxiliaires.

---

## Exercice 21 : Césure (chaînes de caractères, fonctions)

La césure est la façon de couper les mots afin de pouvoir les imprimer sur deux ou plusieurs lignes. Un tiret à la fin de la ligne indique si le mot continue sur la ligne suivante. Exemple:

```
Aussitôt que le message lui parve-
nait, le roi des rois sortait.
```

Dans cet exercice, il est question de compléter un petit programme de césure appellé `Cesure.java`. Il y a 4 méthodes auxiliaires à compléter en utilisant notamment des méthodes prédéfinies de la classe `String`. Le programme à compléter ci-dessous devra lire une phrase sous la forme d'un tableau de chaines de caractères et indiquer les endroits où le(s) mot(s) peu(ven)t être coupé(s).

```
class Cesure {
    public static void main(String[] args) {
        String[] phrase = lirePhrase();
        for (int i = 0; i < phrase.length; i++) {
            cesure(phrase[i]);
        }
    }

    static String[] lirePhrase() {
        // A compléter:
        // retourne un tableau de chaines de caractères
        // introduits par l'utilisateur
    }

    static boolean voyelle(char c) {
        // A compléter:
        // teste si un caractère est une voyelle
    }

    static boolean queVoyelles(String s) {
        // A compléter:
        // teste si une chaîne ne contient que des voyelles
        // utilise la méthode voyelle
    }

    static void cesure(String mot) {
        // A compléter:
        // détermine la césure d'un mot donné et effectue les affichages
        // correspondants (voir exemple de déroulement)
    }
}
```

Voici les trois règles de césure que vous devrez appliquer. Celles-ci ne correspondent évidemment pas aux véritables règles utilisées en français car le programme deviendrait trop compliqué pour le but de cet exercice.

- Un mot ne peut être coupé qu'entre une voyelle et une consonne,
- Une lettre ne peut être seule sur une ligne. Il faut veiller à cette situation au début et à la fin d'un mot
- Il doit y avoir au moins une consonne sur chaque ligne

Votre programme respectera également les règles suivantes:

- On utilisera l'alphabet latin avec les 26 lettres de `a` à `z`
- les voyelles sont `a, e, i, o, u, y`
- on supposera que le mot est toujours sans accents ni ponctuation et en lettres minuscules.

Les exemples d'exécutions ci-dessous illustrent différentes situations possibles:

```
>java Cesure
Donnez le nombre de mots dans votre phrase: 1
Donnez le mot 1 : java

Le résultat est :

ja-
va

>java Cesure
Donnez le nombre de mots dans votre phrase: 1
Donnez le mot 1 : calculer

Le résultat est :

ca-
lcu-
ler

>java Cesure
Donnez le nombre de mots dans votre phrase: 2
Donnez le mot 1 : tapis
Donnez le mot 2 : rouge

Le résultat est :

ta-
pis
rou-
ge

>java Cesure
Donnez le nombre de mots dans votre phrase: 3
Donnez le mot 1 : oeil
Donnez le mot 2 : de
Donnez le mot 3 : boeuf

Le résultat est :
```

```
oeil
de
boeuf


>java Cesure
Donnez le nombre de mots dans votre phrase: 0
entrez une valeur plus grande que 0
```

Les méthodes suivantes de la classe `String` peuvent vous être utile (vous n'aurez pas forcément besoin de toutes ces méthodes):

- `length()`
  Retourne la longueur de la chaine de caractères, c'est-à-dire le nombre de caractères qui la composent.
- `char charAt(int index)`
  Retourne le caractère qui se trouve à la position `index` de la chaine de caractères. Le premier caractère se trouve à la position 0 et le dernier à la position `length()-1`.
- `String substring(int beginIndex, int endIndex)`
  Retourne une nouvelle chaine de caractères composée de la partie de la chaine de caractères actuelle qui commence à la position `beginIndex` et se termine à la position `(endIndex-1)`.
- `String concat(String str)`
  Retourne une nouvelle chaine de caractères i constituée de la chaine courante à laquelle a été concaténée (collée) la chaine `str`.
- `String.valueOf(c)` permet de convertir le `char c` en `String`.

---

## Exercice 22 : Nombres de Fibonacci (fonctions récursives)

Les nombres de Fibonacci sont définis par la suite :

```
F(0) = 0
F(1) = 1
F(n) = F(n-1)+ F(n-2) avec n>1
```

Le but de cet exercice est d'écrire un programme qui calcule la valeur de `F(n)` selon la définition récursive précédente.

Dans le fichier `Fibonacci.java`, définissez la fonction

```
int Fibonacci(int  n)
```

qui calcule la valeur de `F(n)` de manière récursive (cette fonction devra donc faire appel à elle-même) sans utiliser de structure de boucle (for, while, etc…) et sans aucune variable locale.

Pour comparaison, voici une manière itérative (i.e. non récursive) de calculer les `n` premier termes de la suite :

```
static int FibonacciIteratif(int n)
{
  int Fn = 0;    // stocke F(i)  , initialisé par F(0)
  int Fn_1 = Fn; // stocke F(i-1), initialisé par F(0)
  int Fn_2 = 1;  // stocke F(i-2), initialisé par F(-1)

  if (n > 0)
    for (int i = 1; i <= n; ++i) {
      Fn   = Fn_1 + Fn_2;    // pour n>=1 on calcule F(n)=F(n-1)+F(n-
      Fn_2 = Fn_1;           // et on décale...
      Fn_1 = Fn;
    }
  return Fn;
}
```

Note : la méthode récursive est coûteuse en temps de calcul, ne la lancez pas pour des nombres trop élevés (disons supérieurs à 40).

### Exemple de déroulement

```
Entrez un nombre entier compris entre 0 et 40 : 0
Méthode itérative :
    F(0) = 0
Méthode récursive :
    F(0) = 0
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 1
Méthode itérative :
```

```
    F(1) = 1
Méthode récursive :
    F(1) = 1
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 2
Méthode itérative :
    F(2) = 1
Méthode récursive :
    F(2) = 1
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 3
Méthode itérative :
    F(3) = 2
Méthode récursive :
    F(3) = 2
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 7
Méthode itérative :
    F(7) = 13
Méthode récursive :
    F(7) = 13
Voulez-vous recommencer [o/n] ? n
```

---