Fonctions (méthodes)

Les **fonctions** (**Méthodes** en Java) en programmation sont des traitements, qui agissent sur des données.

Les **méthodes** sont des morceaux de code **réutilisables**, que l'on définit à un endroit du programme et que l'on peut appeler depuis un ou plusieurs autre(s) endroit(s)

Fonctions (méthodes)

Les méthodes permettent de:

- éviter la **duplication** de code
- facilement **maintenir** du code (si on veut changer le code, il suffit de changer le code de la fonction)
- avoir un programme bien **structuré** et facile à **comprendre**
- faciliter la **réutilisation** du code

Fonctions (méthodes)

Une fonction est caractérisée par :

- **un corps** : la portion de code à exécuter
- **un nom** : celui par lequel on désignera cette fonction
- **des paramètres** (des **arguments**) : ensemble de variables (extérieures à la fonction) que la fonction prend en entrée, et dont le corps a besoin pour fonctionner
- **un type et une valeur de retour** : la sortie de la fonction, ce qu'elle renvoie au reste du programme
- **l'appel** est l'endroit où l'on utilise la méthode. Il contient le nom de la méthode et les **arguments** que l'on veut lui passer

Fonctions (méthodes)

Syntaxe:

```
return_type name_method (type p1, type p2, ...)
{
  // Body of the method goes here
}
```

```
public static void main(String[] args) {
  int x = 10, y = 15;
  System.out.print(somme(x,y));//appel de la fonction somme
  }

//définition de la fonction somme
static int somme(int a, int b) {
  return a+b;
}
```

Parfois, une méthode ne renvoie pas de valeur. Le mot clé *void* est utilisé comme type de retour si une méthode ne renvoie aucune valeur.

```
1  //soit les tableaux suivants, tab1, tab2, tab3 et m
2  //1. Rechercher et afficher la valeur max dans les trois premiers
       tableaux
3
4          int tab1[]={2,5,16,9,3};
5          int tab2[]={1,5,6,9,13};
6          int tab3[]={12,5,6,9,3};
7          int m[][]={{12,5,6,9,3}, {1,5,6,9,13}, {2,5,16,9,3}};
8
9  //2. Créer une fonction MaxTable qui renvoie comme valeur la valeur
        max dans un tableau
10 //3. Utiliser cette fonction pour afficher la valeur max des
       tableaux précédents
11 //4. Créer une fonction MaxMatrice qui renvoie comme valeur la
       valeur max dans une matrice en faisant appelle á la fonction
       MaxTab
12 //5. Utiliser cette fonction pour afficher la valeur max de la
       matrice m
```

Fonctions (méthodes)

Signature d'une méthode:

Une méthode a une **signature**, qui identifie de **manière unique** la méthode dans un contexte particulier.

La signature d'une méthode est la combinaison des quatre parties suivantes :

☆ **Nom** de la méthode

☆ **Nombre** de paramètres

☆ **Ordre** de paramètres

☆ **Types** des paramètres

Fonctions (méthodes)

Signature d'une méthode:

```
1   int somme(int a){
2   return a+a;
3   }
4
5   int somme(int a, int b){
6   return a+b;
7   }
8
9   int somme(int a, int b, int c){
10  return a+b+c;
11  }
12
13  double somme(double a, double b){
14  return a+b;
15  }
```

Fonctions (méthodes)

Évaluation d'un appel de méthode :

L'appel de méthode se passe par cinq étapes :

1) Les expressions passées en **argument** sont évaluées (en peut passer, lors de l'appel d'une fonction, comme arguments soit des **valeurs concrètes**, soit des **variables** soit des **expressions**)
2) Les valeurs correspondantes sont **affectées** aux **paramètres de la méthode** (les valeurs sont **copiées** dans les paramètres de la méthode)
3) Le corps de la méthode **s'exécute** avec ces valeurs
4) L'expression suivant la première commande ***return*** rencontrée est évaluée
5) La valeur obtenue est **retournée** comme résultat de l'appel : cette valeur remplace l'expression de l'appel

Fonctions (méthodes)

Évaluation d'un appel de méthode :

Si une méthode est sans arguments, les étapes 1 et 2 n'ont pas lieu. Ainsi, si la méthode est sans valeur de retour (type de retour *void*), les étapes 4 et 5 n'ont pas lieu.

Fonctions (méthodes)

Les références et passage de paramètres :

En programmation, **de façon générale**, on dira que :

☆ Un **argument** est passé par **valeur** si la méthode ne peut pas le **modifier**, la méthode **créé une copie locale** de cet argument (travaille avec une **copie**)

☆ Un **argument** est passé par **référence** (adresse) si la méthode peut le **modifier** (on connaît son adresse)

En **Java**, **il n'existe que le passage par valeur**, mais cela a des conséquences différentes selon que le type du paramètre est simple (int, double, etc.) ou évolué (Objet)

Fonctions (méthodes)

Les références et passage de paramètres :

Par exemple:

- pour les entiers le passage est par valeur.
- Par contre, pour les tableaux, le passage des paramètres s'effectue par adresse (par référence). On peut **accéder** et **modifier** les valeurs du tableau passé en paramètre

Fonctions (méthodes)

Les références et passage de paramètres :

```
1  // Qu'affiche ce code:
2
3  public static void main(String[] args) {
4    int x = 10, y = 15;
5    System.out.print(x + " " + y + " ");
6
7    f1(x,y);
8    System.out.println(x + " " + y + " ");
9
10   String s1 = "abcd", s2 = "efgh";
11   System.out.print(s1 + s2);
12
13   f2(s1,s2);
14   System.out.print(s1 + s2);
15   }
16
17 static void f1(int a, int b) {
18   int tmp = a; a = b; b = tmp;
19 }
20
21 static void f2(String s, String t) {
22   String tmp = s; s = t; t = tmp;
23 }
```

Fonctions (méthodes)

Les références et passage de paramètres :

```
1  // Qu'affiche le code suivant:
2  public static void main(String[] args)
3  {
4    int[] tab = {1, 2, 3, 4};
5    System.out.println(tab[0] + " " + tab[1]);
6    f(tab, 0, 1);
7    System.out.println(tab[0] + " " + tab[1]);
8  }
9
10 static void f(int[] t, int i, int j)
11 {
12   int tmp = t[i];
13   t[i] = t[j];
14   t[j] = tmp;
15   t = new int[4];
16 }
```

Fonctions (méthodes)

Les références et passage de paramètres :

R Les modifications faites dans la méthode sur la référence elle-même ne sont pas visibles à l'extérieur de la méthode.

```
1  public static void main(String[] args)
2  {
3      StringBuilder st=new StringBuilder ("Bonjour");
4
5      add(st);
6      replace(st);
7      System.out.println(st);
8
9  }
10
11
12 static void add(StringBuilder s){
13     s.append(" tout le monde");
14     }
15
16 static void replace(StringBuilder s){
17     StringBuilder ch=new StringBuilder("Hello every one");
18     s=ch;
19     }
```

Fonctions (méthodes)

```
1 // Crée un programme qui affiche les nombres premier de 1 á n en cr
      éant les méthodes suivantes:
2 //div(int i, int j): renvoie true si i est divisible sur j
3 //prime(int n): renvoie true si n est premier, false sinon
4 // liste_prime(int n) renvoie une liste des nombres premier de 1 á
      n
5 // afficherliste(n) affiche les éléments d'une liste
6 //modifier le programme pour qu'il affiche  les n premiers nombres
      premiers (ajouter une méthode)
```

Exercice 13 En mathématiques, deux nombres entiers n et m sont dits amicaux ou aimables ou amiables si la somme des diviseurs de n (n exclus) vaut m et la somme des diviseurs de m (m exclus) vaut n. Autrement dit, si la somme des diviseurs de l'un q, coïncide avec la somme des diviseurs de l'autre ; et la somme des deux nombres vaut q.

Par exemple 220 et 284 sont amicaux car, la somme :

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

D'une autre manières :

- ☆ La somme des diviseurs **propres** de **220** est 284 : divp(220) = 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 et la somme de ces nombres est **284**.
- ☆ La somme des diviseurs **propres** de **284** est 220 : divp(284) = 1, 2, 4, 71, 142 et la somme de ces nombres est **220**.

Voici quelques paires de nombres amicaux : (1184, 1210), (2620, 2924), (5020, 5564) et (6232, 6368).

Travail demandé : Modularisez le code au moyen de méthodes auxiliaires pour écrire une méthode ***Amicaux*** qui prend en entrée un tableau d'entiers et affiche toutes les paires de nombres amicaux qu'il contient. Chaque paire ne sera affichée qu'une seule fois. ■