Cours de Java : Les fonctions (méthodes) statiques
Chapitre 3

1. Pourquoi des fonctions? Pourquoi static ?

Les langages de programmation de haut niveau (C, C++, Java, php, ...) permettent de **découper** un programme en plusieurs parties appelées **fonctions** : on parle alors de programmation modulaire. L'utilisation de fonctions se justifie pour de multiples raisons :

- Un programme écrit d'un seul tenant devient difficile à comprendre (et à débugger) dès qu'il dépasse une ou deux pages de texte. L'utilisation de fonctions permet de le diviser en plusieurs parties et de regrouper dans le programme principal les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée à son tour en fonctions plus élémentaires ; ce processus de décomposition pouvant être répété autant de fois que nécessaire.
- Les fonctions permettent de "**factoriser**" le code : plutôt que d'écrire à plusieurs endroits différents la même séquence de code, on va l'écrire une seule fois dans une fonction. Ainsi, il n'y a qu'une seule portion de code à modifier lorsque c'est nécessaire. Cela permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus que les fonctions sont **paramétrables**.
- La programmation modulaire permet le **partage de taches et d'outils communs** entre plusieurs développeurs: le chef de projet affecte le développement de telles ou telles fonctions à certains développeurs.

Ce cours présente les fonctions (méthodes) **statiques**. Une fonction membre d'une classe (déclarée à l'intérieur de la classe) est appelée une **méthode**. Elle est déclarée **static** à l'aide du mot réservé **static**. Voir le paragraphe 2.1.

Quelques règles

- La fonction **main()** est **obligatoirement** une méthode **static**.
- Toute fonction (méthode) appelée directement dans la fonction **main()** doit être **static**.
- Une fonction (méthode) **static** peut être utilisée sans avoir à créer un objet sur la classe qui la contient.

💣 Ce chapitre 3 ne présente que des méthodes statiques car pour éviter de déclarer **static** des méthodes il faut savoir créer des objets dynamiquement avec l'opérateur **new** comme il sera vu dans le chapitre 4. La bonne façon de programmer est de créer des classes avec des méthodes non **static**.

Déclarer **static** des méthodes est plutôt réservé à des classes utilitaires (comme java.lang.Math qui sert comme une librairie de fonctions mathématiques) et sur lesquelles créer des objets ne sert à rien.

Compilation réussie	Erreur de compilation

public class A {
public **static** void f() {….}
public static void main(String []args) {
…
f() ; ***//appel de f() correct, f() est static***
…
}
}
	public class A {
	public void f() {….}
	public static void main(String []args) {
	…
	f() ; ***//appel de f() incorrect***
	…
	}
	}

La méthode f() appelée par la fonction statique main() doit obligatoirement être statique.

Le but d'une fonction est de réaliser une tâche: cette tâche est constituée des instructions que la fonction doit exécuter. Le code d'une fonction est appelé la **définition** de la fonction. .

Il y a 2 situations possibles:
- la fonction ne fait qu'exécuter la tâche et rien de plus: la fonction est alors de type **void**, elle ne retourne rien,
- la fonction exécute une tâche, cette tâche contient un calcul et la fonction doit renvoyer (retourner) le résultat de ce calcul, la fonction est alors du type du résultat du calcul: type **int** ou type **float** ou type **double** ou…

2. Exemples de base

Exemple complet avec le main

```
public class ExemplesFonctions {

	// fonctions static car appelées directement à partir du main
	public static void f() {
		System.out.println("Je suis la fonction f");
	}
	public static void afficheMessage(String message) {
		System.out.println("Message reçu:"+message);
	}
	public static void additionne(int x, int y) {
		int s = x + y ;
		System.out.println("Somme calculée = "+s); ;
	}
	public static int additionneBis(int x, int y) {
		int s = x+y ;
		return s ;
	}
	public static String doubleMot(String s ) {
		String r = s+s ;
		return r ;
	}
	public static void main(String[] args) {
		System.out.println("Avant l'appel de f");
		f() ;
		System.out.println("Après l'appel de f");
		afficheMessage("Bonjour, le contrôle est reporté");
		afficheMessage("Houlala, c'est pas terrible");
		additionne(25,15) ;
		additionne(200,300) ;
		int somme = additionneBis(200,400);
		System.out.println("Valeur retournée = "+somme) ;
		String ch = doubleMot("Bonjour");
		System.out.println(ch);
	}
}
```

Affichage obtenu

```
Avant l'appel de f
Je suis la fonction f
Après l'appel de f
Message reçu:Bonjour, le contrôle est reporté
Message reçu:Houlala, c'est pas terrible
Somme calculée = 40
Somme calculée = 500
Valeur retournée = 600
BonjourBonjour
```

Chaque fonction est appelée dans la fonction main.
L'appel d'une fonction entraîne son exécution.

1er exemple: définition d'une fonction qui ne retourne rien

```
public static void f() {
        System.out.println("Je suis la fonction f");
    }
```

La fonction *f()* n'a pas d'argument ⇨ appel avec f().

2ème exemple: définition d'une fonction qui reçoit un argument et qui ne retourne rien

```
public static void afficheMessage(String message) {
        System.out.println("Message reçu:"+message);
    }
```

La fonction *afficheMessage(String message)* doit recevoir un argument de type *String*.
Lors de l'appel de la fonction par *afficheMessage("Bonjour, le contrôle est reporté"))*, l'argument *message* est initialisé avec le String *"Bonjour, le contrôle est reporté"*.

3ème exemple: définition d'une fonction qui reçoit 2 arguments et qui ne retourne rien

```
public static void additionne(int x, int y) {
        int s = x + y ;
        System.out.println("Somme calculée = "+s); ;
    }
```

La fonction *additionne(int x, int y)* doit recevoir 2 arguments de type entier. Elle ne retourne rien (elle est de type void).
Son exécution est lancée par l'instruction ***additionne*(25,15) ;** , *x* prend alors la valeur *25* et *y* la valeur *15*.
Elle est ensuite de nouveau appelée par l'instruction ***additionne*(200,300) ;** , *x* prend alors la valeur *200* et *y* la valeur *300*.

4ème exemple: définition d'une fonction qui reçoit 2 arguments et qui retourne la somme de ces 2 arguments

```
    public static int additionneBis(int x, int y) {
        int  s = x+y ;
        return s ;
    }
```

La fonction *additionneBis(int x, int y)* doit recevoir 2 arguments de type entier. Elle est de type *int*, elle retourne (renvoie) un entier.
Elle est appelée par l'instruction ***int somme = additionneBis*(200,400) ;** , *x* prend alors la valeur *200* et *y* la valeur *400*. Elle calcule la somme des 2 nombres, range ce résultat dans *s* et retourne (renvoie) ce résultat dans la variable *somme*.

5ème exemple: définition d'une fonction qui reçoit 1 argument de type String et qui retourne un String égal à l'argument doublé ("jean" -> "jeanjean")

```
public static String doubleMot(String s ) {
        String  r = s+s ;
        return r ;
}
```

La fonction *doubleMot(String y)* doit recevoir 1 argument de type *String*. Elle est de type *String*, elle retourne (renvoie) un *String*.
Elle est appelée par l'instruction *String ch = doubleMot("Bonjour");* , *s* prend alors la valeur "Bonjour". Elle double le mot *s* ("Bonjour Bonjour") , range le résultat dans *r* puis retourne *r*.

public static void additionne(**int** x, **int** y)

type de la fonction (méthode)

public static int additionneBis(**int** x, **int** y)

La fonction contient l'instruction **return valeurDeTypeEntier** dans son code.

3. Exemple avec calcul de l'hypothénuse

```
import java.util.Scanner;        // classe utilisée pour réaliser les saisies

public class Hypotenuse {
   public static  void main(String[] args)
   {
        // déclaration des variables locales de la fonction main()
        double am, bm, hypom;          // m comme main()

        Scanner sc;      // déclaration d'une variable référence destinée à pointer sur un objet de type
                         // Scanner
        sc = new Scanner(System.in);  // Création de l'objet de type classe Scanner, l'opérateur new
                                      // renvoie une référence sur cet objet

        // appel de la fonction print() pour l'objet référencé par out
        System.out.print("--> Veuillez saisir la longueur du premier coté : ");
        am = sc.nextDouble();  // appel de la fonction nextDouble() pour l'objet référencé par sc

        System.out.print("--> Veuillez saisir la longueur du second coté : ");
        bm = sc.nextDouble();

        // 1er appel de la fonction calculHypotenuse()
        hypom = calculHypotenuse(am, bm);
        System.out.println("1. Longueur de l'hypoténuse du triangle rectangle");
        System.out.println(" de cotés " + am + " et " +  bm + " : " + hypom + "\n");

        // 2ème appel de la fonction calculHypotenuse()
        hypom = calculHypotenuse(am + 2.5 , 7.8);
        System.out.println("2. Longueur de l'hypoténuse du triangle rectangle");
        System.out.println(" de cotés " + (am+2.5) + " et " +  7.8 + " : " + hypom + "\n");
   }
```

```
   /* DEFINITION de la fonction membre calculHypotenuse()
   Fonction qui calcule la longueur de l'hypoténuse d'un triangle rectangle :
      - 1er paramètre : longueur du 1er coté
      - 2ème paramètre : longueur du second coté
      - valeur de retour : l'hypoténuse*/

    public static double calculHypotenuse(double a, double b)  // entête de la fonction  membre
   {      // début du corps de la fonction membre calculHypotenuse()
          // déclaration des variables locales de la fonction calculHypotenuse()
          double res;
          // appel de la fonction statique  sqrt(). Cette fonction fait partie de la classe Math
          res = Math.sqrt(a*a + b*b);
          return res;      // => retour dans la fonction appelante (ici, il s'agit de la fonction main())
                           // en renvoyant la valeur de res
   }      // fin du corps de la fonction membre calculHypotenuse()
} // fin de la définition de la classe Hypotenuse
```

3.1 La définition de la fonction membre calculHypotenuse()

Le programme précédent définit, en plus de la fonction membre main(), une nouvelle fonction : la fonction membre calculHypotenuse(). Comme la fonction main(), cette fonction **appartient à la classe Hypotenuse** : sa définition est placée à l'intérieur de la définition de la classe. C'est une fonction membre (ou une méthode) de la classe Hypotenuse.
Rôle d'une définition de fonction : définir ou décrire le travail effectué quand la fonction est appelée.
La 1ère ligne de la définition correspond à l'**entête** de la fonction. Elle est suivie par le **corps** de la fonction délimité entre accolades ({ et }).

Remarque : les définitions des différentes fonctions membres d'une même classe doivent être placées les unes à la suite des autres à l'intérieur de la définition de cette classe. Elles ne doivent ni être imbriquées les unes dans les autres, ni être placées à l'extérieur de la définition de la classe.
Exemple correct :

```
public class Exemple
{
        public static ... fct1 (.....)
        {
                .....
        }

        public static ... fct2 (.....)
        {
                .....
        }
}
```

3.1.1 L'entête (prototype) de la fonction membre **calculHypotenuse()**

```
public static double calculHypotenuse(double a, double b)
```

Type de la valeur de retour — Nom de la fonction — Type et nom des paramètres

D'abord, cette entête indique que la méthode nommée **calculHypotenuse()** est :

- **public** : signifie qu'elle peut être appelée (utilisée) à l'extérieur de la classe **Hypotenuse** (cf. cours suivant);
- **static** : signifie que pour appeler cette fonction, il n'est pas nécessaire d'instancier (c'est à dire créer un objet) de la classe **Hypotenuse** (cf. cours suivant);

La suite de l'entête indique que pour appeler cette méthode, il faut lui **fournir "en entrée**" 2 valeurs de type double (correspondant aux **paramètres** de la fonction), et qu'en **sortie** la fonction membre renvoie ou retourne une valeur de type double (**type de la valeur de retour** : double). La simple lecture de l'entête de la fonction calculHypotenuse() permet de savoir comment on peut appeler cette fonction.

Dans l'**entête** de la fonction membre, il faut indiquer **le type et le nom** de chaque paramètre. Le 1er paramètre est de type double et est nommé a. Le second paramètre de nom b est aussi de type double. La ligne d'entête n'est pas terminée par un point virgule. Les noms des paramètres n'ont d'importance qu'au sein du corps de la fonction membre : ils servent à décrire comment la méthode utilise les valeurs qu'on lui transmet quand elle est appelée. An niveau de l'entête, les noms des paramètres désignent les paramètres **formels** (ou muets) de la fonction membre.

Remarque : dans ce chapitre, toutes les méthodes que l'on va définir seront **public** et **static**.

3.1.2 Le corps de la fonction membre calculHypotenuse()

Dans le **corps** de la méthode délimité entre accolades - **{** et **}** -, on écrit le **instructions exécutées** lorsque la fonction membre est appelée :
- il y a tout d'abord la déclaration d'une **variable locale (de la méthode calculHypotenuse())** nommée res de type double qui va servir à stocker le résultat du calcul. Cette variable locale (comme toute variable locale) n'est **visible** (ou **accessible**) **que dans le bloc** (ici la fonction) **où elle est déclarée**.
- ensuite, il y a un appel de la fonction sqrt(). Avant l'appel proprement dit, le système calcule la valeur de l'expression (a*a + b*b) et la transmet à la fonction sqrt(). La fonction sqrt() calcule la racine carrée de la valeur transmise, et la renvoie. La valeur renvoyée est alors affectée dans la variable res.
- enfin l'instruction return res; entraîne la fin de l'exécution de la méthode et renvoie la valeur contenue dans la variable res à la fonction appelante.

Remarques :
- la fonction **sqrt()** est une méthode publique statique de la classe **Math**. Cette classe ne contient que des fonctions membres statiques et fait partie du package java.lang (rappel : java.lang est le seul package dont toutes les classes sont importées automatiquement, c'est pourquoi il n'est pas nécessaire de l'importer). Vous pouvez consulter sa documentation à l'adresse suivante :
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Extrait :

```
static double                    sqrt(double a)
                                 Returns the correctly rounded positive square root of a double value.
```

On en déduit que, pour appeler la méthode sqrt(), il faut lui transmettre une valeur de type double, et que cette méthode renvoie en retour une valeur de type double.
Lors de l'appel de la fonction sqrt() (dans la corps de la fonction calculHypotenuse()), le point "**.**" indique que l'on appelle la fonction sqrt() qui appartient à la classe Math.

Q1 Consulter la documentation de la classe Math.

3.2 Appels de la fonction membre calculHypotenuse()

hypom = calculHypotenuse (am, bm); // 1er appel
hypom = calculHypotenuse (am + 2.5 , 7.8); // 2ème appel

Les variables am, bm et hypom sont des variables locales de la **fonction appelante** (ici la fonction main()). La fonction calculHypotenuse() est la **fonction appelée.**

Lors de l'appel de la fonction calculHypotenuse(), il faut lui transmettre 2 paramètres. Les paramètres fournis à ce niveau se nomment les paramètres **effectifs**. **Dans le cas de paramètres de type "primitif",** on peut utiliser **n'importe quelle expression** du type correspondant comme paramètre

effectif (cf. 2ème appel); au bout du compte, c'est la **valeur** de cette expression qui sera transmise à la fonction lors de l'appel.
L'exécution de chaque appel entraîne :
- l'évaluation (ou le calcul) de chaque paramètre effectif;
- **dans le cas de paramètres de type "primitif",** pour chaque paramètre formel de la fonction, il y a **création d'une variable initialisée à la valeur de l'expression correspondante** : le mode de **transmission est dit par valeur.** Les variables ainsi créées seront **détruites au retour de l'appel** (à la fin de l'exécution de la fonction).

Ainsi, lors du 2ème appel de la fonction :
- une variable nommée a est créée et initialisée à la valeur de l'expression am + 2.5;
- une variable nommée b est créée et initialisée à la valeur 7.8.

- l'adresse de l'instruction qui suit l'appel est empilée. Pour le 2ème appel, il s'agit de l'adresse de l'instruction :

System.out.println("2. Longueur de l'hypoténuse du triangle rectangle");

- le contrôle est alors transféré à la 1ère instruction du corps de la fonction calculHypotenuse (). La variable locale de la fonction, nommée res est elle-même créée. Elle sera elle aussi détruite au retour de l'appel.
- lorsque le contrôle rencontre l'instruction return res; , l'exécution de la fonction appelée se termine et le contrôle est transféré à l'adresse de l'instruction précédemment empilée (cette adresse est dépilée) et la valeur de l'expression res est renvoyée à la fonction appelante (la fonction main()) qui, ici, l'affecte dans la variable hypom.

Remarques :
- lors de l'appel d'une fonction, il ne faut pas indiquer le type des paramètres effectifs;
- l'expression calculHypotenuse(am, bm) est du type de la valeur de retour de la fonction : double.
- les variables locales d'une fonction (comme ses paramètres formels) **ne sont visibles ou accessibles que dans le corps de la fonction correspondante**. Ces variables locales sont **créées au début de l'exécution** de la fonction et sont **détruites à la fin de son exécution**.

Q2 Tester ce premier exemple.

4. L'instruction return

Voici quelques règles générales concernant cette instruction.

- L'instruction **return** précise la **valeur** du résultat renvoyé, et, en même temps, elle **interrompt l'exécution de la fonction** en revenant dans la fonction qui l'a appelée.

```
public static int f() {
        int r ;
        …
        return r ;
}
```

La méthode f() est de type int, elle retourne la valeur de la variable r qui est de type int.

- Une méthode **void** peut contenir une instruction **return**, dans ce cas elle ne reçoit pas de paramètre, elle sert généralement à interrompre l'exécution de la méthode.

```
public static void g() {
        …
        if (condition)
                return ;
}
```

- L'instruction **return** peut être suivie de n'importe quelle expression (obligatoirement du type de la fonction).

Exemple : définition d'une fonction permettant de calculer un polynôme du second degré :

```
public static float fexple(float x, int b, int c) // entête de la fonction membre fexple ()
{ // début du corps de la fonction membre
        return (x * x + b * x + c) ;
} // fin du corps
```

- Une fonction peut ne rien renvoyer du tout. Le type de la valeur de retour est alors void. Dans ce cas et uniquement dans ce cas, la présence de l'instruction return est facultative; et si l'instruction return est absente, le retour est mis en place automatiquement par le compilateur à la fin du bloc qui définit la fonction. Exemple :

```
public static void affiche()      // entête de la fonction affiche ()
{
     System.out.print("Message 1\n");
     return;     // => retour dans la fonction appelante
                 // Cette instruction est ici facultative car c'est la dernière
}
```

Remarque : la fonction affiche() est ici sans paramètre et sans valeur de retour.

- Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat.

Illustration : appels des 2 fonctions précédentes

```
public static  void main(String[] args)
{
    // Appel de la fonction affiche()
    //resm = affiche();    // ne compile pas car on affiche() ne renvoie rien !
    affiche();

    //appel de la fonction fexple()
    fexple(5.4f, 2, 6);       // la valeur de retour de la fonction fexple() n'est pas utilisée.
                              // Comme cette fonction n'affiche rien, cet appel ne sert à rien !
}
```

- L'instruction return peut apparaître à plusieurs reprises dans une fonction (mais je vous conseille de l'éviter dans la mesure du possible). Dans le cas où l'instruction return apparaît à plusieurs reprises, il faut veiller à ce que tous les chemins d'exécution de la fonction renvoie une valeur.

Exemple :

```
// DEFINITION de la fonction absSom() :
// fonction qui calcule la somme de ses 2 paramètres et renvoie la valeur absolue de cette somme
public static double absSom (double u, double v)
{       // début du corps de la fonction
        double s ;
        s = u + v ;
        if (s > 0)
        {
                return (s);
        }
        else
        {
                return (-s);
        }
}       // fin du corps
```

Définition conseillée de la même fonction :

```
public static double absSom (double u, double v)  {
        double s ;
        s = u + v ;
        if (s < 0)
        {
                s = -s;
        }
        return s; // une seule instruction return à la fin de la définition de la fonction
}
```

Q3 Tester les exemples précédents.

5. Passage de paramètres (arguments) à une méthode

5.1 Paramètres de type primitif : les paramètres sont transmis par valeur

Rappel : pour chaque paramètre formel et **à chaque appel** d'une fonction, le système crée une variable initialisée à la valeur de l'expression correspondant au paramètre effectif transmis. Les variables **locales** ainsi créées seront **détruites au retour de l'appel** (à la fin de l'exécution de la fonction).

Exemple mettant en évidence les conséquences et les limitations de ce mode de transmission par valeur :

```
public class Echange {

        public static  void main(String[] args)          {
                int n = 10, p = 20;

                System.out.println("Dans main(), avant appel : " +  n + " ," +  p);
                echange(n,p);   // Appel de la fonction echange()
                System.out.println("Dans main(), après appel : " +  n + " ," +  p);
        }

        // DEFINITION de la fonction echange() :
        // Cette fonction tente de permuter les 2 paramètres qui lui sont transmis
        public static void echange(int a, int b)
        {
                int temp;
                System.out.println("Dans echange(), avant permutation : " +  a + " ," +  b);
                temp = a;
                a = b;
                b = temp;
                System.out.println("Dans echange(), après permutation : " +  a + " ," +  b);
        }
}
```

L'exécution du programme correspondant donne les résultats suivants :

```
Dans main(), avant appel : 10 20
Dans echange(), avant permutation : 10 20
Dans echange(), après permutation : 20 10
Dans main(), après appel : 10 20
```

Explication : lorsque la fonction echange() est appelée, le système crée 2 variables locales (a et b, qui correspondent aux 2 paramètres formels de cette fonction) qui sont distinctes des variables locales de

la fonction main() (n et p). A leur création, les variables a et b **reçoivent des copies** des variables n et p, on dit qu'ils sont transmis par **copie**. Ensuite, la fonction echange() modifie ses variables locales a et b qui sont détruites à la fin de son exécution. Comme la fonction echange() "travaille" sur des copies des variables n et p, on constate, au retour de l'appel de cette fonction, que les variables n et p n'ont pas été modifiées.

Conclusion : comme les paramètres de type primitif sont **transmis par valeur**, la fonction est **incapable de modifier les paramètres effectifs** qui lui sont transmis : les modifications éventuellement réalisées sur les paramètres formels ne sont visibles qu'à l'intérieur de la fonction echange().

Q4 Tester cet exemple.

5.2 Paramètres objets: les paramètres sont transmis par copie de leur référence

Pour chaque paramètre de type objet et **à chaque appel** d'une fonction, le système crée une **variable référence initialisée avec la référence (ou l'adresse)** de l'objet paramètre transmis correspondant. Les variables **références locales à la fonction** sont donc des **copies des références des variables passées**, elles sont alors créées puis seront **détruites au retour de l'appel** (à la fin de l'exécution de la fonction).

Exemple :

```
public class Ref1 {

        public static  void main(String[] args)   {
                int [] tabm;      //(*1)
                tabm = new int [] {4, 5, 6};       //(*2)
                int i;
                String s = "Avant appel de la fonction" ;        //(*3)
                System.out.print("Dans main(), avant appel : ");
                for (i = 0; i < tabm.length ; i++)
                        System.out.print(tabm[i] + " ");
                System.out.println("\n s = "+s);
                fct(tabm, s);                        // Appel de la fonction fct()
                System.out.print("Dans main(), après appel : ");
                for (i = 0; i < tabm.length ; i++)
                        System.out.print(tabm[i] + " ");
                System.out.println("\n s = "+s );
        }

        // DEFINITION de la fonction fct() :
        // elle incrémente tous les éléments d'un tableau d'entiers passés en paramètre
        public static void fct(int [] t ,  String ch) {
                ch = "string modifié par la fonction" ;
                for (int i = 0; i < t.length ; i++)
                        t[i]++ ;
        }
}
```

L'exécution du programme correspondant donne les résultats suivants :

```
Dans main(), avant appel : 4 5 6
  s = Avant appel de la fonction
Dans main(), après appel : 5 6 7
  s = Avant appel de la fonction
```

Explication :

(*1) : **déclaration d'une variable référence** nommée tabm destinée à contenir l'adresse mémoire (ou la référence) d'un tableau de int.

(*2) : l'opérateur **new permet de créer un objet, ce qui implique la réservation de l'espace mémoire occupé par cet objet.** Ici, l'objet créé est un tableau contenant 3 éléments de type int; ils sont respectivement initialisés à 4, 5 et 6. Après avoir créer l'objet, **l'opérateur new renvoie l'adresse mémoire de l'objet créé et cette dernière est affectée dans tabm.**
(*3) : création d'un objet de type String contenant le texte "**Avant appel de la fonction".**

Lorsque la fonction fct() est appelée, le système **crée :**

- **une variable référence nommé t et initialisée avec la référence de l'objet tableau créé dans la fonction main(),**
- **un objet String ayant la même référence que l'objet Sting créé dans le main().**

A cet instant de l'exécution du programme, **les 2 variables références t et tabm pointent sur le même objet tableau d'entiers** ! Attention, seule la variable référence t est visible (ou accessible) dans la fonction fct(). Il en est de même pour la variable String ch dont la référence est initialisée à la référence de l'objet String s du main().
La fonction fct() est capable de modifier le contenu de l'objet tableau dont la référence (ou l'adresse) lui est transmise en paramètre.
En revanche, la fonction fct() ne peut modifier l'objet String passé car les objets String sont des objets constants (objets immutables).
En conclusion, une fonction peut modifier **des paramètres objets à condition qu'ils ne soient pas constants et qu'ils soient munis des méthodes/possibilités appropriées.**

Q5 Tester l'exemple ci-dessus.

Q6 La classe StringBuilder
La classe **StringBuilder** permet de créer des String modifiables, à l'inverse de la classe String qui crée des String "immutables". Cette classe propose de nombreuses méthodes pour créer/modifier une chaine de caractères.
►Création d'un **StringBuilder** :

```
StringBuilder s = new StringBuilder("l'objet chaine créé") ;
```

►Modification compléte de la chaine contenue :

```
s.delete(0, ch.length());
s.append("le même objet chaine modifié");
```

►Rendre opérationnelle la fontion fct() précédente pour modifier la chaine transmise en utilisant dans le programme la classe **StringBuilder** à la place de la classe **String**.
►On peut utiliser la classe **StringBuffer** à la place de la classe **StringBuilder** si le String est manipulé par plusieurs threads.

6. Coercition ou "cast"

En informatique, la coercition ou "cast" est l'action de convertir une expression (qui a forcément un type) dans un nouveau type. On l'appelle aussi **forçage de type** ou **transtypage**.

6.1 Exemple 1 :

```
double d = 2.2000000000001;
float f = 2.2000000000001f;

System.out.println(d);
System.out.println(f);

System.out.println("Après conversion de float en double");
d = f;              // (*1)
System.out.println(d);
```

```
d = 2.2000000000001;


f = (float) d;      // (*2) cast explicite nécessaire car sinon erreur de compilation
System.out.println("Après conversion de double en float");
System.out.println(f);
```

(*1) : Cette instruction demande à la JVM de **convertir une valeur de type float en double** et d'affecter le résultat dans la variable d. **Cette conversion est non dégradante** car :

- la précision du type double est meilleure;
- la plage des réels que l'on peut coder avec un double englobe la plage des réels qu'on peut coder avec un float.

(*2) : Cette instruction demande à la JVM de **convertir une valeur de type double en float** et d'affecter le résultat dans la variable f. **Cette conversion est dégradante, il y a un risque de perte d'information**. Donc, il est nécessaire d'**indiquer explicitement** au compilateur qu'il doit convertir la valeur contenue dans la variable d en float. L'instruction f = d; ne compile pas.

Q6 Tester l'exemple ci-dessus.

6.2 Conversions dégradantes / conversions non dégradantes

Représentation des différents types primitifs mis en jeu :

byte → short → int → long → float → double

Si le nouveau type de l'expression est situé à droite du type de départ, la conversion est non dégradante et donc l'utilisation d'un cast explicite n'est pas nécessaire.

Dans le cas contraire, il s'agit d'une conversion dégradante, il y a un risque de perte d'information. Donc l'utilisation d'un cast explicite est nécessaire.

6.3 Exemples d'utilisation

Exemple 1 :

```
double d = 2.95;
int aa;
aa = (int)d;      // conversion dégradante : affecte dans aa la partie entière du nombre réel 2.95
System.out.println(aa);          // affiche 2
```

Exemple 2 :

```
public static int rand10 ()
{
        int res = (int) (Math.random() *10);    //conversion dégradante => cast explicite
        return res;
}
```

La méthode random() est une méthode statique de la classe Math :

```
static double                    random()
```

Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Q7 Tester cette fonction. Dans quelle plage (ou intervalle) la valeur entière renvoyée par cette fonction est-elle comprise?

Exemple 3 :

```
public static double arrondi(double A, int B)
{
        double res = ( (int) (A * Math.pow(10, B) + 0.5) ) / Math.pow(10, B);
```

```
        return res;
}
```

La fonction arrondi() renvoie un arrondi du nombre passé en 1er paramètre :

- 1er paramètre : nombre à "arrondir"
- 2ème paramètre : nombre de chiffres après la virgule
- valeur de retour : le nombre arrondi

La méthode pow() est une méthode statique de la classe Math :

```
static double                    pow(double a, double b)
                                 Returns the value of the first argument raised to the power of the second argument.
```

Q8 Tester cette fonction. Pour bien comprendre le travail qu'elle effectue, vous pouvez "décomposer" son code et ajouter des affichages.

7. Exercices

Exercice 1 : calcul de l'aire d'un rectangle

Ecrire un programme qui **définit** (en plus de la fonction main()) **et appelle** (dans la fonction main()) 4 fonctions :

- la fonction saisieLongueur(), sans paramètre, qui affiche un message à destination de l'utilisateur (l'invitant à saisir la longueur), réalise cette saisie et renvoie la valeur saisie;
- la fonction saisieLargeur() qui affiche un message à destination de l'utilisateur, réalise la saisie de la largeur et renvoie la valeur saisie;
- la fonction calculAireRectangle(), **sans affichage et sans saisie**, qui renvoie l'aire d'un rectangle dont la longueur et la largeur lui sont transmises en paramètre;
- la fonction afficheAire(), qui affiche le message "Valeur de l'aire du rectangle : " suivi de la valeur de cet aire qui lui est transmise en paramètre.

Exercice 2 :

1. Ecrire la définition d'une fonction sans paramètre qui saisit un entier et le retourne.
2. Ecrire la définition d'une fonction qui reçoit un entier en paramètre, sans affichage et sans saisie, qui renvoie un nombre aléatoire supérieur ou égal à 0 et strictement inférieur au paramètre reçu.

2. Réaliser un programme qui :

- génère un nombre aléatoire en appelant les fonctions créées aux questions précédentes;
- demande à l'utilisateur de deviner le nombre aléatoire généré jusqu'à ce qu'il le trouve. Il faudra le guider à l'aide de messages du type "Trop grand", "Trop petit".

Exercice 3 :

Refaire l'exercice 5 du cours/TP 2 (Les tableaux) en utilisant 2 fonctions **sans affichage et sans saisie**.

La 1ère fonction controleDate() doit contrôler la validité d'une date (jour + mois) transmise en paramètre et renvoyer true si la date est valide et false dans le cas contraire.

La 2ème fonction getNbJourDebutAn() doit renvoyer le nombre de jours écoulés depuis le début de l'année jusqu'à la date transmise en paramètre (on supposera que l'année en cours est l'année 2017).

Exercice 4 :

Soit le programme suivant :

```
public class Ref2 {

	public static  void main(String[] args)
	{
		int [] tabm = null ;
		// la variable référence tabm est initialisée avec la référence null

		int i;

		fct(tabm);	// Appel de la fonction fct()
		System.out.print("Dans main(), après appel : ");
		for (i = 0; i < tabm.length ; i++)
			System.out.print(tabm[i] + " ");
		System.out.println();
	}

	// DEFINITION de la fonction fct() :
	public static void fct(int [] t)
	{
		t = new int []{ 10, 20 } ;
	}
}
```

Tester ce programme. Justifier le résultat obtenu. Modifier la définition de la fonction fct() (et son appel) afin de récupérer dans la fonction main() une référence sur l'objet tableau créé dans la fonction fct().

Exercice 5 :

Le programme source TableauE.java est fourni en suivant.

1. Compléter la définition de la fonction menu() afin de proposer les choix supplémentaires suivants :
   - 2 : affichage des éléments du tableau;
   - 3 : ajout d'une **même** valeur saisie par l'utilisateur à tous les éléments du tableau;
   - 4 : recherche de la valeur minimale présente dans le tableau;
   - 5 : recherche de la valeur maximale présente dans le tableau;
   - 6 : calcul de la valeur moyenne des éléments du tableau;
   - 7 : tri dans l'ordre croissant des éléments du tableau;

2. Pour chaque choix supplémentaire, définir une nouvelle fonction. Les fonctions correspondant aux 4 derniers choix devront être sans affichage et sans saisie.

3. Compléter la définition de la fonction main() et tester votre programme. Il faudra penser à afficher un message d'erreur quand le choix de l'utilisateur n'est pas valide.

Remarque : pour trier le tableau, vous pouvez utiliser la méthode statique **sort()** de la classe **Arrays** (cette classe fait partie du package java.util).

4. Ajouter une fonction qui affecte, dans chaque élément d'un objet tableau de doubles, une valeur tirée **aléatoirement** et comprise dans une plage de valeurs fixée par l'utilisateur. Il faut bien entendu ajouter le choix correspondant dans le menu.

5. Ajouter une fonction dans laquelle l'utilisateur saisit le nombre d'éléments (ou la taille) du tableau référencé par la variable tabm (déclarée dans la fonction main()) et retourne ainsi le tableau avec sa nouvelle taille.

```
/*************************************************************************
* Nom : Exercice 5 cours 3
*  Description : Manipulation d'un tableau de réels
*************************************************************************/
import java.util.Scanner;       // classe utilisée pour réaliser les saisies

public class TableauE {

/* Déclaration d'une variable objet de type Scanner et création avec l'opérateur new,
* l'objet sc est accessible à toutes les méthodes (fonctions) de la classe TableauE
* sc est un attribut de la classe TableauE.
*/
        static Scanner sc = new Scanner(System.in);

        public static  void main(String[] args)     {
                double [] tabm = new double[5];
                int choixm;     // permet de stocker le choix de l'utilisateur

                System.out.println("Ce programme permet la manipulation d'un tableau de
                                                                                        réels.\n");
                do  {   // Appel de la fonction Menu()
                        choixm = menu();
                        if (choixm == 1)
                                // cette structure algorithmique n'est pas forcément à conserver
                                // quand il y a plusieurs choix valides possibles
                        {
                                // Appel de la fonction saisirTab()
                                saisirTab(tabm);
                        }

                }while (choixm != 0);
                System.out.println("Fin du programme\n\n");
        }

        /* Définition de la fonction menu() : A COMPLETER
        Fonction qui affiche le menu, saisit et renvoie le choix de l'utilisateur*/
        public static int menu()  {
                // Déclaration des variables locales de la fonction menu()
                int ch;
                System.out.print("********** Menu ********************\n");
                System.out.print("\tPour saisir les éléments du tableau, tapez \t\t\t1\n");
                System.out.print("\tPour sortir, tapez \t\t\t\t\t0\n\n");
                System.out.print("--> Entrez votre choix : ");
                ch = sc.nextInt();
                System.out.print("\n");
                return ch;
        }
```

```
	/* Définition de la fonction saisirTab() :
	Fonction qui permet de saisir les éléments d'un tableau de doubles
	- 1er paramètre : référence qui pointe sur l'objet tableau de doubles
	- Pas de valeur de retour */
	public static void saisirTab(double [] t)  {
		// Déclaration des variables locales de la fonction saisirTab()
		int i;
		for (i = 0; i < t.length; i++ )    {
			System.out.print("-->Saisir l'élément numéro " + (i+1) +  " : ");
			t[i] = sc.nextDouble();
		}
	}
}
```

Exercice 6 : on étudie toutes les étapes pour effectuer un codage simple d'un fichier texte.

Soit le code suivant qui permet de lire un fichier texte et qui affiche son contenu à l'écran.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class LireFichier {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		byte [] b = null;
		try {
		   b = Files.readAllBytes(Paths.get("C:\\Users\\jeanjean\\Desktop\\texte.txt"));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			System.out.println("Erreur de lecture sur le fichier");
			System.exit(0);
		}
		int taille = b.length;
		char []c = new char[taille] ;
		for (int i=0; i<taille; i++)
			c[i] = (char) b[i];
		for (int i=0; i<taille; i++)
			System.out.print(c[i]);
	}
}
```

L'instruction

```
b = Files.readAllBytes( Paths.get("C:\\Users\\jeanjean\\Desktop\\texte.txt"));
```

lit le fichier dont le nom complet (le « path ») est donné en argument et renvoie les octets lus dans un tableau de bytes.
On peut étudier la documentation Java de la classe **Files**.

61 Créer un fichier texte ASCII avec des majuscules, des minuscules, des chiffres, des caractères accentués…Par exemple :

Bonjour Monsieur Duchemin
Nous vous informons d'un rendez-vous secret le 15 janvier 2021.
Venez seul, sans arme.
El comandante

Modifier le nom donné du fichier et son chemin en concordance avec votre environnement.
Etudier et tester ce programme.

62 Donner un nom de fichier qui n'existe pas. Tester le programme. Justifier le résultat.

On se propose d'écrire une nouvelle classe CoderFichier.java.

63 Créer une fonction **public static char[] lireFichier(String nomFichier)** qui reçoit en argument le nom complet d'un fichier et retourne son contenu dans un tableau de caractères.

64 On veut coder le fichier précédent en suivant un principe assez simple. Chaque lettre minuscule de ce fichier est remplacée par un caractère qui représente le code ascii de la lettre minuscule à coder plus un certain nombre n, les autres caractères étant inchangés.
Par exemple :

on code un 'a', si n=2, le caractère codé est 'c',
on code un 'r', si n=2, le caractère codé est 't',
on code un 'v', si n=2, le caractère codé est 'z'.

Le caractère codé correspond au caractère de la table ascii à coder plus le décalage donné.
Le caractère codé est ainsi obtenu en additionnant la constante n au caractère à coder.

On demande de coder les 2 fonctions suivantes :

public static boolean estMinucule(char c)
qui retourne true si le caractère est une lettre minuscule.
public static char coderCar(char c, int n)
qui reçoit en paramètre le caractère à coder et le nombre n représentant le saut à faire dans la table ascii, et qui retourne le caractère codé.

Ecrire la fonction principale qui code le fichier précédent en appelant les fonctions demandées et affiche à l'écran le texte codé. Vérifier avec soin le fonctionnement.

65 Ecrire la fonction public **static char[] decoderMessage(char []texteCode, int n)** qui reçoit les caractères codés et le saut n utilisé et retourne le texte décodé. Tester cette fonction.

66 On veut sauvegarder le texte codé dans un fichier.
Ecrire la fonction **public static void ecrireFichier(String nomFichier, char []texte)** qui écrit le texte donné dans le fichier dont le nom est fourni. On peut utiliser la méthode statique **Files.write(…)** de la classe **Files**.