



# Les méthodes de classe

CSC 3101

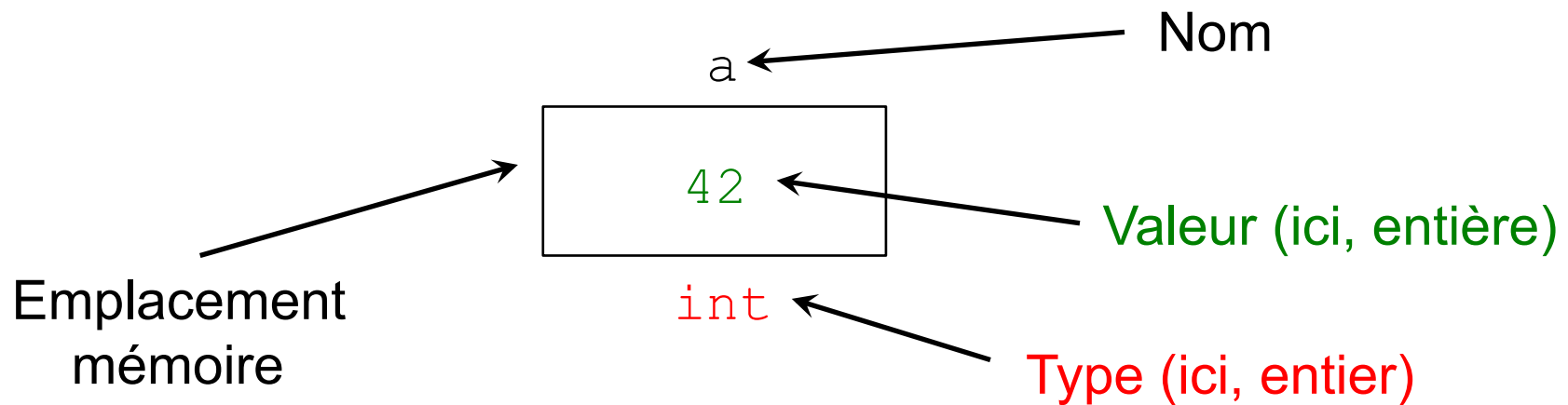
Algorithmique et langage de programmation

Gaël Thomas



# Rappel : la variable

- Une variable **est** un emplacement mémoire
  - Qui possède **un nom**, **un type** et **une valeur**



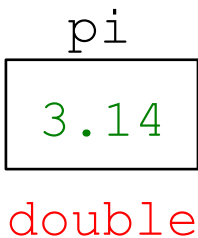
# Rappel : primitif versus référence

## ■ Une variable contient

- Soit une **valeur de type dit primitif**

(`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`)

```
double pi = 3.14;
```

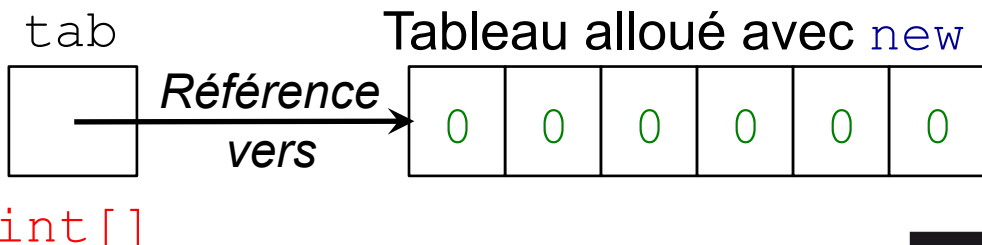


The diagram shows a box labeled 'pi' containing the value '3.14'. Below the box is the word 'double'.

- Soit une **valeur de type dit référence**

(identifiant unique de tableau ou de `String`)

```
int[] tab = new int[6];
```



The diagram shows a box labeled 'tab' containing a reference arrow pointing to a memory block labeled 'Tableau alloué avec new'. The memory block contains six zeros. Below the box is the text 'int[]'.

# Qu'est-ce qu'une méthode ?

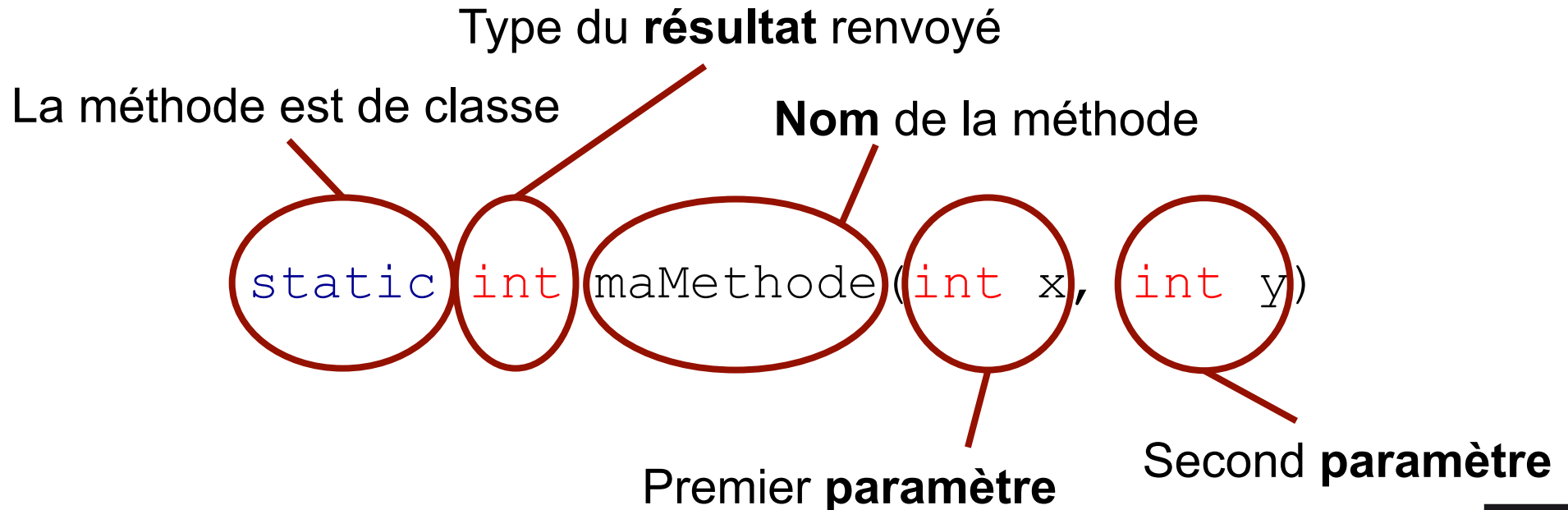
- Une méthode est un regroupement d'instructions
  - Création d'une macro-instruction
    - ⇒ Permet de réutiliser du code
  - Peut prendre des paramètres et renvoyer un résultat

# Qu'est-ce qu'une méthode ?

- Une méthode est un regroupement d'instructions
  - Création d'une macro-instruction
    - ⇒ Permet de réutiliser du code
  - Peut prendre des paramètres et renvoyer un résultat
- Dans ce cours, on étudie les méthodes **de classe**
  - Il existe aussi les méthodes d'instance, la différence sera expliquée en CI4

# Définition d'une méthode de classe (1/2)

- Une méthode de classe possède
  - Un **nom** ( $\Leftrightarrow$  nom de la macro-instruction)
  - Une liste de **paramètres** d'entrées sous la forme `type symbol`
  - Le type de **résultat** renvoyé (`void` si pas de résultat)



# Définition d'une méthode de classe (2/2)

- Une méthode de classe possède un corps délimité par { et }
  - Le corps contient une suite d'instructions
  - Termine avec `return resultat;` (`return;` si pas de résultat)

```
static int maMethode(int x, int y) {  
    if(y == 0) {  
        System.out.println("div par 0");  
        return -1;  
    }  
    return x / y;  
}
```

Corps de la méthode

Termine la méthode de classe

# La méthode main

Nom spécial qui indique que le programme commence ici

Ne renvoie pas de résultat

```
public static void main(String[] args)
```

Prend un unique paramètre :  
les arguments passés au programme

Déclaration magique  
(expliquée en cours 4)



# Invocation d'une méthode de classe

## ■ Utilisation

- Invocation avec `nomMethode (parm1, parm2...)`
- Après l'invocation, l'expression est remplacée par le résultat

```
class MaClass {  
    static int add(int x, int y) {  
        return x + y;  
    }  
  
    public static void main(String[] args) {  
        int res = add(1, 2);  
        System.out.println("1 + 2 = " + res);  
    }  
}
```

Invoque add avec  
x valant 1 et  
y valant 2

# Invocation d'une méthode de classe

## ■ Utilisation

- Invocation avec `nomMethode (parm1, parm2...)`
- Après l'invocation, l'expression est remplacée par le résultat

```
class MaClass {  
    static int add(int x, int y) {  
        return x + y;  
    }  
    Exécute add puis remplace par le résultat (ici 3)  
    public static void main(String[] args) {  
        int res = add(1, 2);  
        System.out.println("1 + 2 = " + res);  
    }  
}
```

# Invocation

Attention

se

- Util...
  - Les méthodes dont déclarées dans la classe, pas dans d'autres méthodes
  - A... expression est remplacée par le résultat

```
class MaClass {  
    static int add(int x, int y) {  
        ...  
    }  
  
    public static void main(String[] args) {  
        ...  
    }  
}
```

# La surcharge de méthode

- Java permet de **surcharger** des méthodes
  - Deux méthodes peuvent avoir le **même nom** pourvu qu'elles n'aient **pas les mêmes paramètres**
  - La méthode appelée est sélectionnée en fonction du type des arguments

```
class Test {  
    static void f(double x) { ... }  
    static void f(int x) { ... }  
    static void g() { f(42); }  
}
```

Appel `f(int x)` car  
42 est un entier

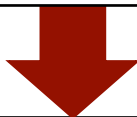


# Mais où est le `return` du `main` ?

- Si une méthode ne retourne rien, un `return` est implicitement ajouté à la fin du corps de la méthode

```
public static void main(String[] args) {  
    int res = add(1, 2);  
    System.out.println("1 + 2 = " + res);  
}
```

Réécrit en



```
public static void main(String[] args) {  
    int res = add(1, 2);  
    System.out.println("1 + 2 = " + res);  
    return;  
}
```

# Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode
- Lors d'une invocation de méthode, l'environnement d'exécution
  - Crée un **cadre d'appel** pour accueillir les variables locales/param.
  - Crée les variables locales/paramètres dans le cadre
  - Affecte les paramètres
- À la fin de l'invocation, l'environnement d'exécution
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

# Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode

- Lors d'une invocation, un **cadre d'appel** est créé.  
= *call frame* en anglais  
L'ensemble des variables locales/paramètres est créé dans cet environnement d'exécution.
  - Crée l'environnement d'exécution
  - Crée les variables locales/paramètres
  - Affecte les valeurs aux variables locales/paramètres

- À la fin de l'invocation, l'environnement d'exécution est détruit.
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

# Variables locales (2/2)

- Avant l'entrée dans `main`
  - Le tableau des arguments est initialisé

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

Tableau des  
arguments



# Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de main

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
    main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

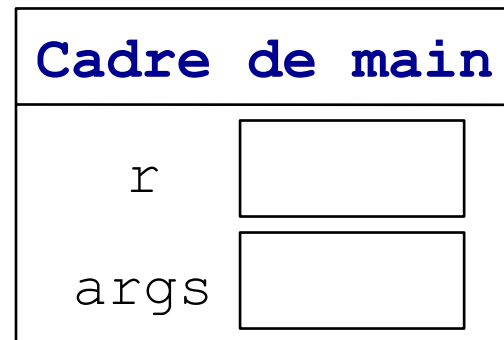
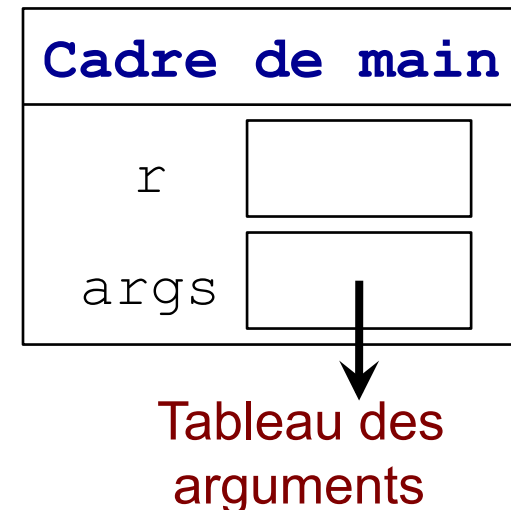


Tableau des arguments

# Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de `main`
  - Affectation paramètre `args` (référence vers tab. arguments)

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
    main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```



# Variables locales (2/2)

- À l'entrée dans add
  - Création cadre add

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
→ int r = add(1, 2);  
    }  
}
```

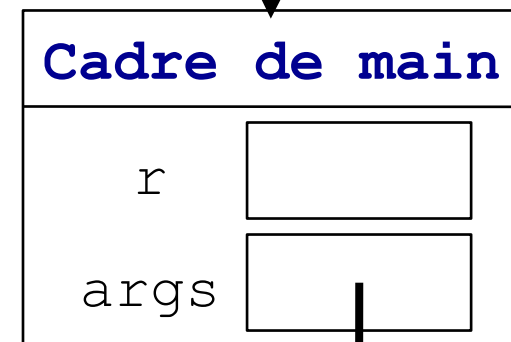
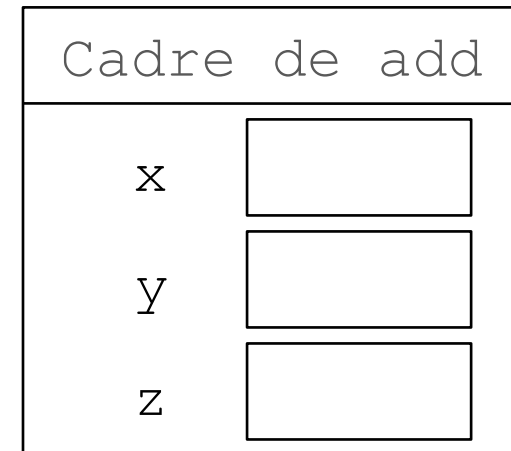


Tableau des arguments

# Variables locales (2/2)

- À l'entrée dans `add`
  - Création cadre `add`
  - Affectation paramètres

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

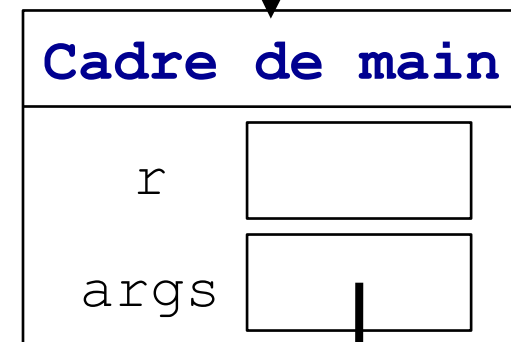
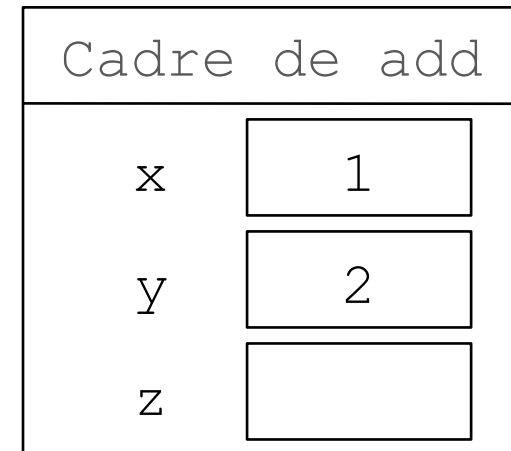


Tableau des arguments

# Variables locales (2/2)

- À l'entrée dans `add`
  - Activation du cadre de `add`

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

Cadre actif  
en bleu/gras

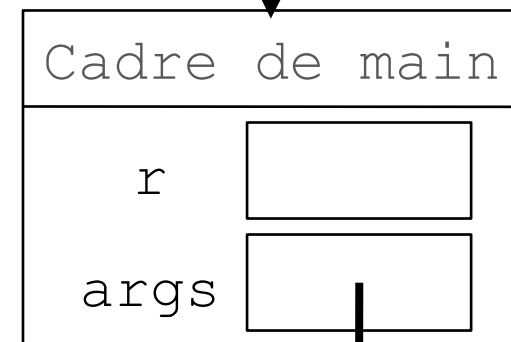
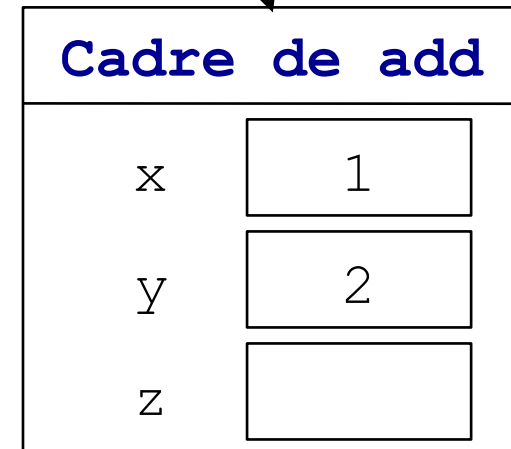


Tableau des  
arguments

# Variables locales (2/2)

## ■ Pendant add

- Utilisation des variables locales du **cadre actif**

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

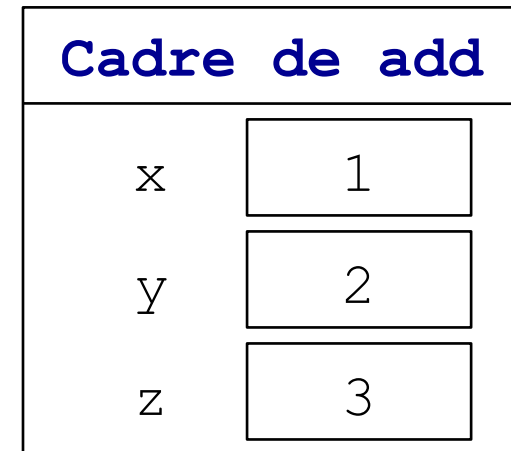


Tableau des arguments

# Variables locales (2/2)

- À la sortie de `add`
  - Activation cadre précédent

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
        int r = add(1, 2);  
    }  
}
```

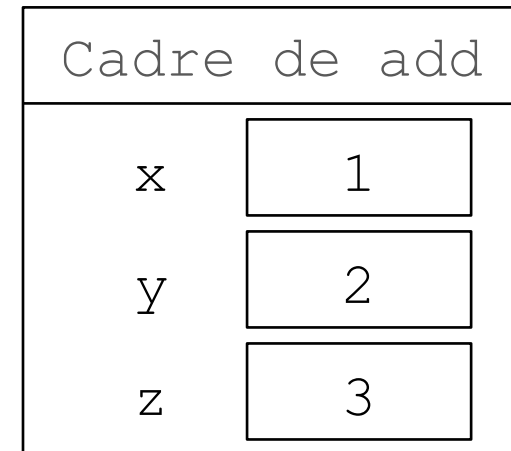


Tableau des arguments

# Variables locales (2/2)

- À la sortie de `add`
  - Activation cadre précédent
  - Suppression cadre précédent + récupère valeur de retour

```
class MaClass {
    static int add(int x, int y) {
        int z = x + y;
        return z;
    }
    public static void
        main(String[] args) {
        int r = add(1, 2);
    }
}
```

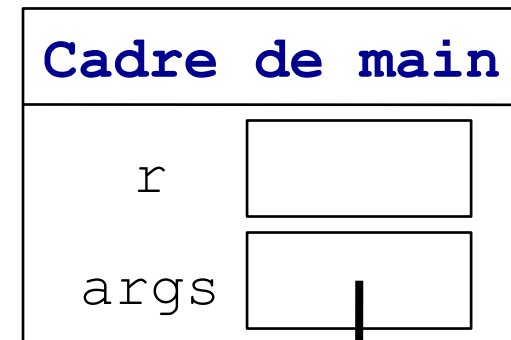



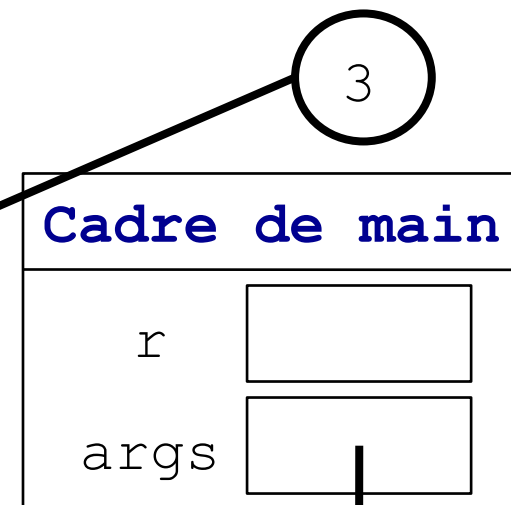
Tableau des arguments



# Variables locales (2/2)

- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
→ int r = a (3 ?);  
    }  
}
```



# Variables locales (2/2)

- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat

```
class MaClass {  
    static int add(int x, int y) {  
        int z = x + y;  
        return z;  
    }  
    public static void  
        main(String[] args) {  
→ int r = add(1, 2);  
    }  
}
```

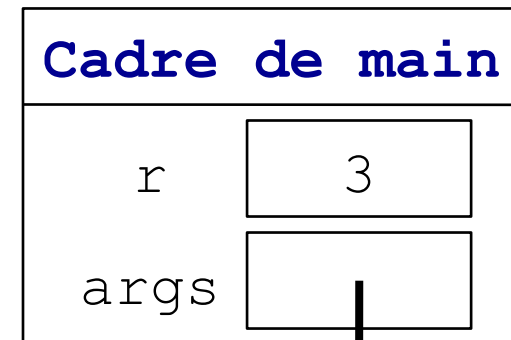
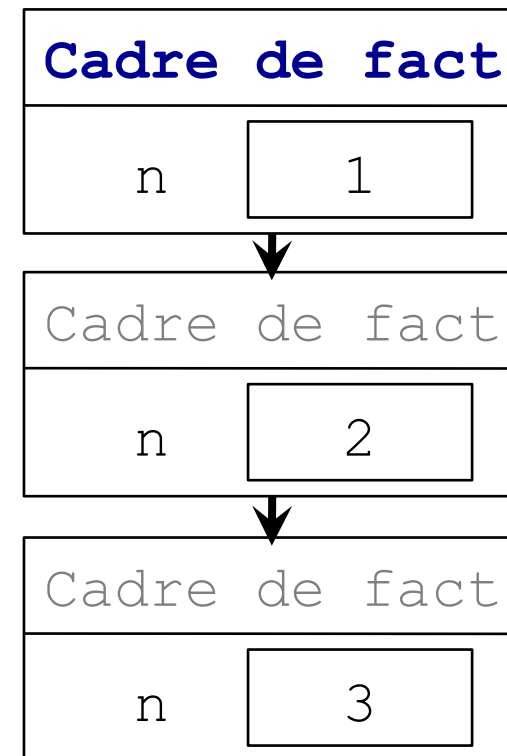


Tableau des arguments

# Variables locales et appels récursifs

- Si une méthode s'appelle elle-même
  - Un nouveau cadre à chaque appel  
⇒ de nouvelles variables locales à chaque appel

```
static int fact(int n) {  
    if(n < 1)  
        return 1;  
    else  
        return n * fact(n - 1);  
}  
/* fact(3) appelle fact(2)  
   qui appelle fact(1) */
```



# Passage par valeur et par référence

- Le passage des arguments peut se faire par
  - Valeur : l'appelé reçoit une copie d'une valeur  
⇒ la copie et l'originale **sont différentes**
  - Référence : l'appelée reçoit une référence vers une valeur  
⇒ la valeur est **partagée** entre l'appelant et l'appelé
  
- En Java :
  - Passage par valeur pour les 8 types primitifs  
(`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` et `double`)
  - Passage par référence pour les autres types  
(`String` et tableaux)

# Passage par valeur – exemple

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

→ 

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```

Cadre de f	
x	<input type="text"/>
y	<input type="text"/>

# Passage par valeur – exemple

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```



Cadre de f	
x	1
y	

# Passage par valeur – exemple

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```



Cadre de f	
x	1
y	2

# Passage par valeur – exemple

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```

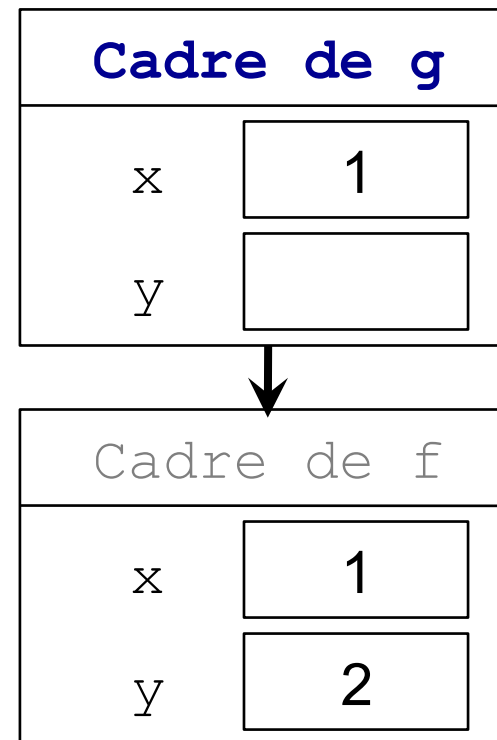
Cadre de f	
x	1
y	2



# Passage par valeur – exemple

- À l'entrée dans g
  - Le `x` du cadre de `f` **est copié** dans le `x` du cadre de `g`

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}  
  
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```

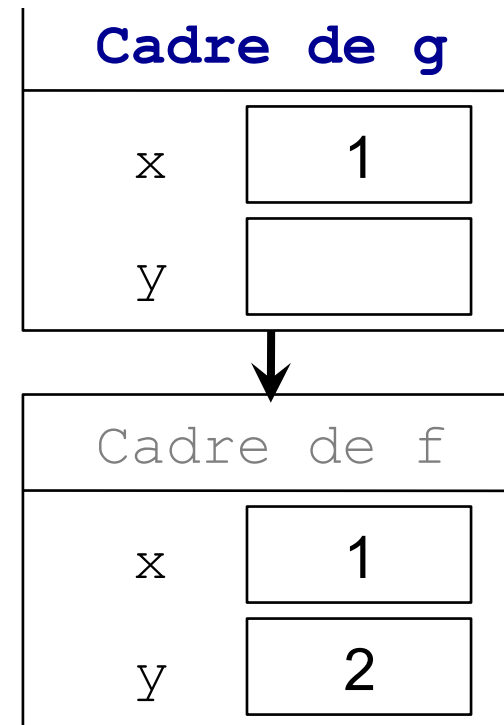


## Attention !

Le fait que les variables de `f` et `g` aient le même nom n'a aucune influence sur l'exécution

Si les variables de `g` se nommaient `a` et `b`, le programme fonctionnerait exactement de la même façon !

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}  
  
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```

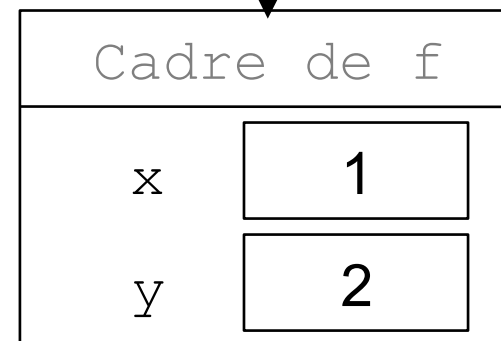
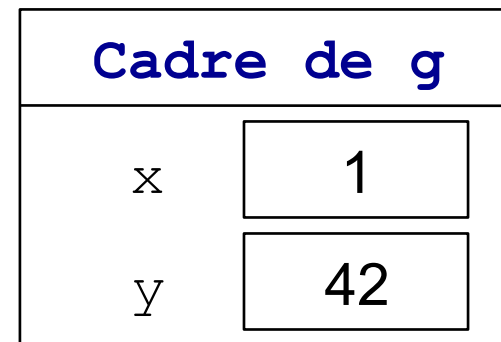


# Passage par valeur – exemple

- L'affectation du  $y$  de  $g$  ne change pas la valeur du  $y$  de  $f$

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```

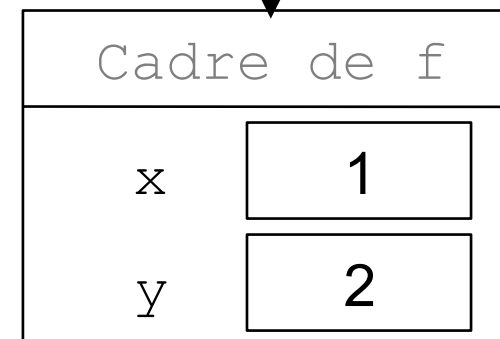
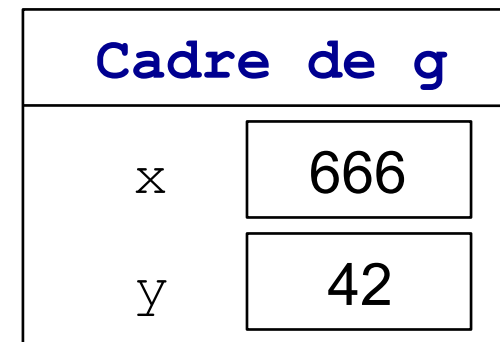


# Passage par valeur – exemple

- `g` modifie la copie de la variable `x` de `f`, pas celle de `f`

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```



# Passage par valeur – exemple

Pour résumer :  
Les variables de l'appelant  
ne sont **jamais** modifiées par l'appelé

```
static void g(int x) {  
    int y = 42;  
    x = 666;  
}
```

```
static void f() {  
    int x = 1;  
    int y = 2;  
    g(x);  
}
```



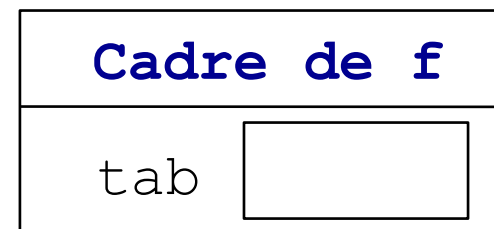
Cadre de f	
x	1
y	2

# Passage par référence – exemple

```
static void g(int[] t) {  
    t[0] = 42;  
}
```

→ 

```
static void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
}
```

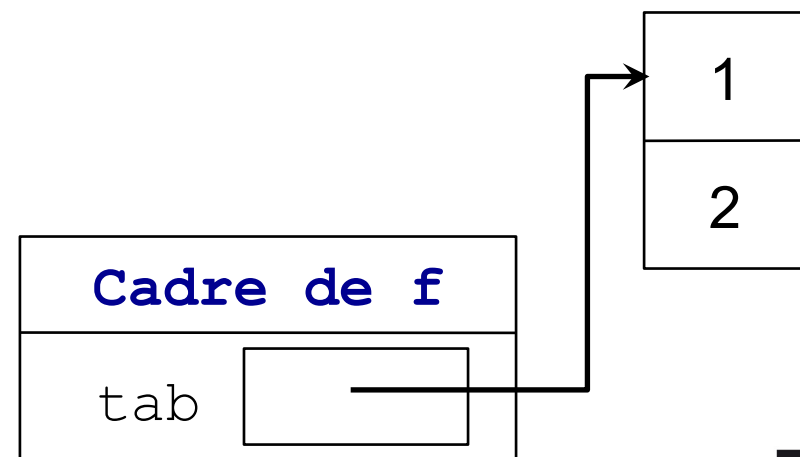


# Passage par référence – exemple

## ■ Rappel :

- Un tableau est alloué dans la mémoire
- La variable `tab` contient une **référence** vers ce tableau

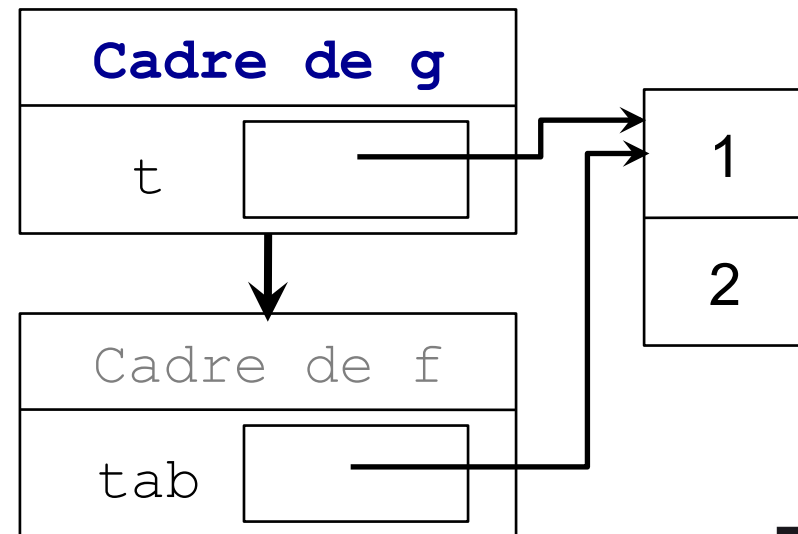
```
static void g(int[] t) {  
    t[0] = 42;  
}  
  
static void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
}
```



# Passage par référence – exemple

- À l'entrée de g
  - t reçoit une copie de tab
  - tab étant une référence, **t et tab référence le même tableau**  
⇒ le tableau est passé par référence

```
static void g(int[] t) {  
    t[0] = 42;  
}  
  
static void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
}
```

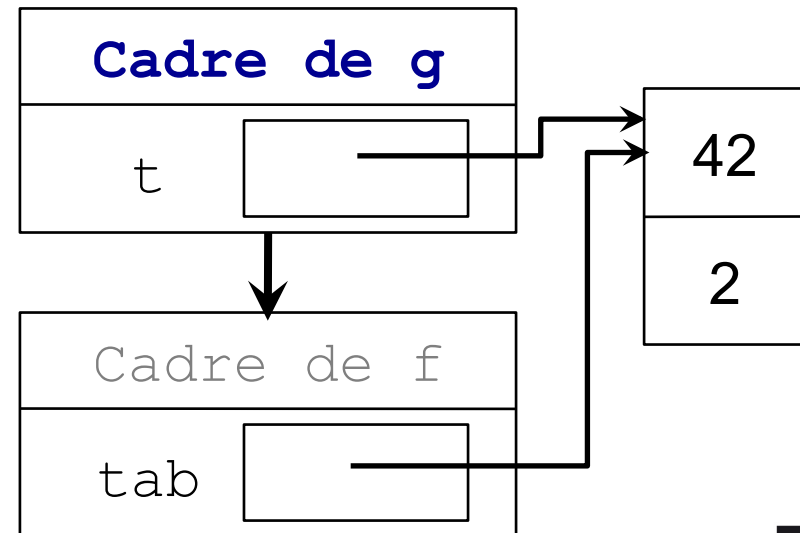




# Passage par référence – exemple

- La modification dans `g` modifie le tableau référencé par `tab`

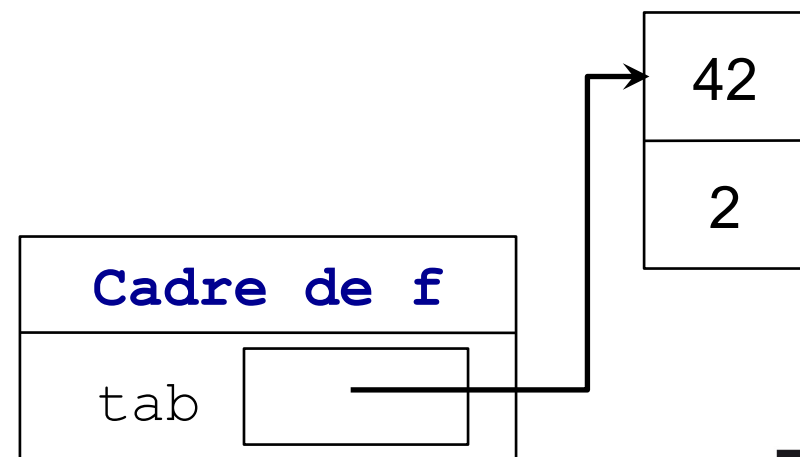
```
static void g(int[] t) {  
    t[0] = 42;  
}  
  
static void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
}
```



# Passage par référence – exemple

- L'appelant peut donc modifier une valeur passée par référence

```
static void g(int[] t) {  
    t[0] = 42;  
}  
  
static void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
}
```



# Passage par valeur et par référence

Pour résumer :

Les variables de l'appelant  
ne sont **jamais** modifiées par l'appelé

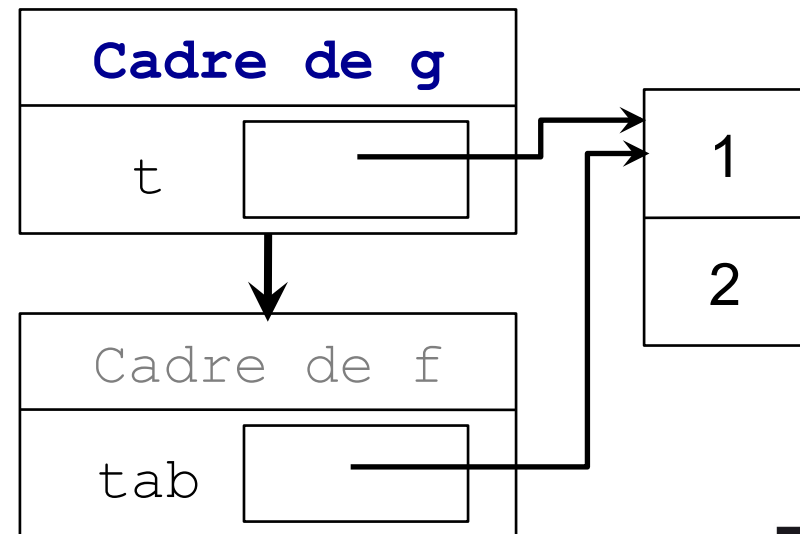
En revanche, les valeurs **référencées**  
à la fois par l'appelant et l'appelé  
**peuvent être** modifiées par l'appelé

# Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

```
void g(int[] t) {  
→ t = new int[3];  
  t[0] = 42;  
}
```

```
void f() {  
  int[] tab = { 1, 2 };  
  g(tab);  
  System.out.println(tab[0]);  
}
```

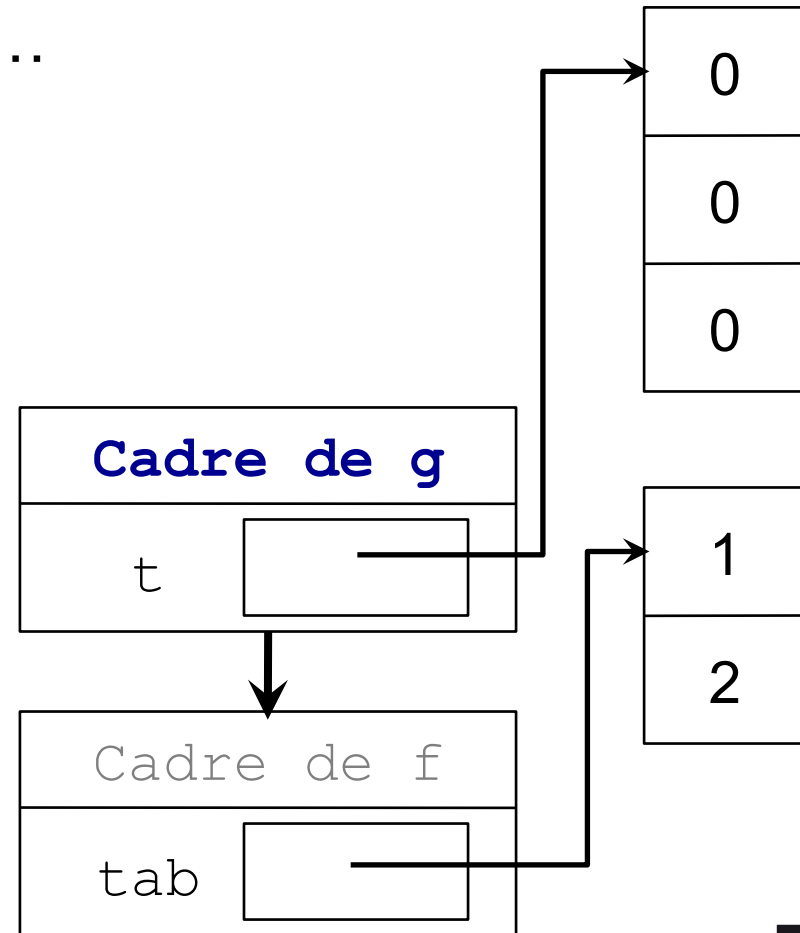


# Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

```
void g(int[] t) {  
    t = new int[3];  
    t[0] = 42;  
}
```

```
void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
    System.out.println(tab[0]);  
}
```

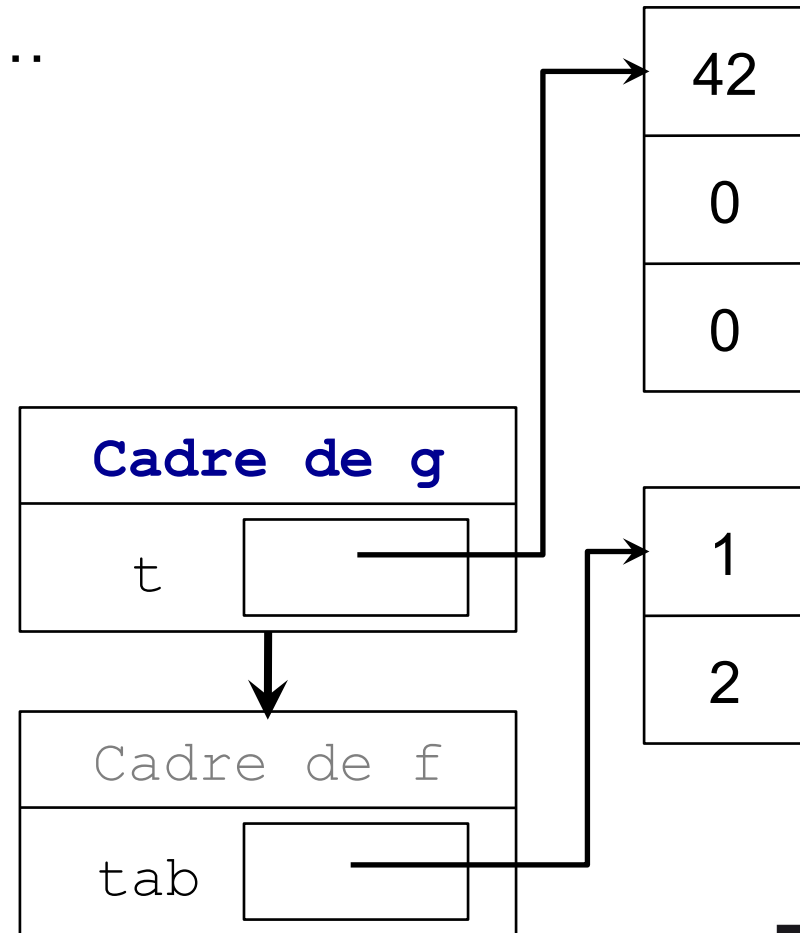


# Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

```
void g(int[] t) {  
    t = new int[3];  
    t[0] = 42;  
}
```

```
void f() {  
    int[] tab = { 1, 2 };  
    g(tab);  
    System.out.println(tab[0]);  
} /* affiche 1 !!! */
```



# Notions clés

- Déclaration d'une méthode de classe

```
static type nom(type arg1, ...) { corps }
```

- Appel d'une méthode de classe

```
nom(arg1, ...)
```

- Notions de cadre d'appel et de variables locales

- Le cadre d'appel est détruit à la fin de l'invocation

- Passage par valeur et passage par référence

- Par valeur pour les 8 types primitifs
- Par référence pour les autres types