

# Mémoire impérative

Programmation orientée objet en Java  
Structures de données et leurs aspects mémoire

**Valentin Honoré**

valentin.honore@ensiie.fr

FISA 1A

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- Comme en C, un tableau est une structure de données qui contient plusieurs éléments du même type

Un tableau de 6 entiers

1	17	4	7	2	13
---	----	---	---	---	----

# Allocation d'un tableau (1/2)

- ▶ Un tableau doit être alloué dans la mémoire avec `new type[n]`  
→ Allocation d'un tableau de  $n$  éléments ayant pour type `type`
- ▶ Par exemple : `new int[6]`

Alloue un tableau de 6 entiers

0	0	0	0	0	0
---	---	---	---	---	---

## Allocation d'un tableau (2/2)

- ▶ L'opérateur `new` renvoie une **référence vers un tableau** (une référence est un **identifiant unique d'une structure de données**)
- ▶ Par exemple, `new int[6]` renvoie une référence vers ce tableau :

0	0	0	0	0	0
---	---	---	---	---	---

Note : Java met à 0 chaque élément lors d'une allocation

- ▶ **il n'existe pas de variable de type tableau en Java !**
- ▶ En revanche, on peut déclarer une variable de type référence vers un tableau :  
`type[] var ;`

→ var est une variable de type **référence vers un tableau**

→ var contient des éléments de type `type`

# Exemple de déclaration

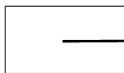
Affectation de la référence

tab référence un tableau de `int`

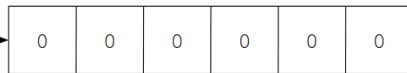
Allocation d'un tableau de 6 `int`

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

Variable `tab`



Référence vers



Le tableau alloué avec `new`

- ▶ On peut aussi allouer un tableau et l'initialiser avec

`type[]` `tab` =  $\{x_1, \dots, x_n\}$ ;

- ▶ Par exemple :

`double[]` `tab` = {2.3, 17.0, 3.14, 8.83, 7.26};

- ▶ En détails, le programme va
  - ☐ Allouer le tableau (comme avec `new`) puis initialiser les éléments
  - ☐ Renvoyer une référence vers le tableau



- ▶ Accès à la taille du tableau avec `tab.length`

- ▶ Accès à un élément avec `tab[indice]` comme en C

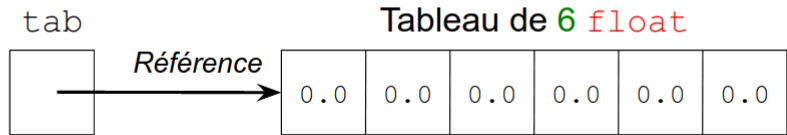
Exemple : `tab[i] = tab[j] * 2;`

**Attention :** Comme en C, les éléments sont indexés à partir de 0

- ▶ Un accès en dehors des bornes du tableau provoque une erreur à l'exécution (`ArrayOutOfBoundsException`)

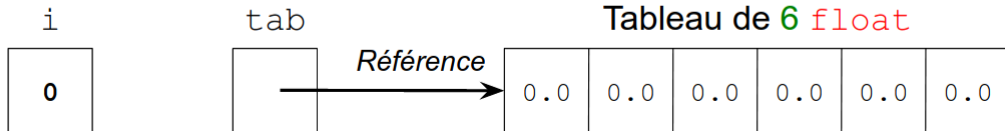
## Exemple d'accès à un tableau (1/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



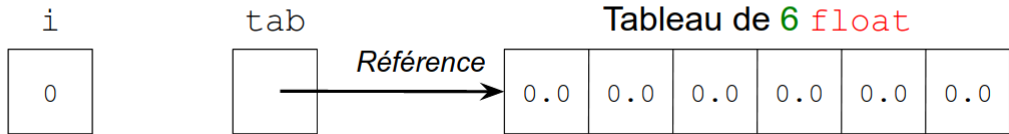
## Exemple d'accès à un tableau (2/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    → for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



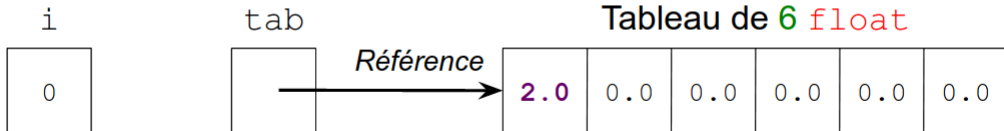
## Exemple d'accès à un tableau (3/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    → for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



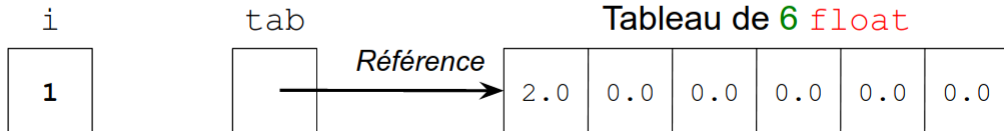
## Exemple d'accès à un tableau (4/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



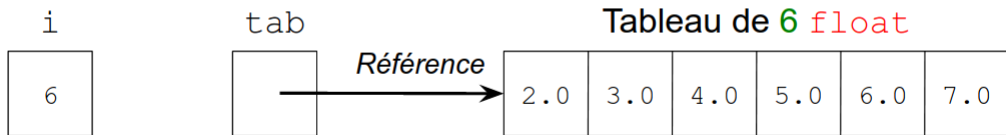
## Exemple d'accès à un tableau (5/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    → for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



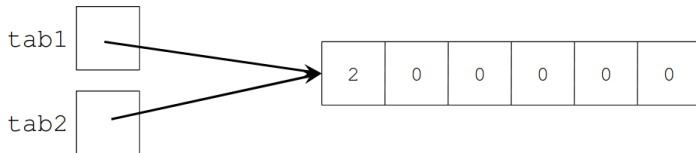
## Exemple d'accès à un tableau : fin de boucle (6/6)

```
public static void main(String[] args) {  
    float[] tab = new float[6];  
    for(int i=0; i<tab.length; i++) {  
        tab[i] = i + 2;  
    }  
}
```



```
public static void main(String[] args) {  
    byte[] tab1 = new byte[6];  
    byte[] tab2 = tab1;  
    tab2[0] = 2;  
    System.out.println("tab1: " + tab1[0]);  
} /* affiche 2
```

- Dans l'exemple précédent, tab1 et tab2 sont deux variables différentes, mais elles référencent le même tableau

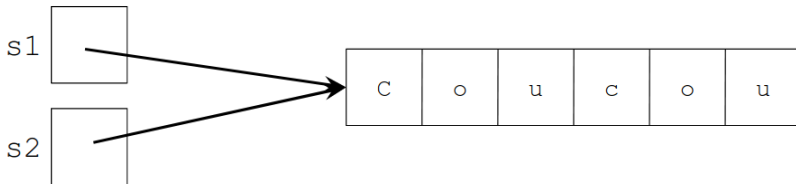




# String et aliasing

- ▶ Comme pour les tableaux, **il n'existe en fait pas de variable de type String en Java !**
- ▶ En revanche, **String** déclare une variable référençant une zone mémoire typée avec le type **String**
- ▶ Exemple :

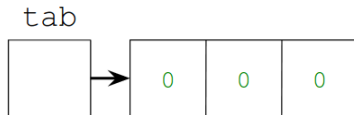
```
String s1 = "Coucou"; String s2 = s1;
```



# Passage par référence vs Passage par valeur

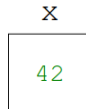
- ▶ Variable de type référence : contient une référence vers une structure de données  
→ **Tableaux** et **String**

```
int[] tab = new int[3];
```



- ▶ Variable de type primitif : contient une valeur  
→ boolean, byte, short, char, int, long, float et double

```
int x = 42;
```



- ▶ Dans `public static void main(String[] args)`,  
args est une **référence vers un tableau de chaînes de caractères** correspondant aux arguments du programme
  - Si aucun argument : `args.length` vaut 0
  - Sinon, `args[0]` est le premier argument, `args[1]` le second etc.

```
class CmdLine {  
    public static void main(String[] args) {  
        for(int i=0; i<args.length; i++) {  
            System.out.println(  
                "args[" + i + "]: " + args[i]);  
        }  
    }  
}
```

- ▶ Allocation d'un tableau avec  
`new type[n]`
- ▶ Déclaration d'une variable référençant un tableau avec  
`type[] var`
- ▶ Accès à un élément avec  
`var[indice]`
- ▶ L'argument du `main` est le tableau des arguments du programme

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- ▶ Structure de données = regroupement de données
  - Permet de lier entre elles des données
  - Simplifie le traitement de ces données
- ▶ Exemple : une structure de données "Personnage" regroupant :
  - une image (l'apparence du personnage),
  - une position,
  - un nombre de points de vie...
- ▶ Exemple : une structure de données "ListePersonnages" regroupant :
  - un ensemble de personnages

- ▶ Le tableau (vu juste avant)
  - ☐ Regroupe un nombre fini d'éléments **homogènes**
  - ☐ Les éléments sont indexés par un **indice**
- ▶ Le tuplet (*aussi parfois appelé enregistrement ou structure*)
  - ☐ Regroupe un nombre fini d'éléments **hétérogènes**
  - ☐ Les éléments sont indexés par un **symbole**
  - ☐ Les éléments s'appellent des **champs** ou **attributs**

En Java

- ▶ Une structure de données (tuple ou tableau) s'appelle un **objet**
- ▶ Un **objet** possède un **type**
- ▶ Le **type d'un objet** s'appelle une **classe**
- ▶ Si la classe d'un objet *o* est *C*, alors on dit que ***o* est une instance de *C***



# Le concept "Classe" : les bases

Un objet est donc une variable qui doit être déclarée avec un type... que l'on appelle **classe**

- ▶ Type complexe (en opposition aux typages primitifs : entiers etc)
- ▶ Regroupe un ensemble de données (de type primitif ou objet !) que l'on appelle **attributs**
  - chaque attribut a un nom, un type, une valeur initiale (*facultatif*)
- ▶ Regroupe un ensemble de **procédures** appelées **méthodes** offrant des fonctionnalités pour manipuler :
  - 1 ses attributs
  - 2 des données extérieures
- ▶ Principe **d'encapsulation** de données
- ▶ **Bilan** : classes représentant les objets du système + fonctions de manipulations + classe exécutable

# Exemple illustratif

```
class Carre {  
  
    /* Encapsulation de 3 entiers */  
    int cote;  
    int origine_x ;  
    int origine_y ;  
  
    /* Une methode permettant de deplacer un objet.  
    * 'this' : acces direct aux donnees encapsulees */  
    void deplace(int x, int y) {  
        this.origine_x = this.origine_x + x ;  
        this.origine_y = this.origine_y + y ;  
    }  
  
    /* Une autre methode definie dans la classe */  
    int surface() {  
        return this.cote * this.cote ;  
    }  
  
}
```

En programmation objet, le concepteur du programme doit déterminer

- ▶ les objets et données appartenant à chaque objet
- ▶ les droits d'accès qu'ont les autres objets à ces données

Encapsulation permet de cacher ou non des données entre objets du programme

Une donnée peut être en accès **public** ou **privé**

On verra dans le module MOOB un descriptif plus détaillé des droits d'accès

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

# Deux étapes pour créer un tuple

► Étape 1 : définition de la classe d'un tuple (i.e., de son type)

- Donne une énumération des champs du tuple
- Utilisation du mot clé `class` suivi d'un identifiant de `type`

```
class Perso {  
    int pointsVie;  
    int x;  
    int y;  
}
```

► Étape 2 : création d'une instance de la classe avec `new`

```
Perso bilbon = new Perso();
```

bilbon référence une **instance** de la classe Perso

## Accès aux champs d'un tuple avec "."

```
class Perso {
    int pointsVie;
    int x;
    int y;
}
```

```
class MonProg {
    public static void main(String[] a) {
        Perso bilbon = new Perso();
        bilbon.pointsVie = 10;
        bilbon.x = 0;
        bilbon.y = 0;
        Perso sauron = new Perso();
        sauron.pointsVie = 1000;
        sauron.x = 666;
        sauron.y = 666;
    }
}
```

# Ne confondez pas classe et instance ! (1/2)

- ▶ Les objets utilisés dans les programmes sont des représentations dynamiques créées à partir du modèle donné par la classe. Chaque **instance** d'une classe :
  - ☐ se conforme à la description que la classe fournit
  - ☐ possède une valeur propre pour chaque attribut (qui caractérisent l'état de l'objet)
  - ☐ peut se voir appliquer toute méthode définie dans la classe
  - ☐ est référencée par une variable
- ▶ Une classe peut évidemment être **instanciée** plusieurs fois !

# Ne confondez pas classe et instance! (2/2)

Une **classe** est une sorte de **moule**

<b>Perso</b>
<pre>pointsVie: int x: int y: int</pre>

Qui permet de créer des **instances** de même type

<b><u>bilbon:Perso</u></b>
<pre>pointsVie:int = 10 x:int = 0 y:int = 0</pre>

<b><u>sauron:Perso</u></b>
<pre>pointsVie:int = 1000 x:int = 666 y:int = 666</pre>



- ▶ Quand on code en Java, on utilise les conventions suivantes :
  - Les noms de classes des tuples commencent par une majuscule
  - Les variables et champs commencent par une minuscule
  - Les méthodes (= fonctions internes aux classes) commencent par une minuscule
    - Visuellement, si on voit un symbole commençant par une majuscule, on sait qu'on parle d'une classe
- ▶ On ne définit qu'une et une seule classe par fichier source
- ▶ Le fichier source définissant la classe X s'appelle X.java

# Tuples versus tableaux : nommage

- ▶ La **classe d'un tuple** possède un nom librement défini  
→ Mot clé **class** suivi du **nom** et de l'énumération des champs

```
class Perso { int pointsVie; int x; int y; }
```



Nom de la classe

- ▶ La **classe** d'un tableau possède un nom imposé  
→ Type des éléments suivi du symbole [ ]

```
int[]
```



Nom de la classe

# Tuples versus tableaux : allocation

- ▶ Un objet est alloué avec le mot clé `new` suivi de la `classe`, suivi de parenthèses dans le cas des tuples, mais pas des tableaux
  - `new` renvoie une référence vers l'objet alloué
- ▶ Par exemple
  - `new Perso()` → alloue une instance de la classe `Perso`
  - `new int[5]` → alloue un tableau de 5 `int`

# Tuples versus tableaux : accès

- ▶ Accès à un champ d'un tuple :  
variable suivie d'un point et du nom du champ

```
sauron.pointsVie = 1000;
```

- ▶ Accès à élément d'un tableau :  
variable suivie d'un indice entre crochets

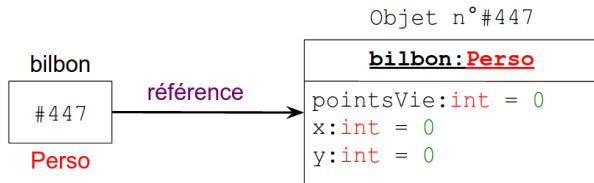
```
tab[3] = 42;
```

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

# Objets et références (1/3)

- ▶ Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- ▶ Une référence contient **l'adresse** d'un objet en mémoire
  - **Comme un pointeur**, contient l'adresse d'une structure
  - **Contrairement aux pointeurs**, la **seule opération autorisée sur les références** est l'affectation d'une référence de même type
- ▶ **Perso** p déclare une **référence** vers un objet de type **Perso**

```
Perso bilbon = new Perso();
```



## Objets et références (2/3)

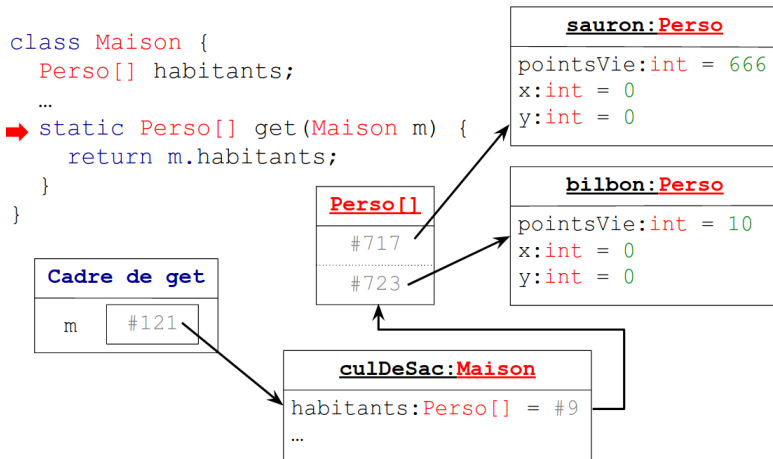
- ▶ Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- ▶ Une référence contient **l'adresse** d'un objet en mémoire
  - **Comme un pointeur**, contient l'adresse d'une structure
  - **Contrairement aux pointeurs**, la **seule opération autorisée sur les références** est l'affectation d'une référence de même type
- ▶ **Perso** p déclare une **référence** vers un objet de type **Perso**
- ▶ De la même façon, **int []** tab déclare une **référence** vers un objet de type **int []**

- ▶ Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- ▶ Une référence contient **l'adresse** d'un objet en mémoire
  - **Comme un pointeur**, contient l'adresse d'une structure
  - **Contrairement aux pointeurs**, la **seule opération autorisée sur les références** est l'affectation d'une référence de même type
- ▶ `Perso p` déclare une **référence** vers un objet de type `Perso`
- ▶ De la même façon, `int[] tab` déclare une **référence** vers un objet de type `int[]`
- ▶ Et `Perso[]` déclare donc une **référence** vers un tableau dans lequel chaque élément est une **référence** vers un `Perso`



# Java ne manipule que des références !

pour les classes et tableaux



Rappel : que veut dire ici static ?

- ▶ **null** : valeur littérale indiquant qu'aucun objet n'est référencé

```
Maison m = new Maison();  
Perso bilbon = new Perso();  
m.proprio = null; /* pas encore de proprietaire */  
  
if(m.proprio == null)  
    m.proprio = bilbon;
```

- ▶ Par défaut, les champs (resp. éléments) de type références d'un tuple (resp. tableau) sont initialisés à **null**

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 **Méthodes et types complexes**
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés



- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 **Méthodes et types complexes**
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- ▶ Membres qui ne dépendent pas d'une instance de la classe
  - attribut statique : partagé, accessible et identique pour toutes les instances de la classe
  - méthode statique : utilisable sans instance de la classe !

→ `System.out` est un attribut de classe de la classe `System`

- ▶ **Attention !**

- Un attribut de classe est une variable contrairement à un champ d'instance : un attribut d'instance est un symbole nommant un élément d'une structure de données

- ▶ Un attribut statique doit être accédé par des méthodes statiques

- ▶ Sauf mention explicite

- On n'utilisera pas d'attributs de classe dans ce cours ni en MOOB
  - On utilisera a priori `static` uniquement pour la méthode `main`

## Petit aparté : initialisation d'un attribut de classe

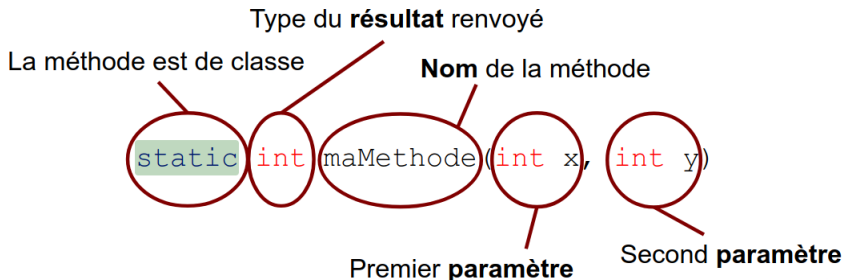
S'il y a besoin d'une initialisation autre qu'une valeur, c'est impossible de le faire dans un constructeur (indépendance vis à vis des objets)

→ bloc spécifique qui sera exécuté lors de la création de la classe

```
public class C {
    static int T[];
    static {
        T = new int[10];
        for (int i=0, i<10; i++)
            T[i]=i;
    }
}
```

# 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)
- ▶ RAPPEL : pas de **this** à l'intérieur d'une méthode statique ! Seulement accès aux attributs **static**





## 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

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

**!! ATTENTION !!**

Les méthodes sont déclarées dans une classe, pas dans d'autres méthodes !

# Invocation d'une méthode de classe

## ► Utilisation

- Invocation avec `nomMethode(arg1, arg2...)`
- 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 a = 2;  
    int res = add(1, a);  
    System.out.println("1 + 2 = " + res);  
}
```

# Plan du cours

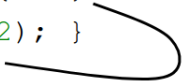
- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 **Méthodes et types complexes**
  - Les méthodes de classe
    - La surcharge de méthode
      - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

# 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 types de paramètres**
  - La méthode appelée est sélectionnée en fonction du type des arguments
  - Si une méthode ne retourne rien, un **return** est implicitement ajouté à la fin du corps de la méthode

```
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



# Rappel : 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

## Rappel : Variables locales (2/2)

```
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);  
    }  
}
```

Similaire aux variables  
locales en C

Cadres d'appel pour les  
fonctions

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 **Méthodes et types complexes**
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés



- ▶ 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**, **short**, **int**, **long**, **float**, **double**, **char**)
  - Passage par référence pour les autres types  
(**String** et tableaux)

## Passage par valeur : types primitifs

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

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

## Traçons ensemble ce qu'il se passe en mémoire

Bilan : pas de pointeurs vers les types primitifs comme en C!

- ▶ Le fait que les variables `y` dans `f` et `g` aient le même nom n'a aucune influence sur l'exécution
- ▶ Si les variables de `g` se nommaient `a` et `g`, le programme fonctionnerait exactement de la même façon !
- ▶ `g` modifie la copie de la variable `x` de `f`, pas celle de `f`

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

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

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

Cette fois, c'est par référence !

Rappel :

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

- ▶ 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

- ▶ 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]);  
}
```

Qu'est ce que f() affiche?

```
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 !

Pourquoi ? Quelqu'un vient expliquer au tableau ?

- ▶ Déclaration d'une méthode de classe

```
static type nom(type param1, ...) { 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



- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 **Méthodes et types complexes**
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- ▶ Prog. impérative = celle que vous utilisez jusqu'à maintenant
- ▶ Un programme C est constitué de
  - Structures de données
  - Et de fonctions ( $\sim$  méthodes) qui manipulent ces structures
- ▶ Pour réutiliser une structure de données en C dans un autre projet
  - Il faut trouver la structure de données et la copier
  - Mais il faut aussi **trouver les méthodes qui manipulent la structure de données et les copier**
    - nécessite une structure claire du code

- ▶ Faire preuve de discipline quand on programme
  - Regrouper les procédures qui manipulent une structure de données dans la classe qui définit la structure de données (ie jusqu'à maintenant des méthodes statiques)
  - Éviter de manipuler directement une structure de données à partir de l'extérieur de la classe
- ▶ Solution partiellement satisfaisante car contraintes sur la manipulation des attributs
- ▶ Méthode d'instance : aide à faire preuve de discipline

- ▶ **But** : simplifier la définition des méthodes qui manipulent une structure de données
- ▶ Principe : associer des méthodes aux instances
  - Méthode d'instance = méthode sans mot clé `static`
  - Méthode qui manipule une structure de données
  - Associée (=encapsulée) dans la classe dans laquelle la méthode est définie
  - Reçoit un paramètre caché nommé `this` du type de l'instance
    - pas besoin de spécifier explicitement ce paramètre, simplifie le code

# Méthode d'instance vs méthode de classe

## Avec méthode d'instance

```
class Monster {  
    int health;  
    void kill() {  
        this.health = 0;  
    }  
}
```

- ▶ paramètre **Monster this** implicitement ajouté
- ▶ **this** est le **receveur** de l'appel

## Avec méthode de classe

```
class Monster {  
    int health;  
    static void kill(Monster m) {  
        m.health = 0;  
    }  
}
```

- ▶ Le receveur d'un appel de méthode d'instance se met à gauche

```
Monster aMonster = Monster.create(aPicture);  
aMonster.kill();
```

- ☐ aMonster = receveur
- ☐ kill = Méthode d'instance appelée

- ▶ Un peu comme si on invoquait

```
Monster.kill(aMonster);  
  
(i.e., this reçoit la valeur aMonster)
```

- ▶ On appelle le kill de **Monster** car la classe du receveur (aMonster) est **Monster**

```
Monster amonster;
```

# Que se passe-t-il si le receveur est null ?

- ▶ Si aMonster vaut `null`

```
Monster aMonster = null;  
aMonster.kill() ;
```

→ erreur de type `NullPointerException`

- ▶ En l'absence d'ambiguïté, **this** peut être omis
  - `health` champ de **Monster** → remplacé par `this.health` à la compilation
  - Les deux codes suivants sont équivalents

```
class Monster {
    int health;
    void kill() {
        this.health = 0;
    }
}
```

```
class Monster {
    int health;
    void kill() {
        health = 0;
    }
}
```

- **ATTENTION** : N'oubliez pas qu'il y a un receveur `this` caché pour les méthodes d'instance !



► En cas d'ambiguïté, utilisez `this` !

- Java utilise la portée lexicale : le compilateur cherche le symbole le plus proche en suivant les blocs de code

```
class Monster {  
    int health;  
    int x;  
    int y;  
    void move(int x, int y) { this.x = x; this.y = y; }  
}
```

Dans ce cas, `this` est nécessaire pour lever l'ambiguïté entre le champ `x` et l'argument `x`

- ▶ Avec **static**, la méthode s'appelle une **méthode de classe**

- ❑ Marquée par le mot clé `static`
- ❑ Uniquement des paramètres explicites

```
static void kill(Monster monster)
```

- ▶ Sans `static`, la méthode s'appelle une **méthode d'instance**

- ❑ Reçoit un paramètre caché nommé **this** du type de la classe
- ❑ Invocation avec receveur à gauche : `monster.kill()`;

```
void kill()
```

 $\longleftrightarrow$ 

```
static void kill(Monster this)
```

# Invocation inter-classe de méthodes (1/3)

► Le mot clé `class` a deux rôles différents en Java

- Comme espace pour définir des classes définissant des tuples
- Comme espace pour définir des méthodes de classe (avec le mot clé `static`) ou non

```
class Perso {  
    int pointsVie;  
    int x;  
    int y;  
}
```

```
class MonProg {  
    static void maFonction(int x) {  
        ...  
    }  
}
```

► On peut bien sûr combiner les deux rôles

```
class Perso { int pv; static void maFonc() { ... } }
```

## Invocation inter-classe de méthode (2/3)

- ▶ Par défaut, Java résout un appel de méthode dans la classe
  - Pour appeler une méthode de classe d'une autre classe :  
préfixer le nom de la méthode avec la classe suivi d'un point

```
class MonProg {  
    static void maFonction(int x) {  
        Perso bilbon = new Perso();  
        Perso.display(bilbon);  
    }  
}
```

```
class Perso {  
    int pointsVie;  
  
    static void display(Perso p) {  
        ...  
    }  
}
```

# Invocation inter-classe de méthode (3/3)

► Pour les méthodes d'instance

- préfixer le nom de la méthode avec l'instance appelant suivi d'un point

```
class Perso {  
    int pv;  
    void maFonc() { ... }  
}
```

```
class Test{  
    public static void main (String[] args)  
    {  
        Perso p = new Perso();  
        p.maFonc();  
    }  
}
```

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

## Améliorer la réutilisabilité du code

Quand on réutilise du code, on est en général intéressé par une fonctionnalité, pas par une mise en oeuvre spécifique

- ▶ Exemple : une classe *École* contenant un ensemble d'élèves
  - ☐ Objet sur lequel je peux ajouter des élèves
  - ☐ Mais savoir que les élèves sont stockés dans un tableau extensible ou une liste chaînée n'est pas essentiel (sauf niveau performance)

# Objectif de la programmation objet (2/2)

Concevoir une application en terme d'objets qui interagissent

Au lieu de la concevoir en terme de structures de données et de méthodes (programmation impérative)

Objet = entité du programme fournissant des fonctionnalités

- ▶ Encapsule une structure de données et des méthodes qui manipulent cette structure de données
- ▶ Expose des fonctionnalités



- ▶ L'objet en Java contient une mise en œuvre
  - ☐ des champs (déjà vus)
  - ☐ des méthodes d'instance (déjà vues)
  - ☐ des **constructeurs** (méthode d'initialisation que l'on va voir)
- ▶ Expose des fonctionnalités
  - ☐ En empêchant l'accès à certains champs/méthodes/constructeurs à partir de l'extérieur de la classe
  - ☐ Principe d'**encapsulation** (on va revenir dessus dans la suite)

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- Pour le moment, on écrit une méthode de classe qui

- ☐ alloue un objet
- ☐ initialise l'objet
- ☐ renvoie l'objet ainsi initialisé

- Exemple :

```
static Perso create (int pv) {  
    Perso p = new Perso();  
    p.pointsVie = pv;  
    p.x = 0;  
    p.y = 0;  
    return res;  
}
```

# Le constructeur : méthode d'instance spéciale

- ▶ Constructeur = méthode simplifiant la création d'un objet
  - ☐ **Méthode d'instance** possédant le nom de la classe
  - ☐ **Pas de type de retour !!**
  - ☐ Peut posséder des paramètres
  
- ▶ Le constructeur est appelé **automatiquement** avec un **new**
  - ☐ **new** commence par allouer un objet
  - ☐ Puis appelle le constructeur avec comme receveur le nouvel objet
  - ☐ Et, enfin, renvoie l'objet

# Un exemple avec et sans constructeur

## Avec constructeur

```
class Perso {  
    int pointsVie;  
    int x,y;  
  
    /* def. constructeur  
       sans type de  
       retour */  
    Perso(int pv) {  
        this.pointsVie = pv;  
        this.x = 0;  
        this.y = 0;  
    }  
}
```

---

```
c = new Perso("Bilbo");
```

## Sans constructeur

```
class Perso {  
    int pointsVie;  
    int x,y;  
  
    static Perso create (int pv) {  
        // alloc + constructeur vide  
        Perso p = new Perso();  
        p.pointsVie = pv;  
        p.x = p.y = 0;  
        return p;  
    }  
}
```

---

```
c = Perso.create("Bilbo");
```

# Et si pas de constructeur défini ?

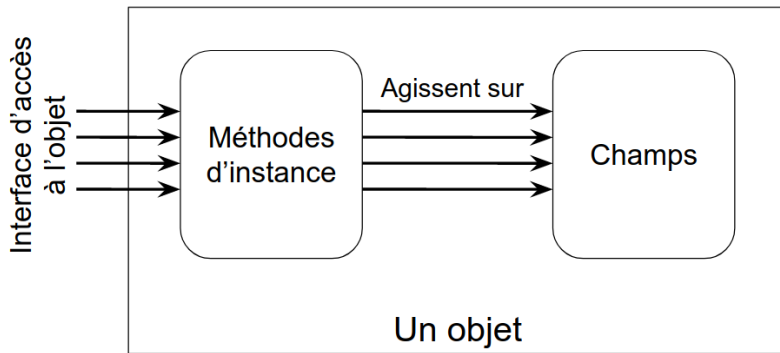
- ▶ Si pas de constructeur, Java génère un constructeur vide sans paramètre qui initialise les champs à 0, +0.0, False, null
- ▶ Bonne manière : définir un constructeur vide et un constructeur "complet"
- ▶ C'était le cas dans ce que l'on a vu dans le cours 3, slide 39

```
Maison m = new Maison();  
Perso bilbon = new Perso();  
m.proprio = null; /* pas encore de proprietaire */  
  
if(m.proprio == null)  
    m.proprio = bilbon;
```

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

# L'encapsulation, un peu plus loin

- ▶ Principe : cacher les détails de mise en œuvre d'un objet
  - pas d'accès direct aux champs de l'extérieur de l'objet





- ▶ Chaque entité (classe, champ, méthode ou constructeur) possède un niveau d'encapsulation
  - Définit à partir d'où dans le programme une entité est visible
- ▶ Permet de masquer les détails de mise en œuvre d'un objet

- ▶ Trois niveaux de visibilité pour une entité en Java (on en verra un 4ème avec l'héritage)
  - ☐ Invisible en dehors de la classe : mot clé **private**
  - ☐ Invisible en dehors du package : comportement par défaut
  - ☐ Visible de n'importe où : mot clé **public**
  
- ▶ Permet de masquer les détails de mise en œuvre d'un objet
  - ☐ Les champs sont **privés** (inaccessibles en dehors de la classe)
  - ☐ Les méthodes sont **publiques**

## L'encapsulation : un exemple

```
package ensiie.perso;

public class Perso { /* visible partout */
    private String name; /* invisible hors
                        du fichier (de Perso) */

    public Perso() { ... } /* visible partout */

    public void setname (String name) { ... } /* visible
                                                partout */

    void display() { ... } /* invisible en dehors
                          du package ensiie.perso */
}
```

- ▶ Attributs **private**
- ▶ Pour chaque attribut, définir des méthodes **d'accès** qui seront **public** : les *accesseurs*
  - Ex. en lecture : **getX**, **getY**, **getPointsVie** pour la classe **Perso**
  - Ex. en écriture : **setX** et **setY**
- ▶ Limiter au strict nécessaire les autres méthodes publiques !

Les principes d'héritage entre les classes complexifient encore ce processus (cf le prochain cours avec la visibilité **protected**)

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

- ▶ Un grand nombre de classes fournie par Java SE
  - Implémentent des structures de données et traitements génériques
  - Forment l'API (*Application Programmer Interface*) du langage
  - <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

Package	Description
java.awt	Interfaces et classes graphiques
java.io	Entrées/sorties
java.lang	Classes de base (importé par défaut)
java.util	Classes "boîte à outils"

Table – Quelques packages couramment utilisés

- ▶ Importer la classe ou le package dans lequel elle est définie

- ☐ une seule classe

```
import java.util.Date ;
```

- ☐ toutes les classes (même celles inutilisées)

```
import java.util.* ;
```

```
import java.util.Date ;

public class AfficherDate {
    public static void main(String[] args) {
        Date today = new Date();
        System.out.println("Nous sommes le " +
            today.toString());
    }
}
```



- ▶ Préciser avant la définition de votre classe le package auquel elle appartient
  - Exemple :

```
package ensiie.ipoo ;

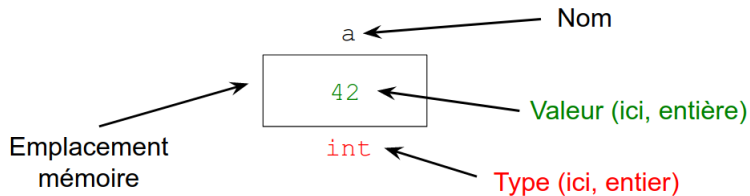
import java.util.Date ;

public class AfficherDate {
    ...
}
```

- ▶ **Attention** : le chemin d'accès au fichier AfficherDate.java doit correspondre au nom de son package.
  - Celui-ci doit donc être dans ensiie/ipoo/AfficherDate.java
  - **ET** être accessible à partir des chemins d'accès à la compilation (cf Cours 2)

- 1 Les tableaux en Java
- 2 Les structures de données : objets et classes
- 3 Manipulation de tuples en Java
- 4 Objets et références, deuxième couche
- 5 Méthodes et types complexes
  - Les méthodes de classe
    - La surcharge de méthode
    - Passage de paramètres par valeur ou référence ?
  - Méthode d'instance
- 6 Un peu plus loin dans la programmation objet
  - Le constructeur : une méthode pour la création d'objets
  - Retour sur l'encapsulation
- 7 Les *packages*
- 8 Notions clés

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



- ▶ Déclaration d'une classe définissant un tuple avec

```
class Nom { typeun champsun; typedeux champsdeux; ... }
```

- ▶ Allocation d'un objet avec l'opérateur `new`  
`new Nom()` si tuple ou `new type[n]` si tableau

- ▶ **En Java, il n'existe que des types références, pas de type objet**

- ▶ **Lors d'un appel de méthode, un objet est passé par référence**

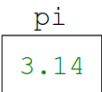
# Notion : valeur de type primitif versus référence

Une variable contient

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

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

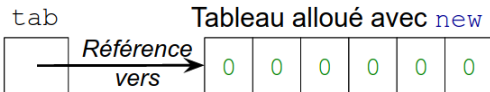
```
double pi = 3.14;
```



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 table of 6 cells, each containing the value '0'. Above the table is the text 'Tableau alloué avec new'. Below the box is the text 'int[]'.

- ① Tableaux
- ② Structures de données
- ③ Méthodes de classe
- ④ Méthodes d'instance