Introduction à la Programmation Orientée Objet

4 - Les différents types de méthodes

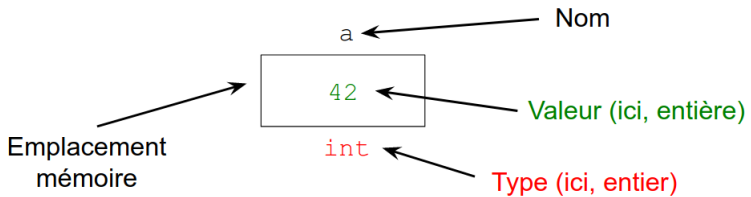
Valentin Honoré

valentin.honore@ensiie.fr

FISA 1A

Rappel : la variable

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

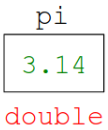


Rappel : valeur de type primitif versus référence

Une variable contient

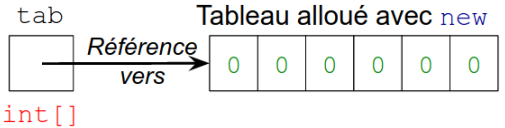
- Soit une **valeur de type dit primitif**
  (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`)

```
double pi = 3.14;
```



- Soit une **valeur de type dit référence** (identifiant unique de tableau ou de `String`)

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



Qu'est-ce qu'une méthode ?

- Une méthode est un regroupement d'instructions
  - Création d'une macro-instruction
    $\rightarrow$ permet de réutiliser du code
  - Peut prendre des arguments et renvoyer un résultat

- Dans un premier temps, on étudie les **méthodes de classe**
  - Il existe aussi les méthodes d'instance, la différence sera expliquée plus loin

Plan du cours

Les membres de classe : static

- Membres qui ne dépendent pas d'une instance de la classe
  - attribut : partagé, accessible et identique pour toutes les instances de la classe
  - méthode : 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'utilise pas d'attributs de classe dans ce cours
  - On utilise static **uniquement pour la méthode `main`**

Petit aparté : initialisation d'un attribut de classe

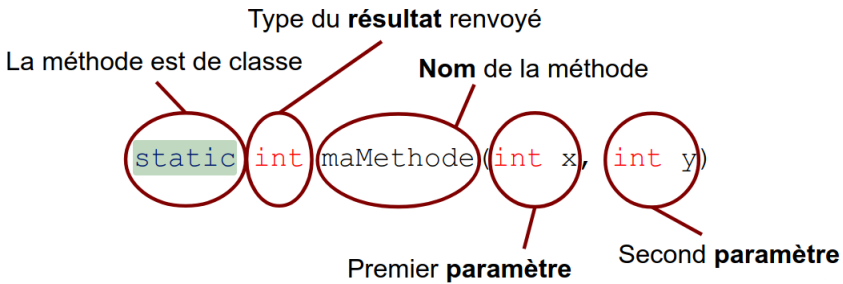
S'il y a besoin d'une initialisation autre qu'une valeur, c'est impossible de le faire dans un constructeur (indépendance vis à vis des objets)
$\rightarrow$ bloc spécifique qui sera exécuté lors de la création de la classe

```
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)

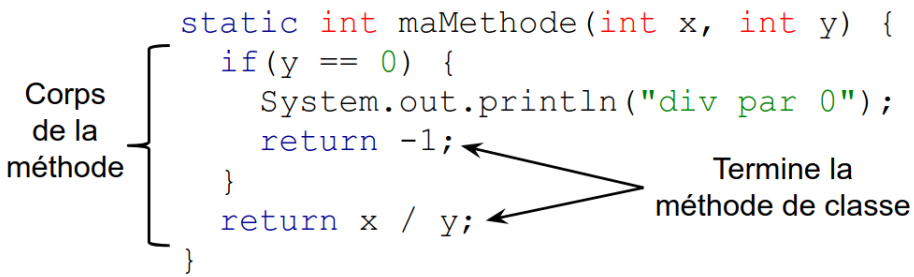
- Une méthode de classe possède
  - Un **nom** ($\leftrightarrow$ nom de la macro-instruction)
  - Une liste de **paramètres** d'entrées sous la forme `type symbol`
  - Le type de **résultat** renvoyé (`void` si pas de résultat)

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

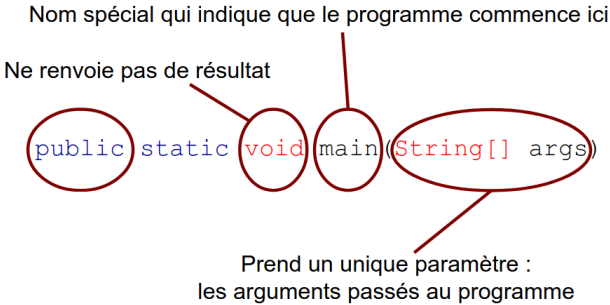


Définition d'une méthode de classe (2/2)

- 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)



La méthode main



!! **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;
  }

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

Exécute `add` puis remplace par le résultat (ici 3)

Plan du cours

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

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 (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

Plan du cours

Passage par valeur et par référence

- Le passage des arguments peut se faire par
  - Valeur : l'appelé reçoit une copie d'une valeur
    $\rightarrow$ la copie et l'originale **sont différentes**
  - Référence : l'appelée reçoit une référence vers une valeur
    $\rightarrow$ 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 de `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

Passage par référence

```
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

En résumé, pour les passages de paramètres des méthodes

- **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**

Petit piège (1/2)

- 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 ?

Petit piège (2/2)

```
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 ?

Notions clés sur les méthodes de classe

- 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

Plan du cours

Limites de la programmation impérative

- Prog. impérative = celle que vous utilisez jusqu'à maintenant
- Un programme est constitué de
  - Structures de données
  - Et de méthodes qui manipulent ces structures
- Pour réutiliser une structure de données 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**

    $\rightarrow$ nécessite une structure claire du code

Ce que l'on fait pour l'instant

- Faire preuve de discipline quand on programme
  - Regrouper les méthodes qui manipulent une structure de données dans la classe qui définit la structure de données
  - Éviter de manipuler directement une structure de données à partir de l'extérieur de la classe
- Solution partiellement satisfaisante car aucune aide fournie par le langage
- Méthode d'instance : aide à faire preuve de discipline

La méthode d'instance

- **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 à 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;
  }
}
```

Invocation d'une méthode d'instance

- 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`

Omission de this

- 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 !

Omission de this

- 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

Notions clés sur les méthodes

- 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()
```

$\leftrightarrow$

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

Plan du cours

Objectifs de la programmation objet (1/2)

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 : des champs et des méthodes

- 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)

Plan du cours

Création d'un objet

- 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;

  /* déf. 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

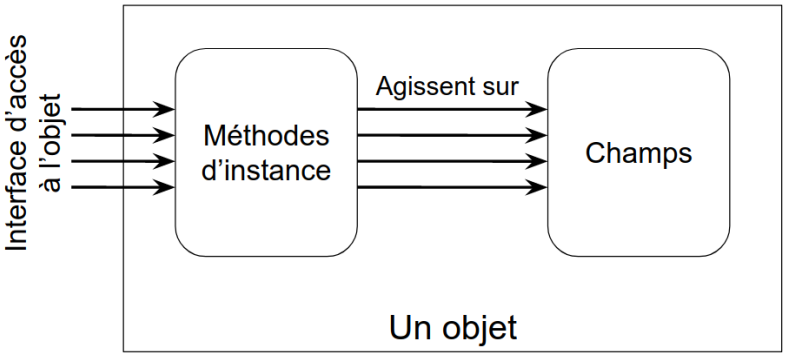
```
Maison m = new Maison();
Perso bilbon = new Perso();
m.proprio = null; /* pas encore de proprietaire */

if(m.proprio == null)
    m.proprio = bilbon;
```

Plan du cours

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



Application aux objets

- 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

Mise en œuvre en Java

- 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 */
```

L'encapsulation : bonnes pratiques

- 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)

Notions clés

- Programmation orientée objet
  - Conception d'un programme à partir d'objets qui interagissent
  - On s'intéresse à la fonctionnalité d'un objet avant de s'intéresser à sa mise en œuvre

- Le constructeur
  - Méthode appelée pendant new pour initialiser un objet

- Principe d'encapsulation pour cacher la mise en œuvre
  - private : invisible en dehors de la classe
  - Par défaut : invisible en dehors du package
  - public : visible partout