TELECOM SudParis

IP PARIS

Programmation orientée objet

CSC 3101

Algorithmique et langage de programmation

Gaël Thomas

Petits rappels : l'objet

- Une structure de données (tuple ou tableau) s'appelle un **objet**
- Un **objet** possède un **type** appelé **sa classe**
  - Si la classe de l'objet o est C, on dit que **o est une instance de C**
- En Java, on ne manipule que des **références** vers des objets
- Une **méthode d'instance** est une méthode associée à l'objet
  - Possède un paramètre implicite du type de la classe nommé `this`

But de la programmation orientée objet

Améliorer la réutilisabilité du code

car une ligne de code coûte très cher !

(~1h de développement par ligne de code)

Que signifie réutiliser du code ?

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

- L'exemple de la classe `Army` dans l'armée de monstres
  - Objet sur lequel je peux appeler `addMonster`
  - Mais savoir que les monstres sont stockés dans un tableau extensible ou une liste chaînée n'est pas essentiel (sauf pour les performances)

Programmation orientée objet

- 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)

⇒ On ne s'intéresse plus à la mise en œuvre d'un objet, mais d'abord aux fonctionnalités qu'il fournit

- **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
  - Des méthodes d'instances
  - Des **constructeurs** (méthodes d'initialisation vues dans ce cours)

- Expose des fonctionnalités
  - En empêchant l'accès à certains champs/méthodes/constructeurs à partir de l'extérieur de la classe

  ⇒ principe **d'encapsulation** vue dans ce cours

Plan du cours

1. Le constructeur
2. L'encapsulation
3. Les champs de classe

Création d'un objet

- Jusqu'à maintenant, pour créer un objet, on écrit une méthode de classe qui
  - Alloue l'objet
  - Initialise l'objet
  - Renvoie l'objet initialisé

- Par exemple

```
static Compte create(String name) {
  Compte res = new Compte();
  res.name = name;
  res.solde = 0;
  return res;
}
```

Le constructeur

- Constructeur = méthode simplifiant la création d'un objet

- Méthode d'instance spéciale pour initialiser 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 pendant un `new`
  - `new` commence par allouer un objet
  - Puis appelle le constructeur avec comme receveur le nouvel objet
  - Et, enfin, renvoie l'objet

Exemple avec la classe Compte

Avec constructeur

```
/* initialisation */
class Compte { …
  Compte(String name) {
    this.name = name;
    this.solde = 0;
  }
}
```

```
/* création/init. */
new Compte("Tyrion");
```

Sans constructeur

```
/* création/initialisation */
class Compte { …
  static Compte create(String
                              name) {
    Compte res = new Compte();
    res.name = name;
    res.solde = 0;
    return res;
  }
}
```

```
/* création/initialisation */
Compte.create("Tyrion");
```

Exemple avec la classe `Compte`

Avec constructeur

```
/* initialisation */
class Compte { …
  Compte(String name) {
    this.name = name;
    this.solde = 0;
  }
}
```

Sans constructeur

```
/* création/initialisation */
class Compte { …
  static Compte create(String
                       name) {
    Compte res = new Compte();
    res.name = name;
```

```
/*
new
```

Remarque :

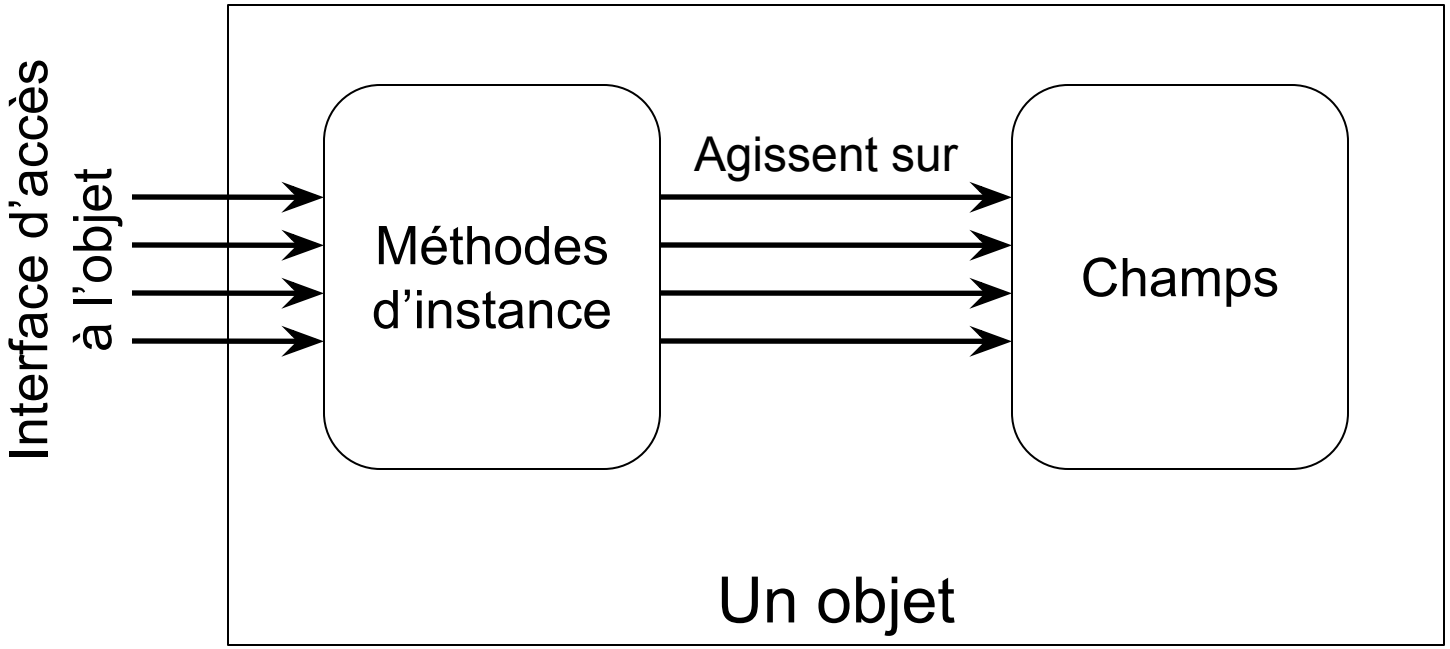
Si pas de constructeur, Java génère un constructeur sans paramètre qui initialise les champs à `0`/`null`

Plan du cours

1. Le constructeur
2. L'encapsulation
3. Les champs de classe

L'encapsulation

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



Mise en œuvre : la visibilité

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

La visibilité en Java

- Trois niveaux de visibilités pour une entité en Java

  (techniquement quatre, mais le dernier est vu en CI6)

  - 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`

- En général
  - Les champs sont privés (inaccessibles en dehors de la classe)
  - Les méthodes sont publiques

La visibilité par l'exemple

```
package tsp.bank;

public class Bank {  /* visible partout */
  private Node node; /* invisible de l'extérieur
                        du fichier (de Bank) */

  public Bank() { … } /* visible partout */
  public void lookup() (String name) { … } /* visible
                                              partout */

  void display(Ba) { … } /* invisible en dehors
                            du package tsp.bank */
}
```

Plan du cours

1. Le constructeur
2. La visibilité
3. Les champs de classe

Les champs de classe

- Champ de classe
  - Champ précédé du mot clé `static`
  - Définit **une variable globale**, indépendante d'une instance

  Exemple : `System.out` est un champ de classe de la classe `System`

- Attention !
  - Un champ de classe est une variable mais pas un champ d'instance : un champs d'instance est un symbole nommant un élément d'une structure de donnée

- Dans la suite du cours
  - On n'utilise pas de champs de classe
  - **On utilise `static` uniquement pour la méthode `main`**

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

- Visibilité pour cacher les détails de mise en œuvre d'un objet
  - `private` : invisible en dehors de la classe
  - Par défaut : invisible en dehors du package
  - `public` : visible de partout