

Stage de Programmation Objet

LP DAWIN



Informatique



Partie 2



Spécificités, interfaces et
packages

Partie 2 : Spécificités, interfaces et packages

Méthodes et attributs de classe OU d'instance

On distingue les méthodes :

- **d'instance** (= non statiques) : elles sont appliquées sur l'**instance** courante. On peut alors désigner cette instance avec le mot-clé `"this"`.

Méthodes et attributs de classe OU d'instance

On distingue les méthodes :

- **d'instance** (= non statiques) : elles sont appliquées sur l'**instance** courante. On peut alors désigner cette instance avec le mot-clé "this".
- **de classe** (= statiques) : elles ne sont pas liées à une instance, mais à la **classe**. Ainsi, `this` est interdit, ainsi que l'accès aux attributs non statiques.

Méthodes et attributs de classe OU d'instance

On distingue les méthodes :

- **d'instance** (= non statiques) : elles sont appliquées sur l'**instance** courante. On peut alors désigner cette instance avec le mot-clé "this".
- **de classe** (= statiques) : elles ne sont pas liées à une instance, mais à la **classe**. Ainsi, `this` est interdit, ainsi que l'accès aux attributs non statiques.

Méthodes et attributs de classe OU d'instance

On distingue les méthodes :

- **d'instance** (= non statiques) : elles sont appliquées sur l'**instance** courante. On peut alors désigner cette instance avec le mot-clé "this".
- **de classe** (= statiques) : elles ne sont pas liées à une instance, mais à la **classe**. Ainsi, `this` est interdit, ainsi que l'accès aux attributs non statiques.

C'est exactement la même chose pour les attributs.

On désigne les méthodes/attributs de classe avec le mot-clé "**static**".

Exemples...

Exemple Salle : attributs

```
class Salle {  
    // attributs d'instance :  
  
    int capacite;  
    boolean salleMachine;  
    boolean libreService;  
    int numero;  
  
    // attributs de classe :  
  
    static      int    nbSalles      = 0;  
    static final char TYPE_SEANCE_COURS = 'C';  
    static final char TYPE_SEANCE_TD   = 'D';  
    static final char TYPE_SEANCE_TP   = 'P';  
  
    ...  
}
```

→ capacite est bien lié à une salle, mais pas nbSalles.

Exemple Salle : constructeur

```
...  
  
// constructeur :  
  
Salle(int uneCapacite, boolean estUneSalleMachine,  
      boolean estEnLibreService, int unNumero) {  
    capacite      = uneCapacite;  
    salleMachine  = estUneSalleMachine;  
    libreService  = estEnLibreService;  
    numero        = unNumero;  
    nbSalles++; // attribut statique  
}  
  
...
```

Exemple Salle : méthode d'instance

```
...  
  
// methode d'instance :  
  
/**  
 * S'agit-il d'une salle machine au 3eme etage ?  
 * @return vrai s'il s'agit d'une salle machine au 3eme etage  
 */  
boolean salleMachineAu3eme() {  
    return this.salleMachine    // this est facultatif ici  
        && (this.numero / 100 == 3);  
}  
  
...
```

→ le fait qu'une salle soit une salle machine au 3ème est bien lié à chaque salle particulière (instance).

Exemple Salle : méthode de classe

```
...

// methode de classe :

/**
 * Capacite requise pour une salle , en fonction du type de seance.
 * @param typeCours un caractere designant le type de seance
 * @return la capacite minimale pour ce type de seance
 */
static int capaciteMin(char typeSeance) {
    int capacite = 0;
    switch(typeSeance) {
        case TYPE_SEANCE_COURS:
            capacite = 120;
            break;
        case TYPE_SEANCE_TD:
            capacite = 30;
            break;
        case TYPE_SEANCE_TP:
            capacite = 15;
            break;
        default:
            System.out.println("Type de seance incorrect : " + typeSeance);
            break;
    }
    return capacite;
}
...
}
```

Partie 2 : Spécificités, interfaces et packages

Objets immuables

En Java, il existe des objets **immuables** :

*An object is considered **immutable** if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications.*

Java documentation, Oracle

state = état = valeurs des attributs de l'instance

Objets immuables

Mutables :

- par défaut, les objets que l'on définit

Immuables :

- String
- toutes les classes associées aux types primitifs :
 - Integer pour int
 - Character pour char
 - etc
- on peut en définir soi-même, voir la doc (faisable mais pas simple)
 - en particulier attributs constants (`final` et types immuables)

Conséquence importante :

Un objet immuable passé en paramètre est passé en entrée, pas en entrée/sortie. . . contrairement à un objet mutable.

final sur variables et paramètres

Pour protéger et optimiser le code, on peut aussi penser à indiquer `final` :

- les variables qui ne bougeront pas
- les paramètres qui ne bougeront pas

Exemple :

```
int sommeCapacites(final Salle salle1, final Salle salle2) {  
    salle1 = ...; // interdit par final, mais :  
    salle1.setNom(...); // autorise par final  
    ...  
    final int somme = salle1.getCapacite();  
    somme = somme + salle2.getCapacite(); // interdit par final  
    ...  
    final int somme2 = salle1.getCapacite() + salle2.getCapacite(); // ok  
    ...  
}
```

Partie 2 : Spécificités, interfaces et packages

Énumérations

En Java, une **énumération** est un type (similaire à une classe), ne pouvant prendre que certaines valeurs possibles, listées par le développeur.

Direction.java

```
package mygame;  
  
enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
}
```

Énumérations

utilisation :

```
char input = ...
Direction d;
switch (input) {
    case 'N':
        d = Direction.NORTH;
        break;
    case 'S':
        d = Direction.SOUTH;
        break;
    case 'E':
        d = Direction.EAST;
        break;
    case 'W':
        d = Direction.WEST;
        break;
    default: // should never happen
        System.out.println("Forgotten direction:" + input);
        break;
}
```

Énumérations

Mais on peut faire plus → attributs, constructeurs et méthodes!!!
Bref ce sont comme de vraies classes, dont *les instances sont fixées*.

```
enum Direction {  
    NORTH(0,1), SOUTH(0,-1), EAST(1,0), WEST(-1,0);  
  
    final int moveX;  
    final int moveY;  
  
    Direction(int mvX, int mvY) {  
        moveX = mvX;  
        moveY = mvY;  
    }  
  
    Direction opposite() {  
        switch (input) {  
            case 'N': return Direction.SOUTH;  
            case 'S': return Direction.NORTH;  
            case 'E': return Direction.WEST;  
            case 'W': return Direction.EAST;  
            default: // should never happen  
                System.out.println("Forgotten direction:" + input);  
                break;  
        }  
    }  
}
```

```
Direction d = ...
```

Partie 2 : Spécificités, interfaces et packages

Mini-projet : v2 → v3

Niveau 1

- ➊ passer en **static** les méthodes et champs de classe
- ➋ passer en **final** les paramètres / attributs / variables constants
- ➌ en particulier rendre **Action immuable**
- ➍ définir un **enum ActionType** (quitter ou poser)

Niveau 2

- ➊ action 's' : sauvegarder (*save*) le plateau dans un fichier
- ➋ action 'l' : charger (*load*) depuis un fichier

Partie 2 : Spécificités, interfaces et packages

Interfaces

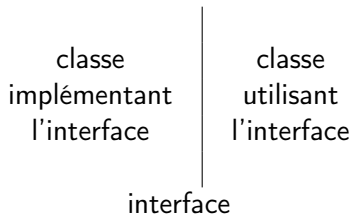
Une **interface** est une liste de signatures de méthodes (sans code).

ISalle.java

```
interface ISalle {  
  
    /**  
     * S'agit-il d'une salle machine au 3eme etage ?  
     * @return vrai s'il s'agit d'une salle machine au 3eme etage  
     */  
    boolean salleMachineAu3eme();  
  
    /**  
     * Affiche le plan de la salle.  
     */  
    void afficherPlan();  
}
```

- pas d'attribut
- pas de constructeur

Interfaces : pour quoi faire ?



L'intérêt :

- rendre l'utilisation indépendante de l'implémentation

Partie 2 : Spécificités, interfaces et packages

Interface : implémenter

Une classe peut implémenter une interface. Elle doit alors :

- signaler qu'elle l'implémente via le mot-clé `implements`
- contenir une implémentation de chacune des méthodes de l'interface. On signale ces méthodes avec l'annotation `@Override`.

Salle.java

```
class Salle implements ISalle {  
  
    @Override  
    boolean salleMachineAu3eme() {  
        return this.salleMachine && this.numero/100==3;  
    }  
  
    @Override  
    void afficherPlan() {  
        System.out.println (...);  
    }  
}
```

Partie 2 : Spécificités, interfaces et packages

Classe utilisant une interface

Les interfaces sont des types (comme les classes).

```
class Batiment {  
    ISalle [] salles;  
  
    ...  
  
    void afficherPlans() {  
        ...  
        salles[i].afficherPlan(); // independant de Salle  
        ...  
    }  
  
    void ajouterSalle() {  
        ...  
        // ici on choisit l'implementation (pas independant)  
        ISalle s = new Salle(...);  
        salles[i] = s;  
    }  
}
```

- en utilisant une interface (via son type), on devient indépendant de l'implémentation
- à la création d'un objet, on doit choisir une implémentation (logique) → seul point à modifier pour changer d'implémentation

Interfaces : exemple

Nous verrons les collections dans la Partie 4.
C'est un cas classique d'utilisation d'interface :

on a besoin de stocker un ensemble de bâtiments :

- on choisit l'interface Set et l'implémentation HashSet :

```
Set<Batiment> batiments = new HashSet<>();
```

- puis on utilise les méthodes de Set :

```
batiments.add(new Batiment (...));
```

- si un jour on veut utiliser TreeSet au lieu de HashSet, une seule ligne à changer :

```
Set<Batiment> batiments = new TreeSet<>();
```

Une classe peut implémenter plusieurs interfaces

Salle.java

```
class Salle implements ISalle , ICoordonnees {

    // methodes de ISalle :

    @Override
    boolean salleMachineAu3eme() {
        return this.salleMachine && this.numero/100==3;
    }

    @Override
    void afficherPlan() {
        System.out.println (...);
    }

    // methodes de ICoordonnees :

    @Override
    float latitude() {
        ...
    }

    @Override
    float longitude() {
        ...
    }
}
```

Partie 2 : Spécificités, interfaces et packages

Packages

En Java, chaque classe se trouve dans un fichier, mais aussi dans un **package**, qui sert simplement à regrouper des classes (*namespace*).
→ traduit parfois par “paquetage”

Cela forme une **arborescence**, qui se retrouve à l'identique dans le système de fichiers.

Fichier : src/gestionsalles/Salle.java

```
package gestionsalles;  
  
class Salle { ... }
```

Fichier : src/gestionsalles/io/ExportCSV.java

```
package gestionsalles.io;  
  
class ExportCSV { ... }
```

Packages

Intérêt : structurer le code.

Tout le code se trouve dans les packages (classes mais aussi interfaces, etc).

Partie 2 : Spécificités, interfaces et packages

Visibilité et modificateurs d'accès

Pour isoler des portions de code, on définit la **visibilité** des classes / méthodes / attributs, en utilisant des **modificateurs d'accès**.

Limiter l'accès, c'est rendre indépendant de l'implémentation

→ évite de propager un choix d'implémentation

→ même principe qu'avec les interfaces : `ISalle s = new Salle()`

Ça n'a l'air de rien, mais c'est très important pour garder un code souple, lisible, robuste.



Modificateurs d'accès : classe

Pour une classe, il n'y a que 2 niveaux de visibilité, définis par les modificateurs d'accès :

- **public** : classe accessible de n'importe quel package

```
package gestionsalles;  
  
public class Salle { ... }
```

- **(pas de modificateur)** : classe accessible seulement depuis son package ("package-private")

```
package gestionsalles.io;  
  
class ParserCSV { ... }
```

→ inaccessible depuis Salle (classe utilitaire pour ExportCSV, pas besoin ailleurs).

Modificateurs d'accès : attributs et méthodes

Pour les attributs et les méthodes, il y a 4 niveaux de visibilité, définis par les modificateurs d'accès :

modificateur	attribut/méthode accessible :
public	de <u>n'importe où</u>
private	uniquement depuis <u>sa classe</u>
(rien) ("package-private")	seulement depuis <u>son package</u>
protected	uniquement depuis <u>son package et ses sous-classes</u> (à venir, Partie 3)

Visibilité : exemple

Salle.java

```
public class Salle implements ISalle, ICoordonnees {  
  
    // attributs inaccessibles en ecriture depuis l'exterieur :  
    private float latit, longit;  
  
    ...  
  
    // acces en lecture a la latitude :  
    @Override  
    public float latitude() {  
        return latit;  
    }  
  
    ...  
  
    // une methode interne :  
    private float convertToStandardLatitude() {  
        return latit * ...;  
    }  
}
```

Partie 2 : Spécificités, interfaces et packages

Mini-projet : v3 → v4

Niveau 1

- ➊ gérer la visibilité des classes / attributs / méthodes
- ➋ modifier les packages : `boardgames.game.fill` et `boardgames.io`
- ➌ utiliser une interface `IGame` avec une méthode
`void updatePlay(GameState state, Action action);`

Niveau 2

- ➊ ajouter le jeu “crush” : comme “fill”, mais le joueur gagne dès qu’il place un pion sur l’un de ceux de son adversaire
→ seul `updatePlay` diffère. . .