



# Développement Logiciel Java

03/02/2025



# L'approche Objet

- La notion d'objet consiste à modéliser notre code en fonction de concepts réels correspondant aux besoins métiers.
- Ces objets contiennent alors des attributs qui les modélisent et définissent leur état
- Ils possèdent des comportements qui leurs sont propres et qui leur permettent de modifier leur état ou d'interagir avec l'extérieur.
- Certaines propriétés ou comportements sont accessibles depuis l'extérieur, d'autres sont privés et accessibles uniquement depuis l'intérieur de l'objet (notion d'encapsulation).



# La notion de classe

La classe est le moule à partir duquel seront construits les objets d'un même type.

Elle est définie dans un fichier dédiée et elle contient:

- Les attributs (paramètres) de l'objet. Ils peuvent être de type primitifs ou être également des références à d'autres classes (composition).
- Les comportements de l'objet. Il s'agit de méthodes avec leurs signatures et comportements.

Instancier un objet revient à réserver une zone mémoire pour les attributs propres à sa classe et à stocker la référence dans une variable. Chaque instance est unique mais plusieurs variables peuvent référencer la même instance.



# L'encapsulation

- La bonne pratique en java consiste à n'exposer hors de la classe que ce qui doit légitimement l'être.
- En particulier les paramètres ne devraient jamais être lus ou écrits directement mais au travers de méthodes dédiées (accesseurs et mutateurs).



# L'héritage

- En modélisant des situations complexes, il arrive souvent de pouvoir déterminer des comportements et propriétés communs à des catégories d'objets différents.
- L'héritage permet de définir des classes "parents" qui modélisent ces caractéristiques communes.
- Les classes héritées peuvent ensuite ajouter des paramètres, des méthodes, ou même surcharger celles de leur parent pour implémenter des comportements différents.
- En java on ne peut hériter que d'une seule classe, mais plusieurs niveaux d'héritages sont possibles.



# Le polymorphisme

- Polymorphisme des objets:

Si plusieurs classes héritent d'un même type *parent* (même à plusieurs niveaux d'héritage), le compilateur est tout à fait capable d'exploiter différents d'objets de type *parent* sans s'inquiéter de leur classe finale effective. Au moment de l'exécution, il déléguera l'exécution des méthodes à la bonne sous-classe sans se tromper.

- Polymorphisme des méthodes:

Il est tout à fait possible de définir plusieurs fois la même méthode avec des paramètres différents. Le compilateur est capable de déterminer à quelle méthode vous faites référence en fonction des paramètres que vous lui passez.



# Niveau d'abstraction

- Compte tenu de la capacité d'héritage et de polymorphisme de java, il est toujours intéressant de se poser la question du niveau d'abstraction applicable à notre code.
- Il convient donc de toujours typer les variables que l'on manipule avec le plus haut niveau d'abstraction disponible afin de découpler un maximum le code de l'implémentation.
- Exemple: Si mon système RH contient un type *Employee* dont héritent les classes *Developer* et *Architect*. La méthode *GeneratePaySlip()* qui génère la fiche de paie n'a sans doute pas besoin d'être implémentée pour chacune des deux sous-classe, mais peut se contenter de gérer des *Employee*.
- De la même manière, lorsque je manipule des listes en java, je n'ai pas besoin d'en connaître le détail d'implémentation, le type *List* est bien souvent suffisant.



# Notion de type

- En Java, toute variable doit être déclarée avec un type qui ne changera pas.
- Le type de la variable sert de garde-fou au compilateur qui vérifie que les informations qu'on essaie d'y écrire sont les bonnes.
- Il existe deux types de variables:
  - Types primitifs
  - Types références



# Les Types primitifs

- Les types primitifs ne sont pas des objets mais de simples conteneurs pour des valeurs.
- Ils ne possèdent donc pas de méthodes ou d'attributs.
- A chaque type primitif est associé un wrapper objet l'encapsulant, et lui associant des méthodes telles que `toString()`
- Les wrappers objets n'ont pas besoin d'être instanciés avec `new`.

# Les Types primitifs

Type Primitif	Valeurs	Type Wrapper
boolean	true ou false	Boolean
char	0 à 65535	Char
byte	-128 à 127	Byte
short	-32768 à 32767	Short
int	$-2^{31}$ à $2^{31}-1$	Integer
long	$-2^{63}$ à $2^{63}-1$	Long
float	$-3.40282347E+38$ à $3.40282347E+38$	Float
double	$-1.79769313486231570E+308$ à $1.79769313486231570E+308$	Double



# Les Types références

- Contrairement aux types primitifs, les types références stockent des références sur une zone mémoire (le *heap*).
- Tant que la zone mémoire est référencée depuis le programme, la zone mémoire est maintenue.
- Dès qu'aucune référence ne pointe vers une zone mémoire, celle-ci est détruite par le garbage collector



```
// Création d'une variable référence nommée tr1
// sur un objet de type TestReference
TestReference tr1; // pour l'instant tr1 vaut null
// Allocation d'un objet et stockage de son adresse
// (donc de sa référence) dans la variable tr1
tr1 = new TestReference();

// Création d'une variable référence nommée tr2
// sur un objet de type TestReference
TestReference tr2; // pour l'instant tr2 vaut null
// Copie le contenu de tr1 donc de l'adresse de l'objet
// dans tr2
tr2 = tr1;
```



# Les Types Références

- Une comparaison d'égalité entre deux variables de types références se fera sur l'adresse mémoire et non sur les objets contenus à l'intérieur.
- Une paramètre de méthode, une valeur de retour transmettent de la même manière les adresses des objets référencés, et non les objets eux même.



# La Superclasse Object

- Tous les objets héritent implicitement par défaut de la classe Object.
- Object définit un certain nombre de méthodes que chaque classe est ensuite libre de surcharger au besoin:
  - equals
  - hashCode
  - toString
  - finalize
  - getClass, .class et instanceof
  - clone
  - notify, notifyAll et wait



# La méthode equals

- La comparaison `==` sur des types références ne compare que le contenu des variables (les adresses mémoires) et ne donnera donc `true` que si les variables réfèrent strictement le même objet Java (même adresse mémoire).
- Dans des cas réels, deux objets peuvent être considérés égaux selon d'autre critères (si certains de leurs paramètres sont égaux par exemple).
- C'est pourquoi il peut être utile de ré-écrire la méthode `equals` pour proposer un test d'égalité pertinent.



# La méthode `HashCode`

- Permet d'obtenir un hash (integer) à partir des propriétés d'un objet.
- Utilisé pour optimiser le stockage de données notamment dans les HashMaps.
- Si deux objets sont égaux par la méthode `equals` ils **doivent** avoir le même `hashCode` (l'inverse n'est pas vrai, il peut y avoir collision).
- Si l'on redéfinit `equals` il faut donc redéfinir `hashCode` également.



# La méthode `toString`

- Permet de définir le comportement appliqué à l'objet quand il doit être présenté sous forme de chaîne de caractères.
- Facilite le débug en affichant les caractéristiques importantes de l'objet dans la console.
- La méthode `System.out.println` appelle automatiquement `toString` sur l'objet qui lui est passé avant de l'afficher.



# Déclarer une classe

```
visibilité class NomClass [extends Parent] implements [Interface1  
Interface2...] {  
  
    // Corps de la classe  
  
}
```

## Visibilité:

- public: visible par toute la base de code
- Pas de visibilité: visible par les autres éléments du même package
- private et protected n'ont pas de sens ici (sauf pour une classe imbriquée)



# Déclarer une classe - Attributs

```
visibilité type nomAttribut [=valeur ou référence];
```

Visibilité:

- **public** : accès sans restriction
- **protected** : pour un accès limité à la classe, ses héritiers et à tous les objets du package qui l'héberge;
- **<pas de valeur>**: pour un accès limité à la classe et à tous les objets du package qui l'héberge ;
- **private**: pour un accès limité au type de la classe.



# Déclarer une classe - Attributs

- Attributs constants: ajouter le mot-clé final.
- static: méthodes et attributs attachés à la classe et non à ses instances.



# Déclarer une classe - Constructeur

- Lors de l'instanciation d'un objet avec le mot-clé `new`, le système:
  - Demande au système d'exploitation un morceau de mémoire de la taille d'un objet Client ;
  - Exécute le constructeur de chaque attribut défini dans la classe provoquant ainsi une initialisation par défaut (exemple : les valeurs numériques sont toutes à 0 et les références sont positionnées à null pour non allouées) ;
  - Recherche si vous avez redéfini un constructeur pour cette classe Client et, si tel est le cas, l'exécute ;
  - Retourne au programme appelant une référence sur l'objet nouvellement alloué.
- On définit un constructeur avec une méthode qui porte le même nom que la classe et n'a pas de valeur de retour.



# Surcharge et chaînage des constructeurs

- De la même manière qu'avec une méthode, il est possible de surcharger les constructeurs avec différents paramètres.
- Il est possible de chaîner les constructeurs (de la même classe ou d'une superclasse) avec le mot-clé *this()*.



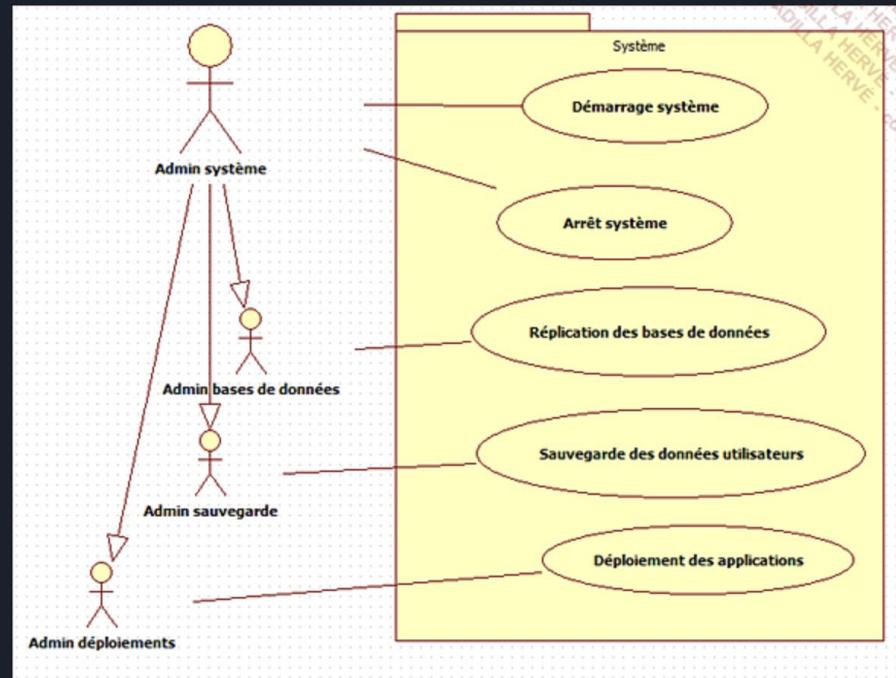
# Unified Modeling Language

Différents types de diagrammes modélisant la structure et le comportement d'une application

- **Diagrammes de cas d'utilisation** : ils décrivent les services rendus par le système du point de vue de l'utilisateur. Ces vues mettent en scène des acteurs qui peuvent être humains ou représenter d'autres systèmes.
- **Diagrammes d'objets** : ils montrent l'état d'une application à un instant donné en nommant les instances des classes.
- **Diagrammes de séquences** : ils montrent les interactions entre les objets pendant l'exécution du programme. L'accent est donné sur l'ordre de ces interactions dans cette représentation temporelle.
- **Diagrammes de classes** : ils capturent la structure statique d'organisation des classes.
- **Diagrammes de composants** : ce sont des vues modulaires de l'application qui regroupent des classes qui collaborent.
- **Diagrammes de déploiement** : ils modélisent l'aspect matériel de l'application.
- **Diagrammes de collaboration** : ils montrent comment des objets sont organisés pour travailler ensemble. L'accent est donné sur les communications existantes entre les objets.
- **Diagrammes d'états-transitions** : ils représentent le comportement d'un objet sous la forme d'un automate à états.
- **Diagrammes d'activités** : ils représentent le flux d'exécution d'un processus ou d'une opération.

# Diagrammes de cas d'utilisation

Le rôle des diagrammes de cas d'utilisation est de délimiter le périmètre de l'application en précisant ses « acteurs » et les différents scénarios possibles qu'ils peuvent jouer dans le système. Un cas d'utilisation représente un service fonctionnel de l'application décrit dans le cahier des charges.



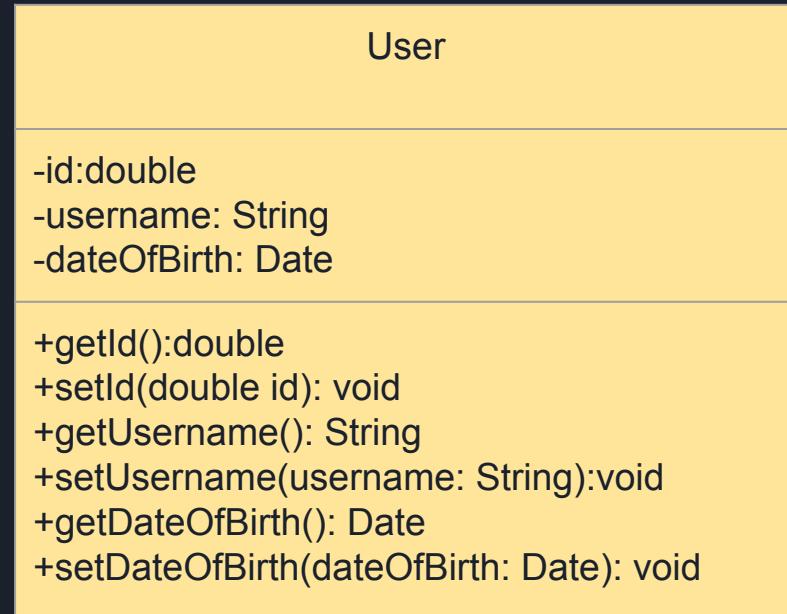
# Diagrammes de classe

Une classe est représentée par un rectangle divisé verticalement en trois parties :

- En haut : le nom de la classe.
- Au milieu : les attributs de la classe (les variables).
- En bas : les comportements de la classe (les méthodes).

Le caractère de début de ligne indique la visibilité d'une méthode ou d'un paramètre:

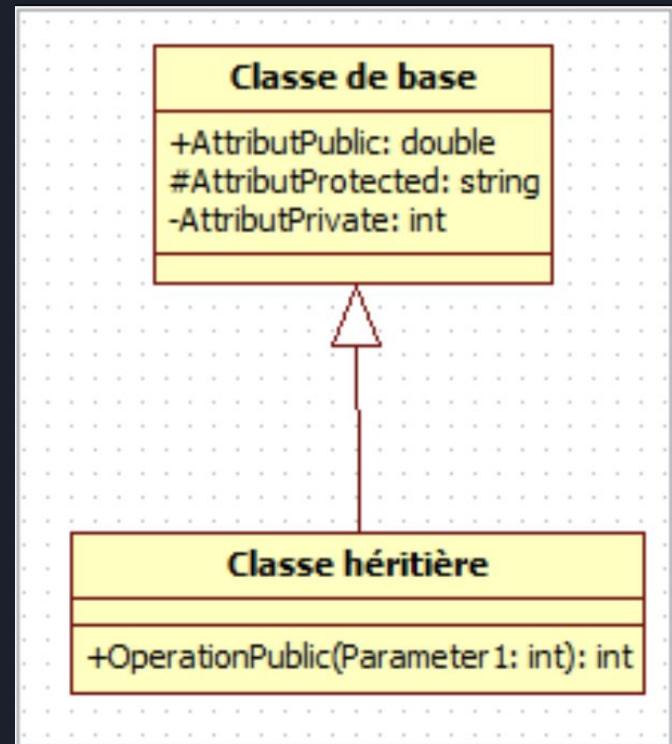
- Public: “+”
- Protected: “#” (uniquement visible aux classes enfants)
- Private: “-”



# Diagramme de classe, héritage

L'héritage est représenté par une flèche au triangle fermé, allant de la classe dérivée vers la classe mère.

La classe enfant hérite des attributs et méthodes *public* et *protected* du parent.





# Diagramme de classe, réalisation

Dans le cas d'une classe implémentant une interface, on procède comme pour l'héritage, mais la ligne est en pointillés.

Une même classe peut implémenter plusieurs interfaces.

# Diagrammes de classes - Relations

Relation simple:



Relation bi-directionnelle:



# Diagrammes de classes - Agrégation & Composition.

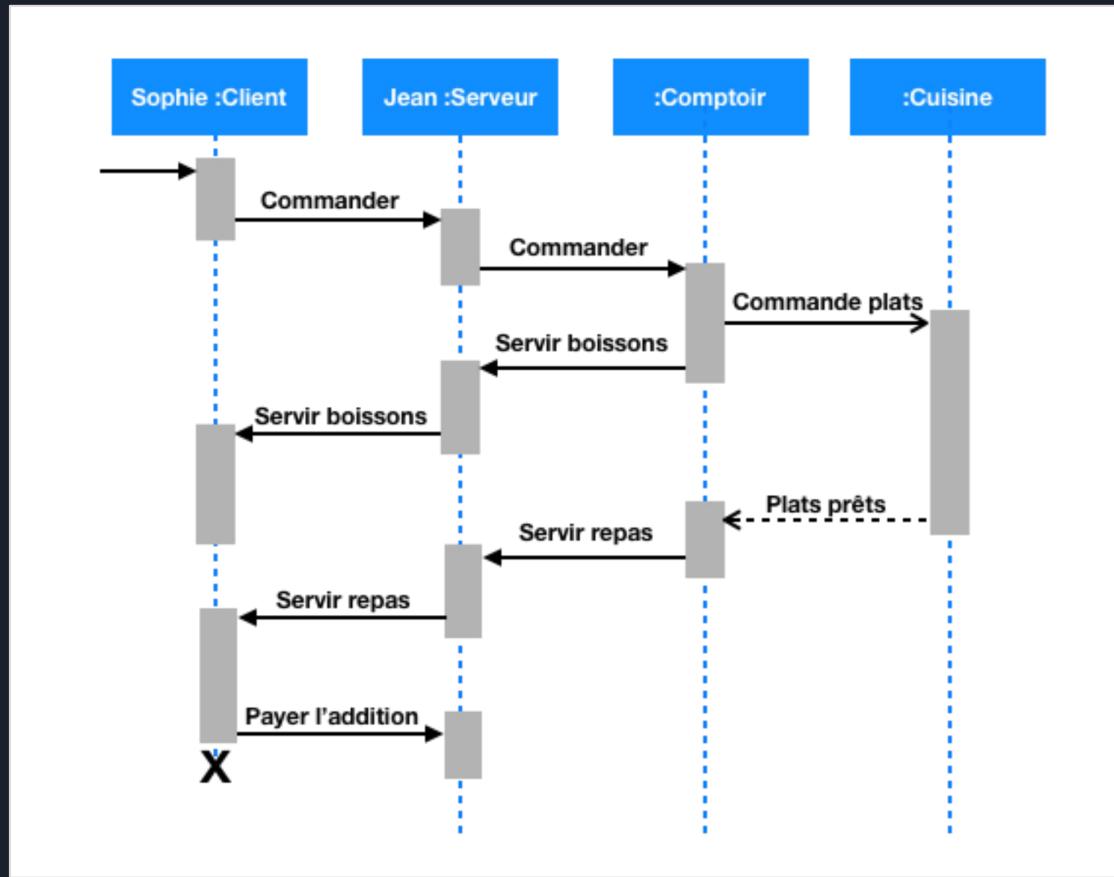
L'agrégation et la composition sont des cas particuliers de relations dénotant une “appartenance” d'une classe à une autre.



Si le losange est vide c'est une agrégation (l'article peut exister si la commande est détruite).

Si le losange est plein, c'est une composition (si la commande est détruite, les articles sont détruits).

# Diagrammes de Séquence



# Exercice

Notre système d'application TODO List commence à prendre forme.

Un utilisateur (*User*) contient les informations suivantes:

- Un identifiant unique (*id*) généré automatiquement à la création et en lecture seule
- Un prénom (*firstName*)

Une tâche (*Task*) peut contenir les informations suivantes:

- Un identifiant unique (*id*) généré automatiquement à la création et en lecture seule
- Un titre (*title*)
- Une description (*description*)
- Un indicateur de completion (“done”)
- Une référence à l'utilisateur l'ayant créée

Additionnellement, une tâche avec échéance (*DatedTask*) est un type de tâche particulier contenant une information supplémentaire:

- Une date d'échéance (*dueDate*)

Créez le diagramme UML de ces objets, puis leurs implémentations avec les méthodes `equals`, `hashCode` et `toString`.

Modifiez votre programme pour être capable de lister, créer, supprimer et modifier les tâches.