



Cursus JAVA

M2I Formations 2021

Christian Lisangola



MODULE JAVA

Les fondamentaux de la POO

Introduction

1.1.



Niveau de perception d'un objet

Il y a deux niveaux de perception d'un objet :

- ❑ **le niveau externe, visible, utile au programmeur utilisateur**, celui qui utilise l'objet et voudrait, par exemple, effectuer un calcul de surface. Cet utilisateur n'a pas besoin de connaître les détails techniques des différentes méthodes, tout comme nous n'avons pas besoin de savoir comment un moteur fonctionne pour pouvoir conduire une voiture.
- ❑ **le niveau interne, qui concerne le programmeur concepteur**, celui qui se charge des détails d'implémentation, comme définir la façon de calculer la surface d'un rectangle.

1.2.

Classes, Objets, Attributs et Méthodes

Classes



Une classe est un nouveau **type de données** dont les instances sont des objets.

Voici la syntaxe de déclaration :

```
class Personne{...}
```

Une fois la nouvelle classe définie, il est possible de déclarer des variables du nouveau type, qui sont des objets :

```
Personne moi ;
```

Classes



Lorsque l'on souhaite déclarer plusieurs classes dans un même programme, il existe deux façons de faire :

1. La première consiste à déclarer plusieurs classes dans un même fichier. Ici la commande “**java nom_fichier.java**” va être lancée pour la compilation et créer plusieurs fichier “**nom_fichier.class**” pour chaque classe.
2. Déclarer une classe par fichier, puis de compiler chaque fichier séparément.
C'est cette dernière qui sera la plus souvent utilisée.

Attributs



Les attribut sont aussi appelés les “variables membres”, car ils sont membres, donc les données liées à l’objet.

Voici la syntaxe de définition des attributs :

```
class NomClass{  
    type nom_attr1;  
  
    type nom_attr2;  
    ...  
}
```

Attributs

```
class Personne {  
    String nom;  
    String prenom;  
    String pays;  
    boolean estMarie;  
    int nombreEnfants;  
}
```

Création des objets

La création d'un objet ou d'une instance d'une classe est similaire à la construction d'une maison à partir d'un plan, et cette étape est appelée **instanciation**.

Syntaxe :

NomClasse nomInstance=new NomClasse()

Plus tard, nous allons expliquer la **signification des parenthèses** dans l'instanciation.

Pour accéder au attributs ou variables membres d'un objet :

nomInstance.nomAttribut

Pour affecter une valeur à un attribut : **nomInstance.nomAttribut=valeur;**

Création des objets : initialisation

NomClasse nomInstance=**new** NomClasse()

Cette ligne crée une instance de type **NomClasse** et initialise tous ses attributs avec des valeurs par défaut :

- les entiers sont initialisés à 0,
- les doubles à 0.0,
- les booléens à false
- les objets à null.

Création des objets

Voici un exemple d'objet :

```
public static void main(String[] args){  
    Scanner keyb=new Scanner(System.in);  
    Personne moi=new Personne();  
    System.out.print("Etes-vous mariés ? ");  
    moi.estMarie= keyb.nextLine().equals("oui");  
    System.out.print("Votre prenom et nom : ");  
    moi.prenom=keyb.nextLine();  
    moi.nom=keyb.nextLine();  
    System.out.print("Combien d'enfants avez-vous ? ");  
    moi.nombreEnfants= keyb.nextInt();  
    System.out.printf("Nom : %s\n", moi.prenom,moi.nom);  
    System.out.printf("Etat martial : %s\n",moi.estMarie?"Marié":"Célibataire");  
    System.out.printf("Enfants : %d\n", moi.nombreEnfants);  
}
```

Méthodes



Les méthodes ou fonctions membres sont des fonctions qui sont déclarées au sein d'une classe.

Les paramètres d'une méthode sont **des variables externes à la classe** : chaque méthode d'une classe a accès aux attributs de celle-ci, qui **ne doivent donc pas être passés en arguments**.

On parle de portée de classe : les attributs sont des variables globales à la classe.

Pour invoquer une méthode : **nomInstance.nomMethode(arguments)**;

Méthodes

```
class Personne {  
    String nom;  
    String prenom;  
    boolean estMarie;  
    int nombreEnfants;  
  
    String nomComplet(){  
        return prenom+" "+nom;  
    }  
}
```

```
System.out.printf("Nom complet : %s\n" moi.nomComplet());
```

Public & private

1.3.

Private



Le mot-clé **private** est employé pour signaler quelle partie de la classe restera inaccessible à un utilisateur de la classe. Tout élément de la classe déclaré avec le mot-clé **private** est donc **un détail d'implémentation, inaccessible depuis l'extérieur de la classe.**

C'est dans cette catégorie que l'on déclare toujours les attributs, comme dans la classe, mais également **certaines méthodes internes qui ne peuvent être appelées qu'au sein de la même classe.** Une tentative d'accès à un élément privé d'une classe depuis l'extérieur de celle-ci produira un message d'erreur à la compilation.

```
private String nom;  
private String prenom;  
private boolean estMarie;  
private int nombreEnfants;
```

Accesseurs et Manipulateurs



Les attributs sont en règle générale privés, inaccessibles à l'extérieur, il peut néanmoins être utile de les utiliser depuis l'extérieur de la classe, notamment pour modifier ou connaître leur valeur.

Pour cela, on peut proposer des **méthodes publiques** pour accéder aux attributs, en lecture ou en écriture .

- Les méthodes qui retournent la valeurs d'un attributs(privés) sont les “accesseurs” ou “**getters**” .
- Les méthodes qui permettent de modifier la valeur d'un attribut(privé) sont appelés manipulateurs ou “**setters**”. Ces méthode prennent toujours un paramètre qui sera ensuite affecté comme valeur à l'attribut sans rien retourner(retourne un void).

Accesseurs et Manipulateurs

```
public String getNom() {  
    return nom;  
}
```

```
public void setNom(String nom) {  
    this.nom = nom;  
}
```

Accesseurs et Manipulateurs



Le fait de garder les attributs privés et de ne les manipuler qu’au travers des méthodes permet de prévenir les erreurs de l’utilisateur :

Le concepteur peut par exemple imposer qu’une hauteur de rectangle soit positive dans une classe Rectangle, garantissant ainsi l’intégrité des données.

De plus, l’interface permet au concepteur de librement modifier sa représentation interne de la classe sans que l’utilisateur n’en soit affecté.



TP 1

Ecrire un programme Java qui permet de calculer les dimensions d'un Rectangle.

MODULE JAVA

2. Constructeurs

2.1.

Introduction

Introduction



Jusque-là nous avons créé des objets sans les initialiser, c'est-à-dire leur fournir des valeurs initiales lors de l'instanciation.

La solution que nous avons utilisée jusqu'à présent est d'utiliser des setters afin de mettre à jour les valeurs des attributs après la création de l'instance.

Cette méthode est toutefois mauvaise dans le cas général pour 2 raisons:

- elle implique qu'il y ait un manipulateur pour chaque attribut même quand celui-ci n'est pas nécessaire ou que ces attributs soient publics.
- Elle oblige également l'utilisateur à initialiser individuellement tous les attributs, au risque que certains soient oubliés.

Constructeurs



Un constructeur est une méthode spéciale à invoquer systématiquement lors de la déclaration d'un objet et qui est chargée d'effectuer toutes les opérations requises en début de vie de l'objet en question, dont l'initialisation des attributs.

La syntaxe d'un constructeur est la suivante :

```
NomClass(param_1, param_1,...){  
    this.attr_1=param_1;  
    this.attr_2=param_2;  
    ...  
}
```

Constructeurs



Les constructeurs diffèrent des méthodes traditionnelles sur quelques points :

- ils n'ont pas de type de retour(pas même un void),
- et **ils doivent avoir le même nom que la classe.**

Ils peuvent par contre être surchargés.

```
public Personne(String nom, String prenom, boolean estMarie, int nombreEnfants) {  
    this.nom = nom;  
    this.prenom = prenom;  
    this.estMarie = estMarie;  
    this.nombreEnfants = nombreEnfants;  
}
```

Constructeurs : initialisation



Voici la syntaxe de déclaration d'un objet avec initialisation:

```
Personne moi=new Personne( nom: "Lisangola", prenom: "Christian", estMarie: true, nombreEnfants: 8);
```

2.4.

**Fin de vie,
affichage et
comparaison**

Un objet est en fin de vie lorsque le programme n'en a plus besoin, en d'autres termes , lorsque la référence vers cet objet n'est plus utilisée dans la suite du programme.

Afin de permettre à ce que cette zone mémoire soit utilisée pour d'autres tâches ,il est donc utile de libérer la zone mémoire qui était associée à l'instance.

Dans certains langages de programmation, le programmeur doit spécifier explicitement que l'objet est en fin de vie et qu'il faut donc libérer la zone mémoire qui lui est allouée, ce qui n'est pas le cas en Java.Il existe un programme « **ramasse-miettes** », ou « **garbage collector** » en anglais, qui s'occupe de cette tâche et qui est lancé périodiquement pendant l'exécution d'un programme Java.

Affichage



Lorsque l'on veut afficher un objet dans la console au moyen de la méthode `println`, seule la référence vers l'objet est affichée, ce qui n'est pas très utile.

Afin de remédier à ce problème, Java prévoit qu'une méthode retourne une représentation de l'instance sous forme d'une chaîne de caractères. Cette méthode doit être déclarée dans une classe de l'objet que l'on souhaite afficher et avoir la syntaxe suivante :

String toString() {...}

Cette méthode est ensuite invoquée automatiquement par la méthode `println`.

```
public String toString(){
    return "Hauteur : "+this.hauteur+"\nLargeur : "+this.largeur;
}
```

```
Rectangle rect1=new Rectangle( hauteur: 2.0 , largeur: 3.0 );
System.out.println(rect1);
```

Comparaison



La comparaison de deux objets avec l'opérateur `==` ne compare que les références des objets et non leurs valeurs.

Java prévoit donc une méthode dédiée à la comparaison d'objets : **la méthode `equals`**, que l'on a déjà utilisée dans le cours pour comparer les chaînes de caractères.

Il suffit donc d'écrire une méthode `equals` de la manière suivante :

```
public boolean equals(Rectangle rect){  
    if(rect==null){  
        return false;  
    }  
    return this.largeur==rect.getLargeur() && this.hauteur==rect.getHauteur();  
}
```

```
Rectangle rect1=new Rectangle( hauteur: 2.0, largeur: 3.0);  
Rectangle rect2=new Rectangle(rect1);  
System.out.printf("Rectangle 1 = Rectangle 2 : %b\n", rect1.equals(rect2));
```

TP 4 : Laboratoire



Le but de cet exercice est de créer des « souris » par différents biais et de les faire « évoluer » au cours du temps.

Voici les détails de l'exercice : <https://bit.ly/3wm7D8V>

TP 5 : Bibliothèque



Le but de cet exercice est de simuler de façon très basique la gestion d'une bibliothèque. La bibliothèque contient des exemplaires d'œuvres écrites par des auteurs.

Il s'agira de modéliser chacun de ces éléments dans votre programme.

Voici les détails de l'exercice : <https://bit.ly/3wm7D8V>

MODULE JAVA

3. Héritage

3.1.

Concepts

Après les notions d'**encapsulation** et d'**abstraction**, la troisième notion fondamentale de la programmation orientée objet est **l'héritage**. Pour illustrer cette notion, on peut imaginer vouloir représenter **les personnages d'un jeu**.

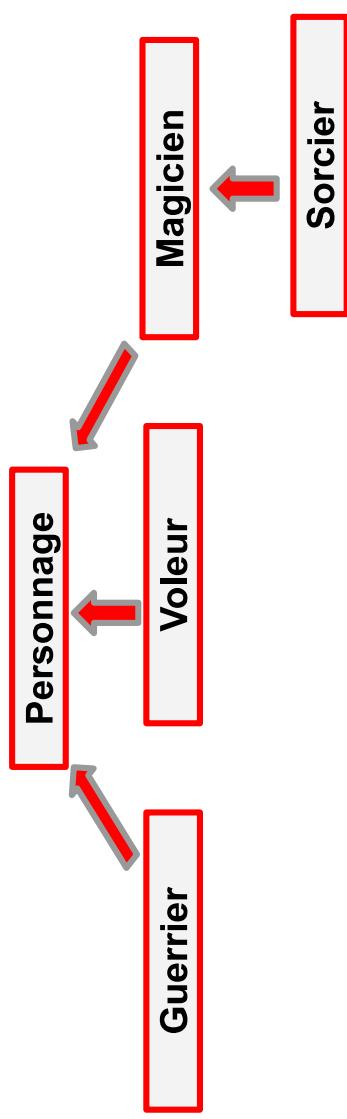
On pourrait créer une classe spécifique à chaque type de personnage, qui aurait comme attributs **un nom, une durée de vie** mais aussi des éléments propres à son identité comme **une arme pour un guerrier**. Ces classes partageraient également une méthode pour rencontrer d'autres personnages qui, dans le cas d'un voleur, lui permettrait de **les voler**.

Avec une telle solution il y aura beaucoup de duplication de code et poserait des problèmes de maintenance.

En regardant les classes ci-dessous, nous pouvons voir beaucoup de duplication des attributs(**nom**, **dureeDeVie**) et de méthodes(**rencontrer**)

Guerrier	Voleur
- nom: String	- nom: String
- dureeDeVie: int	- dureeDeVie: int
- arme: Arme	- arme: Arme
+ rencontrer(): void	+ rencontrer(): void
+ voler(): void	+ voler(): void

Sorcier	
- nom: String	- nom: String
- dureeDeVie: int	- dureeDeVie: int
- baguette: Baguette	- baguette: Baguette
- batonMagique: BatonMagique	- batonMagique: BatonMagique
+ rencontrer(): void	+ rencontrer(): void



Héritage : Syntaxe



```
class Personnage{  
    private String nom;  
    private int dureDevie;  
    public void rencontrer(Personnage p){}  
  
}  
  
class Guerrier extends Personnage{  
    private String arme;  
}  
}
```

3.2.

Héritage : Droit d'accès protected

Protected



Dans le cadre d'une relation d'héritage, la sous-classe dispose aussi des membres privés de la super-classe **mais n'y a pas accès**. En effet, le droit d'accès privé limite la visibilité à l'enceinte de la classe, ce qui constraint une éventuelle sous-classe à utiliser les getters et setters prévus. Il existe un troisième type d'accès au sein d'une hiérarchie de classe : **le droit d'accès protégé**.

Le droit d'accès protégé assure la visibilité des membres d'une classe dans toutes les classes de sa descendance et se désigne par le mot-clé protected dans la même syntaxe d'utilisation que public et private. Le niveau d'accès protégé est une **extension du niveau privé**, qui accorde des droits d'accès privilégiés à toutes les sous-classes, mais également à toutes les classes du même paquetage, ce qui nuit à une bonne encapsulation. Il est donc peu recommandé d'utiliser ce droit d'accès protégé en Java.

3.3.

Héritage : Constructeurs

Héritage : Les constructeurs

```

class Personnage{
    private String nom;
    private int dureeDevie;

    public Personnage(String nom,int dureeDevie){
        this.nom=nom;
        this.dureeDevie=dureeDevie;
    }

    public void rencontrer(){
        System.out.println("Bonjour à tous!");
    }
}

class Guerrier extends Personnage{
    private String arme;

    public Guerrier(String arme,String nom,int dureeDevie){
        super(nom,dureeDevie);
        this.arme=arme;
    }

    @Override
    public void rencontrer() {
        super.rencontrer();
        System.out.println("Bonne après midi");
    }
}

```

Il faut cependant noter que si la super-classe possède un constructeur par défaut(sans paramètre), il est automatiquement invoqué par le compilateur, donc pas besoin de l'invoquer explicitement avec **super(arguments)**. Il est recommandé de toujours déclarer au moins un constructeur , ceci permet à la sous-classe d'initialiser les membres.

MODULE JAVA

4. Polymorphisme

4.1.

Introduction

Introduction



Après l'encapsulation,l'abstraction et l'héritage,la dernière notion fondamentale de la programmation orientée objet est **le polymorphisme**.

Le mot polymorphie vient du grec et signifie "**qui peut prendre plusieurs formes**". Dans la programmation orientée objet, le polymorphisme est utilisé en relation avec les **fonctions**, les **méthodes** et les opérateurs(c++ par exemple).

Des **fonctions/méthodes** de mêmes noms peuvent avoir des comportements différents ou effectuer des opérations sur des données de types différents.

4.2.

Classes et méthodes abstraites

Introduction



Avec l'héritage, il est possible de **généraliser des concepts** qui sont présents dans plusieurs classes du programme en **les incorporant à une super-classe**. Ainsi, on construit une structure arborescente reliant différents éléments à un niveau plus abstrait. **Cela fait partie des mécanismes d'abstraction de l'orientée objet.**

Toutefois, au niveau le plus élevé d'une hiérarchie de classe, il est parfois impossible de définir et fournir l'implémentation d'une **méthode générale** qui devra pourtant exister dans toutes les sous-classes.

Par exemple, calculer la surface d'une figure géométrique quelconque se révèle difficile, tandis qu'à un niveau plus bas on peut le mettre en oeuvre au travers d'une classe Cercle qui héritera par exemple de la classe **FigureGeométrique**.

Il est donc possible de définir dans une super-classe des méthodes sans leurs fournir une implémentation, c'est-à-dire un corps.

Surcharge et redéfinition

Il existe en Java **deux façons possibles d'avoir des méthodes avec le même nom dans une seule classe** : la **surcharge (overloading)** et la **redéfinition (overriding)**.

- ❑ S'il existe plusieurs méthodes avec le même nom **dans une classe**, nous parlons de **surcharge**.
- ❑ La redéfinition n'existe que pour des méthodes héritées. Si une **sousclasse** définit une méthode qui est déjà spécifiée dans une super-classe en utilisant exactement le même nom, la même liste de paramètres et un type de retour compatible, alors la méthode est redéfinie. Il s'agit dès lors d'une méthode polymorphe.

Pour les **types de base**, un type de retour compatible est simplement ce même **type**. Par exemple, une méthode qui retourne un int ne peut qu'être redéfinie par une méthode qui retourne aussi un int. Pour les types évolués, il est possible de retourner un objet d'une sous-classe du type de retour de la méthode héritée.

MODULE JAVA

5. Le modificateur final, attribut et méthodes statiques

5.1.

Attributs/Variables statiques

Attributs statiques

Nous connaissons maintenant trois types de variables:

- Les variables locales qui sont déclarées dans un corps de méthode,
- les paramètres de méthodes
- les variables d'instance. Ces dernières sont aussi appelées les attributs et elles stockent les valeurs spécifiques à une instance de la classe en question.

Prenons pour exemple une classe Employe représentant un employé qui prendra sa retraite à l'âge de 60 ans.

Un attribut `ageRetraite` pourrait être défini dans la classe, contenant 65. Le **problème est que cette valeur sera stockée dans chaque instance séparément.**

Si la règle change et que l'âge de retraite est élevé à 65 ans, nous sommes obligés de changer cet attribut dans chaque instance, ce qui a comme conséquence, beaucoup de travail.Pour résoudre ce problème, nous introduire des variables classe.

Variables statiques



Les variables statiques sont des attributs qui vont être partagés par toutes les instances d'une classe.
Et si leur valeur change, elle change pour toutes les instances.

Ainsi, les variables statiques n'ont pas besoin d'objets pour être utilisables. Elles sont initialisées au chargement du programme et elles peuvent être appelées sans création préalable d'une instance de la classe. Un tel appel se présente ainsi :

```
class Employe{  
    private static int ageRetraite=65;  
}
```

```
Employe.ageRetraite = 67;
```

5.2.

Méthodes statistiques

Méthodes statiques

Tout comme les variables statiques, les méthodes statiques peuvent être invoquées sans instancier la classe. Nous avons déjà vu les méthodes de la classe **Math**, méthodes que nous invoquons sans créer une instance de la classe **Math** à l'instar de **Math.pow(...)**, **Math.sqrt(...)**, **Math.floor(...)**, etc.

- Il n'est pas permis d'utiliser des membres non statiques(variables, méthodes) au sein des méthodes statiques car **il n'est pas garanti qu'un objet existe lors de l'exécution du programme**.vu que ce méthode peuvent être invoqué sans qu'une instance ne soit créée.
- Il est permis d'appeler toutes les méthodes et variables statiques de la classe.
- Il est également possible de créer un objet dans une méthode statique et d'utiliser les méthodes non statiques de l'objet que l'on vient de créer via cet objet.

MODULE JAVA

6. Interfaces

Introduction



Java ne tolère que l'héritage simple ; chaque classe ne pouvant avoir qu'une seule super-classe.

Il s'agit là d'un choix des concepteurs du langage lié au fait que la gestion de l'héritage multiple peut être lourde ; cette notion générant des ambiguïtés potentielles, par exemple quand 2 classes dont une sous-classe hérite possède des méthodes possédant une même signature.

L'héritage multiple est possible dans d'autres langages de programmation, comme le C++.

Introduction



La notion **d'interface** parle à cette problématique. Elle permet **d'imposer un contenu à des classes sans pour autant mettre en place un lien d'héritage.**

La relation qui existe entre une interface avec les classes qui les implémente est une relation **“se comporte comme”** et non **“est un” comme dans l'héritage.**

Définition



Comme les classes abstraites, les interfaces ne peuvent pas être instanciées.
Elles ne possèdent pas de constructeur.

- Java ne nous oblige pas à ajouter le mot réservé **abstract** puisque **toutes ces méthodes seront nécessairement abstraites, ainsi que publiques.**
- Le mot-clé **public** n'est pas nécessaire non plus. Il en va de même pour les constantes d'une interface : elles sont toutes publiques.

Mais comme ce sont des constantes, elles sont aussi toutes finales et statiques. Ces constantes sont initialisées en même temps qu'elles sont déclarées :

```
public interface IMonInterface{  
    int MA_CONSTANTE=5;  
    void maMethode();  
    Int maMethode2(int param);  
}
```

```
Class MaClass implements IMonInterface{  
    @Override  
    void maMethode(){...}  
    Void maMethode2(int param){...}  
}
```

Lien de la démo : <https://bit.ly/3w9VzaN>

Interface et Héritage



Les interfaces peuvent être liées entre elles par un lien d'héritage. Les sous-interfaces héritent du contenu des super-interfaces, et le lien d'héritage est établi par le mot-clé extends.

interface A{...}

interface B extends A{...}

Interface C extends A{...}

Interfaces vs classes abstraites

Remarquons que l'introduction des méthodes abstraites **rapproche beaucoup les interfaces des classes abstraites.**

La question se pose donc de quand utiliser une interface et quand une classe abstraite ?

La principale différence entre les deux réside dans le fait que les **interfaces ne peuvent pas avoir d'état – de variables d'instance**. Elles ne possèdent que des **constants et elles n'ont pas de constructeur**.

Pour cette raison, nous utilisons des interfaces quand nous voulons **modéliser un comportement ou un lien fonctionnel qui est indépendant d'un éventuel état de l'objet**. Tandis que nous utilisons des classes abstraites quand les objets ont des états.

Exercices



https://docs.google.com/document/d/1JcpigSnuP8fHUCzmDpHuS4ZRU38JPCNquXw98_mFuw/edit?usp=sharing

THE END.