



POO / OPP : tout est objet !!





PROGRAMMATION ORIENTÉE OBJET

POO / OPP : TOUT EST OBJET !!

LES PILIERS DE LA POO

La programmation par objet est un **paradigme*** de programmation informatique.

Elle consiste en la définition et l'interaction de briques logicielles appelées objets

- un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.

Pour faire simple, la POO est une approche de la programmation informatique et du traitement des solutions aux problèmes (paradigme) en considérant les éléments logiciels comme des objets.

Comme tout concept qui se respecte, la POO est basé sur des piliers qui renforcent son adoption et voici **ces 5 piliers essentiels**:

1. **Objet (et classe)** : Nous l'avons déjà abordé en début de Java. Une classe est un moule à partir duquel on va créer un objet. L'objet étant une instance de la classe.
2. **Encapsulation** : Ce pilier est un mécanisme consistant à cacher les données et certaines implémentations de l'objet et à avoir accès aux données que par les services proposés par l'objet (méthodes dites publiques) Garantir l'intégrité des données et masquer l'implémentation de certaines méthodes de la classe.
3. **Héritage** : Ce principe consiste donc à créer une classe (classe fille) qui partagera les caractéristiques d'une autre classe (dite classe mère).
4. **L'abstraction** : permet à un programmeur de mieux concevoir en pensant en termes général plutôt qu'en termes spécifiques les différents comportements d'une classe.
5. **Polymorphisme** : **poly** qui signifie plusieurs et **morphe** qui signifie forme. En héritage, cela qui consiste à créer une méthode dans la classe mère qui sera polymorphe : *Ceci voudra dire que cette méthode aura donc plusieurs implémentations en fonction des classes filles qui vont les implémenter.*

*un paradigme est une manière de penser et d'organiser le code.

QU'EST-CE QU'UN OBJET ?

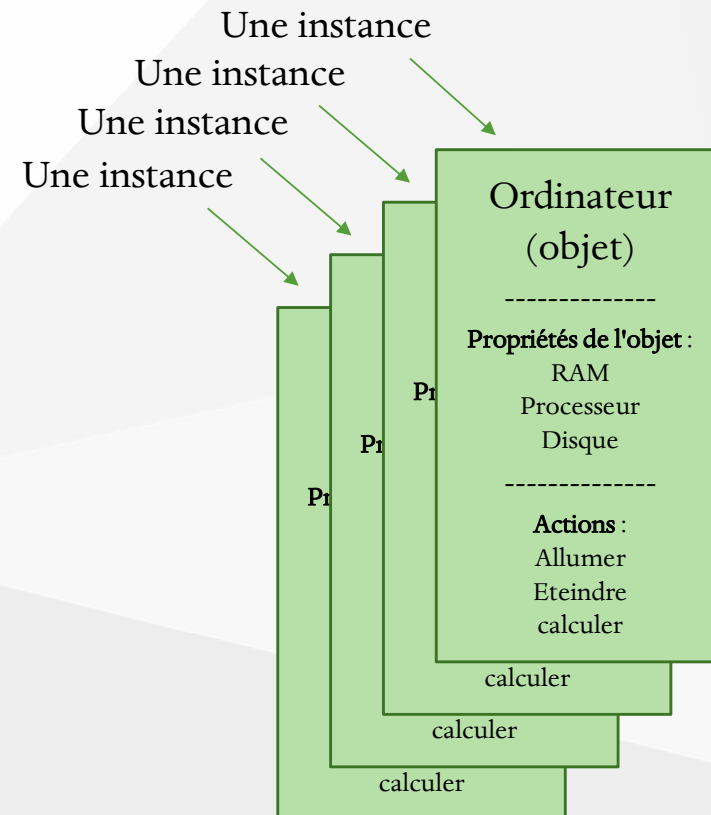
En POO, tout est objet et c'est pareil dans la vie réelle.

Par exemple - L'ordinateur est un objet :

- Il possède **des propriétés**, caractérisées sous forme de données. (RAM, processeur, disque etc..)
- Il peut réaliser **des actions** (s'allumer, s'éteindre, calculer, etc...)
- Il peut **interagir avec d'autres objets**. (un écran, un humain etc...)

La logique veut que l'on puisse avoir plusieurs objets, mais qui auront certainement des caractéristiques différentes.

- On parle alors **d'instance**.



COMPRENDRE LA NOTION DE CLASSE

En POO, un objet né de l'instance d'une classe.
(la base de la POO)

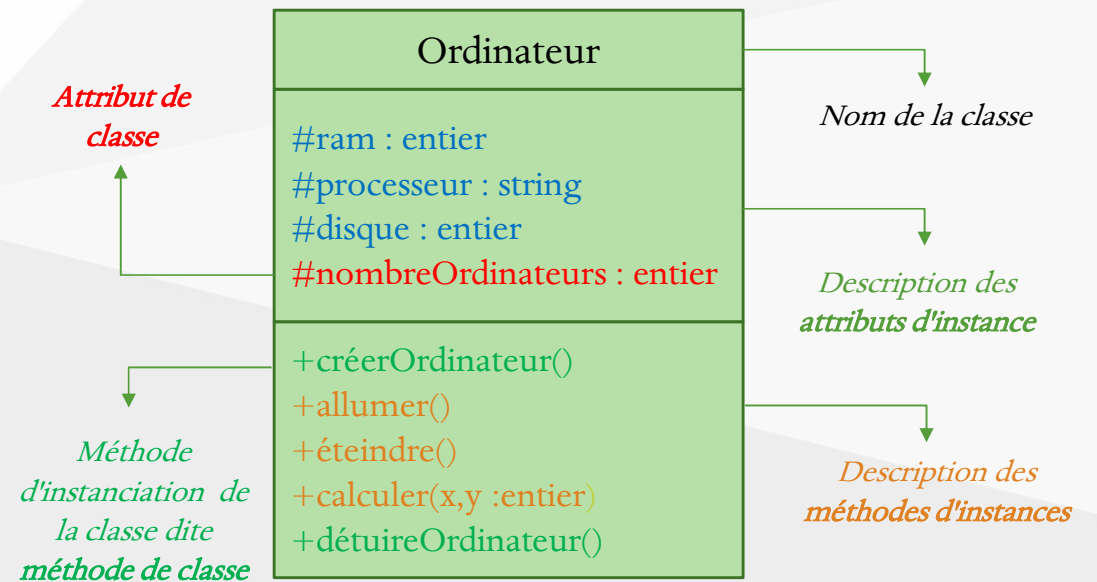
Une classe, c'est le schéma ou le plan qui nous permet d'instancier notre objet.

On peut instancier autant d'objets à partir d'une classe mais un objet généré ne peut changer de classe en cours de route.

En résumé, la classe est un catalogue de variables (propriétés) et d'actions internes (méthodes) qui interagissent entre eux pour donner un résultat.

Un objet instancié est mis en mémoire : il est référencé

Note : Il est possible d'utiliser des propriétés dans vos méthodes et vous pouvez appeler des méthodes dans d'autres méthodes.



COMPRENDRE LA NOTION DE CLASSE

Attributs d'instance

- Où sont déclarés les attributs d'instance ?
 - Ils sont écrits dans la classe, mais en dehors des méthodes, constructeurs ou blocs de code.
- Quand sont-ils créés ?
 - Ils sont créés quand on crée un objet avec le mot-clé `new`, et ils sont supprimés quand l'objet est détruit.
- Comment les utiliser ?
 - Dans la classe elle-même : tu peux utiliser directement leur nom.
 - En dehors de la classe ou depuis une autre instance : tu dois écrire : `nomDeLObjet.nomDeLAttribut`.
- ✓ Ces attributs contiennent des **valeurs importantes** pour l'objet.
- ✓ Plusieurs méthodes, le constructeur, ou d'autres blocs de code dans la classe ont besoin d'y accéder.
- ✓ Ces valeurs font partie de l'**état** de l'objet, c'est-à-dire ce qui définit l'objet à un moment donné.

Attributs de classe

- Où sont déclarés les attributs de classe ?
 - Ils sont déclarés dans une classe, en dehors des méthodes, constructeurs ou blocs, mais avec le mot-clé `static`.
- Quand sont-ils créés ?
 - Ils sont créés une seule fois quand le programme démarre, et sont supprimés à la fin du programme.
- Comment y accéder ?
 - On y accède généralement avec : `NomDeLaClasse.nomDeLaVariable`
- Combien de copies ?
 - Il n'y a qu'une seule copie de chaque attribut static, peu importe combien d'objets tu crées à partir de cette classe.

COMPRENDRE LA NOTION DE CLASSE

Méthodes d'instance

Une méthode d'instance est une méthode qui appartient à un objet (et non à la classe elle-même).

Cela veut dire que :

- Tu dois créer un objet pour pouvoir l'appeler.
- Elle peut utiliser :
 - les attributs et méthodes non statiques de l'objet,
 - mais aussi les éléments statiques de la classe.
- Quand tu appelles cette méthode, Java transmet automatiquement une référence à l'objet qui l'a appelée.
- Cette référence s'appelle `this` et te permet de dire :
 - "Je parle de cet objet précis."

Méthodes de classe

- Une méthode statique est une méthode qui appartient à la classe, et non à un objet.
- Donc :
 - Elle ne peut pas utiliser les attributs d'instance (car elle n'est liée à aucun objet).
 - Elle peut être appelée sans créer d'objet.
- Pour la créer, on utilise le mot-clé `**static**`.
- Pour l'appeler :
 - `NomDeLaClasse.nomDeLaMéthode();`

EXEMPLE DE DÉCLARATION DE LA CLASSE COMPUTER

Ici, nous déclarons simplement notre classe.

```
public class Computer { no usages

    // déclaration des propriétés de l'ordinateur
    public int ram; no usages
    public String cpu; no usages
    public String drive; no usages

    // Déclaration des actions de l'ordinateur
    public void start() {} no usages
    public void stop() {} no usages
    public int resolve(int x ,int y) { return x+y;} no usages

}
```




LE MOT CLÉ STATIC

UTILISATION ET UTILITÉ

LE MOT CLÉ STATIC

Le mot clé `static` en Java est utilisé pour indiquer que le membre d'une classe (variable, méthode, ou bloc) appartient à la classe elle-même, *plutôt qu'aux instances de cette classe*.

Examinons les différentes utilisations et les implications du mot clé `static` en Java :

1. Variables statiques :

- Une variable statique est partagée par toutes les instances de la classe.
- Elle est initialisée une seule fois, au moment où la classe est chargée en mémoire

2. Méthodes statiques :

- Une méthode statique peut être appelée sans créer une instance de la classe.
- Elle ne peut pas accéder directement aux membres non statiques (variables d'instance ou méthodes non statiques) de la classe.

3. Blocs statiques :

- Un bloc statique est utilisé pour initialiser les variables statiques.
- Il est exécuté une seule fois, lorsque la classe est chargée.

UTILITÉ DU MOT CLÉ STATIC

Tout au long de votre codage, le mot clé `static` peut jouer un rôle différent :

1. Partage de données :
 - Les variables statiques sont utilisées pour partager des données entre toutes les instances d'une classe. Par exemple, un compteur qui doit être commun à toutes les instances.
2. Méthodes utilitaires :
 - Les méthodes statiques sont souvent utilisées pour les fonctions utilitaires qui ne dépendent pas de l'état de l'instance. Par exemple, les méthodes de la classe `Math` comme `Math.sqrt()`.
3. Initialisation :
 - Les blocs statiques sont utilisés pour l'initialisation complexe des variables statiques, là où une simple initialisation inline ne suffit pas.

REPRISE DE NOTRE CLASSE COMPUTER

Ajoutons du contenu static

```
public class Computer { no usages

    // déclaration des propriétés de l'ordinateur
    public int ram; no usages
    public String cpu; no usages
    public String drive; no usages

    // declaration d'une variable statique
    public static int numberOfComputers; 2 usages

    // Déclaration d'un bloc static
    static {
        numberOfComputers=0;
    }

    // Déclaration d'une méthode statique
    public static void displayNumberInstance() { no usages
        System.out.println("Number of Computers: " + numberOfComputers);
    }

    // Déclaration des actions de l'ordinateur
    public void start() {} no usages
    public void stop() {} no usages
    public int resolve(int x ,int y) { return x+y;} no usages

}
```



NIVEAUX DE VISIBILITÉ

JE TE VOIS ! JE TE VOIS PLUS !

NIVEAUX DE VISIBILITÉ DE JAVA

- **public** : accessible de n'importe où - La classe aura accès à ce membre mais aussi n'importe quelle autre classe.
 - *Par exemple, notre méthode main est public afin que la JVM puisse l'invoquer depuis l'extérieur de la classe car il n'est pas présent dans la classe courante.*
- **protected** : un membre marqué dans une classe comme protégé peut être manipulé :
 - Dans la classe qui définit ce membre
 - Dans les classes qui dérivent de la classe considérée.
 - Dans toutes les classes définies dans le même package que celle qui définit le membre protégé.
- **package private** : mode de visibilité par défaut, en l'absence de mot clé. Un membre sera considéré comme étant en "package private". Ce membre sera visible dans tout code dans le même package.
- **private** : Dans une classe, la déclaration d'un membre en private indiquera que seule la classe pourra y accéder.
- **final** : selon le contexte,
 - Pour une variable, indique que sa valeur ne peut être changée, une fois initialisée.
 - Pour une méthode, indique que la méthode ne pourra pas être redéfinie par une classe héritant de la classe ayant définie la méthode.
 - Pour une classe, il empêche la classe d'être héritée par une autre classe.

CONTRÔLE DE L'ACCÈS AUX PROPRIÉTÉS HÉRITÉES

private

Dans la classe de base, mettre **private** rendra **inaccessible** même pour les sous-classes.

Une classe dérivée n'a pas plus de privilèges du fait de l'héritage.

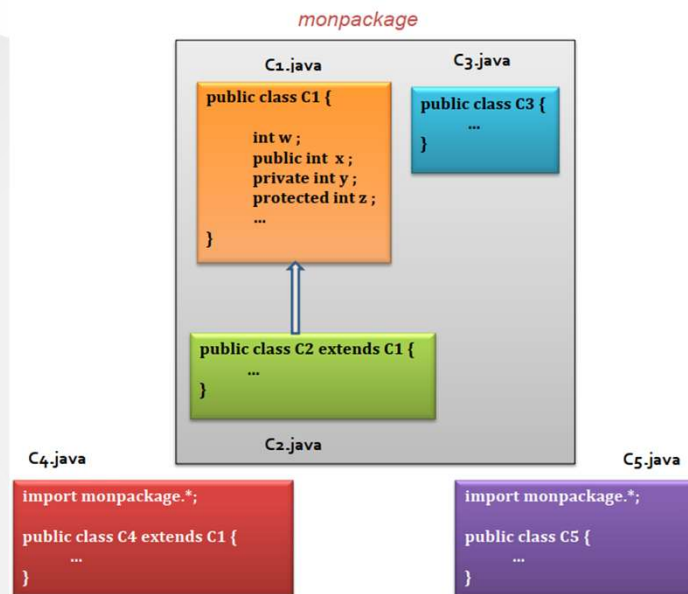
- *Par exemple, la classe mère définit ses setters en private pour interdire à la classe enfant de pouvoir redéfinir ses setters.*

protected

Protected appliqué à un attribut ou une méthode d'une classe mère permet de conserver une protection semblable à celle de **private**, tout en rendant possible l'accès à cet attribut ou cette méthode dans les classes filles ou les classes du même package.

RÉSUMÉ : NIVEAUX DE VISIBILITÉ DE JAVA

- ✓ C1, C2 et C3 sont 3 classes publiques appartenant à *monpackage*.
- ✓ Dans *monpackage*, la classe C2 hérite de C1.
- ✓ Les classes C4 et C5 importent le package *monpackage* mais n'appartiennent pas à ce package.
- ✓ La classe C4 hérite de C1.



Accès à →	accès package w	public x	private y	protected z
C1	oui	oui	oui	oui
C2	oui	oui	non	oui
C3	oui	oui	non	oui
C4	non	oui	non	oui
C5	non	oui	non	non

Le tableau résume les différents mode d'accès des membres d'une classe.

Modificateur du membre	private	aucun	protected	public
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui



ENCAPSULATION

CACHER L'IMPLÉMENTATION

LE PRINCIPE D'ENCAPSULATION

L'encapsulation est l'un des principes non négligeables de la POO.

Mais en quoi cela consiste ?

C'est relativement simple en fait : cela consiste à cacher le fonctionnement interne de votre objet en imposant à l'utilisateur de l'objet de passer par vos méthodes publiques.

Cela permet notamment 2 choses :

- La sécurisation des données de votre objet, car l'utilisateur est obligé d'utiliser les méthodes que vous lui mettez à disposition
- Réaliser des traitements internes à l'objet sans que l'utilisateur ne le sache. Car oui certains traitements n'ont pas besoin d'être visibles pour l'utilisateur de l'objet, car soit cela n'a pas d'intérêt ou parce que tout simplement il s'en fout. C'est ce qu'on appelle l'abstraction.

LE PRINCIPE D'ENCAPSULATION

GETTER - ACCESSEURS - ACCESSORS

En Java, le **getter** est une méthode utilisée pour protéger les données et faire en sorte de rendre le code sûr.

Le getter retourne une valeur du type de l'attribut concerné.

La convention veut que la méthode suive la syntaxe **public type getNomAttribut()**

- Cette méthode est **public** et l'attribut est **private**, forçant ainsi à passer par cette méthode d'accesseur pour accéder à l'attribut de l'instance.

SETTER - MUTATEURS - MUTATORS

En Java, le **setter** est une méthode utilisée pour protéger les données et faire en sorte de rendre le code sûr.

Le setter fixe et met à jour la valeur de l'attribut concerné. Il permet de mettre en place la logique de contrôle sur l'attribut.

La convention veut que la méthode suive la syntaxe **public void setNomAttribut(paramètres..)**

- Cette méthode est **public** et l'attribut **private** forçant ainsi à passer par cette méthode mutateur pour fixer ou mettre à jour l'attribut de l'instance.

MODIFICATION DE NOTRE CLASSE

- Nos attributs d'instance et de classe deviennent privés par le mot-clé **private**
- Du fait de la mise en place de l'encapsulation, la création de méthodes d'accès à nos attributs sont mis en place au travers **d'accesseurs** et **de mutateurs** (**accessors/mutators**) appelés **getter** et **setter**
- **Ce sont dans ces méthodes que nous mettrons en place toute la logique de contrôle des données dit les tests métiers.**

Donc, gardez cet automatisme : si un utilisateur veut changer un attribut de ma classe, il doit passer par une méthode de mutateurs.

```
public class Computer { 1 usage

    // déclaration des propriétés de l'ordinateur
    private int ram; 2 usages
    private String cpu; 2 usages
    private String drive; 2 usages

    // declaration d'une variable statique
    private static int numberOfComputers; 4 usages

    public String getCpu() { no usages
    |     return cpu;
    | }

    public void setCpu(String cpu) { no usages
    |     this.cpu = cpu;
    | }

    public int getRam() { no usages
    |     return ram;
    | }

    public void setRam(int ram) { no usages
    |     this.ram = ram;
    | }

    public String getDrive() { no usages
    |     return drive;
    | }
}
```

MODIFICATION DE NOTRE CLASSE

Le principe de l'abstraction est ici appliqué en indiquant certaines méthodes en **private**.

- Ici, c'est la méthode public `start()` qui appelle la méthode private `chargingBios()` qui appelle elle-même une méthode private `chargingOS()`
 - L'utilisateur de la classe `Computer` n'a pas à connaître comment est implémenté le chargement du Bios et l'OS.

```
// Déclaration des actions de l'ordinateur

private void chargingOS () { } 1 usage

private void chargingBios() { 1 usage
    chargingOS();
}

public void start() { no usages
    chargingBios();
}

public void stop() {} no usages
public int resolve(int x ,int y) { return x+y;} no usages
}
```

THIS

Le mot-clé **this** désigne dans une classe, l'instance courante de la classe elle-même.

Il est utilisé à différentes fins :

Rendre univoque :

- Rendre le code explicite et non ambigu
 - Exemple différence entre paramètre et attribut

S'auto-désigner comme référence :

- Fait référence à l'instance elle-même comme paramètre d'une méthode

Désigner l'instance de la classe qui encadre :

- Dans le cas de classes imbriquées, c'est-à-dire qu'une classe interne utilise l'instance de la classe externe, le mot-clé **this** préfixé du nom de la classe externe permet de désigner l'instance de la classe externe. S'il n'est pas préfixé, il désigne l'instance de la classe interne.

Appeler un autre constructeur de la classe :

- Un constructeur peut appeler un autre constructeur de la classe en utilisant le mot-clé **this** (par l'exemple dans le cas de l'héritage).

REPRISE DE NOTRE CLASSE COMPUTER

Intégrons le mot-clés `this`

```
// Déclaration des actions de l'ordinateur

private void chargingOS () { } 1 usage

private void chargingBios() { 1 usage
|   this.chargingOS();
| }

public void start() { no usages
|   this.chargingBios();
| }

public void stop() {} no usages
public int resolve(int x ,int y) { return x+y;} no usages
```

```
public String getCpu() { no usages
|   return this.cpu;
| }

public void setCpu(String cpu) { no usages
|   this.cpu = cpu;
| }

public int getRam() { no usages
|   return this.ram;
| }

public void setRam(int ram) { no usages
|   this.ram = ram;
| }

public String getDrive() { no usages
|   return this.drive;
| }

public void setDrive(String drive) { no usages
|   this.drive = drive;
| }

public static int getNumberOfComputers() { no usages
|   // ici je ne peux pas appliquer this
|   // ce n'est pas une instance
|   return numberOfComputers;
| }
```



LES MÉTHODES

PRINCIPE ET CARACTÉRISTIQUES

SYNTAXE D'UNE MÉTHODE

```
// en-tête de la méthode
modificateur returnType nomMethode(ListeParamètres) {
    // corps ou contenu de la méthode

    // si type de retour
    return ...
}
```

Chaque méthode doit inclure les deux parties décrites dans la syntaxe :

- L'en-tête de la méthode
- Le corps de la méthode

- L'en-tête de la méthode fournit des informations sur la manière dont les autres méthodes peuvent interagir avec elle. Une en-tête de méthode est également appelé une déclaration.
- Entre deux accolades, le corps de la méthode contient les instructions permettant d'effectuer le travail de la méthode : son implémentation.
 - Techniquement, une méthode n'est pas obligée de contenir des instructions dans son corps : un stub.
- Modificateur : le modificateur d'accès pour une méthode Java peut être l'un des mot-clés suivants : public, private, protected ou si non spécifié, package par défaut.
 - Le plus souvent, les méthodes sont accessibles en public.
- La signature d'une méthode est la combinaison du nom de la méthode et du nombre, des types et de l'ordre des arguments.
 - Un appel de méthode doit correspondre à la signature de la méthode appelée.

MÉTHODES

STATIC

C'est un mot-clé qui, lorsqu'il est associé à une méthode, en fait une méthode liée à une classe.

- La méthode `main()` est statique afin que JVM puisse l'invoquer sans instancier la classe.
- Cela évite également le gaspillage de mémoire inutile qui aurait été utilisé par l'objet déclaré uniquement pour appeler la méthode `main()` par la JVM.

VOID

C'est un mot-clé utilisé pour spécifier qu'une méthode ne retourne rien.

- Comme la méthode `main()` ne renvoie rien, son type de retour est `void`.
- Dès que la méthode `main()` se termine, le programme java se termine également.
- Par conséquent, cela n'a aucun sens de revenir de la méthode `main()` car JVM ne peut rien faire avec la valeur de retour de celle-ci.

MÉTHODES : DIFFÉRENCE ENTRE ARGUMENTS ET PARAMÈTRES

Arguments

Lorsqu'une méthode est appelée, les valeurs transmises lors de l'appel sont appelées arguments.

- Ceux-ci sont utilisés dans l'instruction d'appel de méthode pour envoyer une valeur de la méthode appelante à la méthode appelée.
- Pendant l'appel, chaque argument est toujours attribué au paramètre dans la définition de la méthode.
- Ils sont aussi appelés paramètres réels.

Paramètres

Les valeurs qui sont écrites lors de la définition de la méthode sont appelées paramètres.

- Ceux-ci sont utilisés dans l'en-tête de la méthode appelée pour recevoir la valeur des arguments.
- Les paramètres sont des variables locales auxquelles sont attribués les arguments lorsque la méthode est appelée.
- Ils sont aussi appelés paramètres formels.

PASSAGE DE PARAMÈTRES : TYPES PRIMITIFS

Considérons la fonction suivante et un appel de cette fonction :

```
static void augmenter(int i) {  
    i++;  
}  
  
public void test() {  
  
    int j=1;           // (1)  
    augmenter(j);      // (2)  
    System.out.println(j); // (3)  
  
}
```

1. Nous avons déjà vu ce que cela donne.

Lors de l'appel de la fonction, le passage de paramètre s'effectue par valeur. Cela signifie qu'une variable i locale à la fonction est créée et que l'on recopie dans l'espace mémoire associé à cette variable la valeur associée à j

2. i est incrémenté de 1

3. On sort de la fonction. La variable locale est effacée et le résultat affiché est 1 (soit la valeur de la variable j).

Conclusion : si on passe un paramètre correspondant à un type primitif à une méthode, sa valeur ne peut pas être modifiée par la méthode.

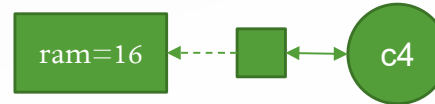
PASSAGE DE PARAMÈTRES : TYPES RÉFÉRENCES

Prenons l'exemple ci-dessous :

```
// Création d'une instance de Computer
Computer c4 = new Computer();
System.out.println(c4.getRam());
// mise à jour de la valeur de la RAM de c4
c4.setRam(16);
// appel d'une méthode de classe en donnant c4 comme paramètre
Computer.setupRam(c4);
// affichage
System.out.println(c4.getRam());
```

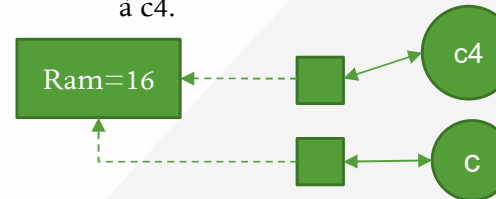
```
public static void setupRam(Computer c) { no usages
    c.setRam(c.getRam()+10);
}
```

1. Création et instantiation de l'objet p1

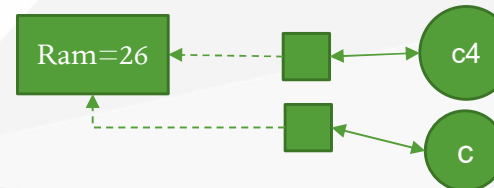


2. Lors de l'appel de la méthode, le passage de paramètres s'effectue par valeur **mais cette fois-ci, c'est la référence qui est passée par valeur.**

1. Une variable locale c est déclarée et la valeur de la référence correspond à c4.



3. Dans la méthode, ram est augmentée de 10.



4. A la sortie de la méthode, la variable locale est effacée et le résultat affiché est 26.


Conclusion : Si on passe un paramètre correspondant à une référence à une méthode, l'objet référencé peut être modifié par la méthode.

MÉTHODES : SURCHARGE


La surcharge permet à différentes méthodes d'avoir le même nom, mais des signatures différentes où la signature peut différer en fonction du nombre de paramètres d'entrée, du type de paramètres d'entrée ou des deux.

Quelques règles :

1. On ne peut pas surcharger par type de retour.
2. On peut surcharger des méthodes statiques.
3. On ne peut pas surcharger la méthode main.



```
public class Calculatrice {  
  
    // Méthode pour additionner deux entiers  
    public int additionner(int a, int b) {  
        return a + b;  
    }  
  
    // Méthode pour additionner trois entiers  
    public int additionner(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    // Méthode pour additionner deux nombres à virgule flottante  
    public double additionner(double a, double b) {  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        Calculatrice calc = new Calculatrice();  
  
        System.out.println("Addition de deux entiers : " + calc.additionner(5, 10));  
        System.out.println("Addition de trois entiers : " + calc.additionner(5, 10, 15));  
        System.out.println("Addition de deux doubles : " + calc.additionner(5.5, 10.5));  
    }  
}
```



```
public class Exemple {  
    // Ces méthodes ne peuvent pas coexister dans la même classe  
    public int maMethode(int a, int b) {  
        return a + b;  
    }  
  
    // Cette méthode n'est pas autorisée car elle ne diffère que par le type de retour  
    public double maMethode(int a, int b) {  
        return a + b;  
    }  
}
```



LES CONSTRUCTEURS

CYCLE DE VIE D'UN OBJET

RÔLE DU CONSTRUCTEUR

Il est possible de déclarer des méthodes particulières dans une classe que l'on nomme **constructeurs**.

Un constructeur a pour objectif d'initialiser un objet nouvellement créé afin de garantir qu'il est dans un état cohérent avant d'être utilisé.

Un constructeur a la signature suivante :

```
public class MaClasse {  
    // Signature du constructeur  
    public MaClasse(TypeParam1 nomParam1, TypeParam2 nomParam2) {  
        // Corps du constructeur  
    }  
}
```

- Un constructeur se distingue d'une méthode car il n'a jamais de type de retour (pas même **void**). De plus un constructeur a obligatoirement **le même nom que la classe**.

Lorsqu'un **Computer** est créé par l'application avec l'opérateur **new** comme avec l'instruction suivante :

```
// Création d'une instance de Computer  
Computer c4 = new Computer();
```

- Alors, la JVM crée l'espace mémoire nécessaire pour le nouvel objet de type **Computer**, puis elle appelle le constructeur et enfin elle assigne la référence de l'objet à la variable **computer**.

Donc le constructeur permet de réaliser une **initialisation complète** de l'objet selon les besoins des développeurs.

LES CONSTRUCTEURS

- Chaque classe a un constructeur.
 - Si nous n'écrivons pas explicitement un constructeur pour une classe, le compilateur Java construit **un constructeur par défaut** pour cette classe.
- Chaque fois qu'un nouvel objet est créé, au moins un constructeur sera appelé.
- La règle principale des constructeurs, c'est qu'ils doivent avoir le même nom que la classe.
- Une classe peut avoir plusieurs constructeurs. (**surcharge**).
- Si votre classe ne contient qu'un seul constructeur sans paramètre dont le corps est vide, alors vous pouvez supprimer cette déclaration car le compilateur le générera automatiquement.

AJOUT DU CONSTRUCTEUR À NOTRE CLASSE

Ajout des constructeurs symbolisant la surcharge des méthodes dans la classe.

- Ici deux constructeurs ayant une signature différente.
- A noter que les constructeurs ne retournent pas de type d'où la non-présence du mot-clé void

C'est également dans le constructeur que nous allons gérer notre attribut de classe nombreOrdinateurs.

```
// Déclaration des constructeurs
// le constructeur par défaut
public Computer() { no usages
    numberOfComputers++;
}

// le constructeur avec toutes les propriétés
public Computer(int ram, String cpu, String drive) { no usages
    this.ram = ram;
    this.cpu = cpu;
    this.drive = drive;
    numberOfComputers++;
}

// un constructeur avec seulement 2 propriétés
public Computer(int ram, String cpu) { no usages
    this.ram = ram;
    this.cpu = cpu;
    this.drive = null;
    numberOfComputers++;
}
```

LES CONSTRUCTEURS

Constructeur privé

Il est tout à fait possible d'interdire l'instantiation d'une classe en Java.

Pour cela, il suffit de déclarer tous ses constructeurs avec une portée `private`.

Un cas d'usage courant est la création d'une classe outil :

- Une classe outil ne contient que des méthodes de classe.
- Il n'y a donc aucun intérêt à instancier une telle classe. Donc, on déclare un constructeur privé pour éviter une utilisation incorrecte.

Appel d'un constructeur dans un constructeur

Certaines classes peuvent offrir différents constructeurs à ses utilisateurs.

Souvent ces constructeurs vont partiellement exécuter le même code.

Pour simplifier la lecture et éviter la duplication de code, un constructeur peut appeler un autre constructeur en utilisant le mot-clé `this` comme nom du constructeur notamment lors de l'héritage.

Cependant, un constructeur ne peut appeler qu'un seul constructeur et, s'il le fait, cela doit être sa première instruction.

LES CONSTRUCTEURS

Appel d'une méthode dans un constructeur

Il est tout à fait possible d'appeler une méthode de l'objet dans un constructeur.

Cela est même très utile pour éviter la duplication de code et favoriser la réutilisation.

Attention cependant au statut particulier des constructeurs. Tant qu'un constructeur n'a pas achevé son exécution, l'objet n'est pas totalement initialisé.

La méthode finalize

Si un objet souhaite effectuer un traitement avant sa destruction, on peut implémenter la méthode `finalize`.

Cette méthode a la signature suivante :

```
protected void finalize() {  
}
```

Notes : Dans la pratique cette méthode n'est utilisée que pour des cas d'implémentation très avancés. Mais il est déconseillé de surcharger la méthode mère. Nous laissons le Garbage-Collector faire son travail



UTILISATION DE LA CLASSE

EXEMPLE

Pour utiliser la classe, il suffit donc d'utiliser les constructeurs.

1. Il construit une instance sur la définition de la classe
2. A partir de cette instance, nous avons dès lors accès à toutes les méthodes d'instance.

Notes :

1. Depuis l'instance, nous ne pouvons pas accéder aux propriétés qu'à partir des setters et getters.
2. Depuis la classe, nous avons accès seulement aux propriétés et méthodes static

```
// code...
System.out.println("Hello World!");

// Création des instances
Computer computer = new Computer( ram: 64, cpu: "Intell", drive: "500Gb");
Computer computer2 = new Computer( ram: 64, cpu: "Intell");
Computer computer3 = new Computer();
// affichage
System.out.println(computer.getCpu());
// modification d'une propriété
computer.setCpu("AMD");
System.out.println(computer.getCpu());
// Affichage du nombre d'instance
System.out.println(Computer.getNumberOfComputers());
```

AGRÉGATION ET COMPOSITION

AGRÉGATION ET COMPOSITION

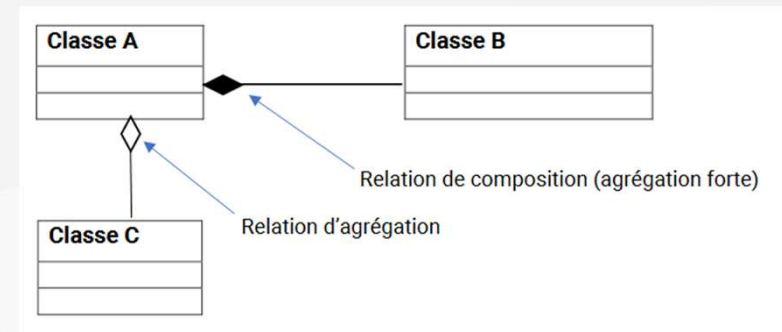
Le type d'association, entre classes, a un impact majeur lors de l'implémentation, c'est-à-dire lors de la programmation.

Au moment de l'élaboration du diagramme de classes, l'identification du type d'association est un travail qui est rigoureux et qui demande une analyse approfondie.

En effet, l'agrégation est un cas particulier d'association exprimant une relation de contenance.

L'agrégation exprime le fait qu'une classe est composée d'une ou plusieurs autres classes. Il existe deux types d'agrégation.

1. Agrégation faible
2. Agrégation forte appelée Composition.



L'agrégation est modélisée par un trait et à l'extrémité un losange vide. La composition est modélisée en UML par un trait et un losange noir plein à l'extrémité

AGRÉGATION

Soit l'exemple suivant modélisant un produit appartenant à une catégorie.



Un produit appartient à une seule catégorie et une catégorie peut avoir un ou plusieurs produits.

Cette association peut être lue de cette façon : **un produit fait partie d'une catégorie**. Aussi, cette association peut être lue de cette façon : **Une catégorie est composée d'un ou plusieurs produits**.

Lorsqu'il s'agit d'association « **fait partie** » ou « **est composé de** » ou « **avoir un** », nous parlons d'agrégation.

Ce type d'association est modélisé par un trait et à l'extrémité un losange à côté du conteneur ou du composite.



Ici, le composite est la classe Catégorie et le composant est la classe Produit.

COMPOSITION

Une composition est une agrégation forte. La question qui se pose est que signifie agrégation forte.

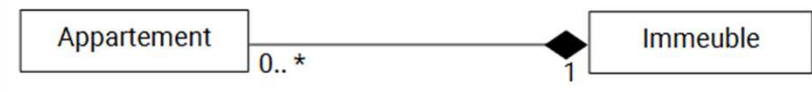
Est-ce que la suppression d'une catégorie élimine automatiquement ces produits ?

NON ! Nous pouvons conclure que l'agrégation entre la classe Catégorie et la classe Produit n'est pas une agrégation forte, ce n'est pas une composition.

Une association de type composition se traduit par la suppression d'un objet composite élimine automatiquement tous les objets composants.

Ce qui exprime une agrégation forte.

Soit l'exemple suivant, modélisant un appartement appartient à un immeuble.



Ici, il s'agit d'une agrégation forte ou ce qu'on appelle **Composition**.

Pourquoi ?

La destruction de l'immeuble entraîne automatiquement la destruction des appartements associés.

La modélisation d'une composition est exprimée par un losange plein côté le composé.

Dans notre exemple, le composé (composite) est la classe Immeuble et le composant est la classe Appartement.



L'HÉRITAGE

PLUS DE RICHESSES DANS JAVA !

HÉRITAGE AGRÉGATION FAIBLE COMPOSITION AGRÉGATION FORTE

Est-un (is-a)

Cette relation permet de créer une chaîne de relation d'identité entre des classes.

Elle indique qu'une classe peut être assimilée à une autre classe qui correspond à une notion plus abstraite ou plus générale.

On parle d'héritage pour désigner le mécanisme qui permet d'implémenter ce type de relation.

Superclasse = classe de base = Mère

Sous-classe = classe dérivé = Enfant

- La classe A est un enfant de la classe Mère

A un (has-a)

Cette relation permet de créer une relation de dépendance d'une classe envers une autre.

Une classe a besoin des services d'une autre classe pour réaliser sa fonction.

On parle également de relation de composition pour désigner ce type de relation.

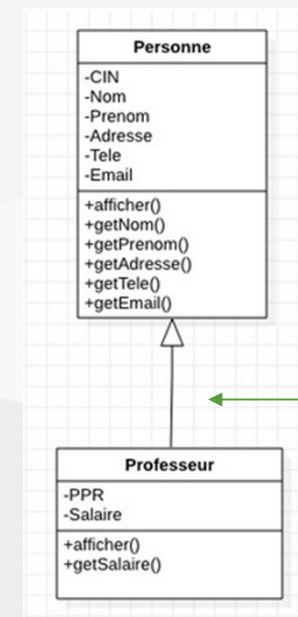
L'HÉRITAGE (IS-A)

Principe

En Java et dans tous les langages orientés objet, l'héritage est un mécanisme qui permet à une classe d'acquérir tous les comportements et attributs d'une autre classe, ce qui signifie que vous pouvez créer une nouvelle classe simplement en indiquant en quoi elle diffère d'une classe qui a déjà été développée et testée.

Lorsque vous créez une classe en la faisant hériter d'une autre classe, la nouvelle classe contient automatiquement les champs de données et les méthodes de la classe d'origine.

Exemple d'héritage



A noter : le sens de la flèche.
Une classe fille connaît sa classe mère mais pas l'inverse

LA COMPOSITION (HAS-A)

La situation “a un” décrit la composition.

- Par exemple, **Entreprise** contenant un tableau d'objets **Departements**.

- un département **est une** entreprise



- une entreprise **a des** départements



- Par conséquent, cette relation n'est pas un héritage ; c'est une forme de confinement appelée **composition** - relation dans laquelle une classe contient un ou plusieurs membres d'une autre classe, lorsque ces membres ne continueraient pas d'exister sans l'objet qui les contient.

- De même, chaque objet **Departement** peut contenir un tableau d'objets **Employe**.

- un employé **est un** département



- un département **a des** employés.



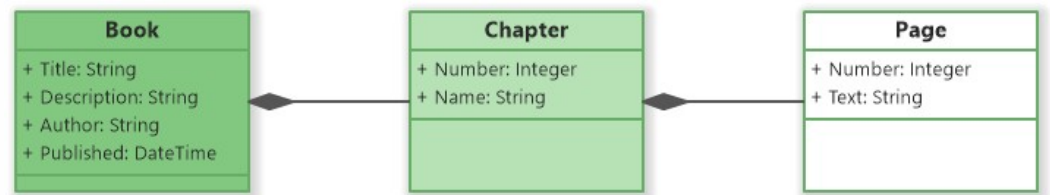
- Cette relation n'est pas non plus un héritage; il s'agit d'un type spécifique de confinement appelé **agrégation**: relation dans laquelle une classe contient un ou plusieurs membres d'une autre classe, lorsque ces membres continueraient d'exister sans l'objet qui les contient.

LA COMPOSITION (HAS-A)

La composition est le type de relation le plus souvent utilisé en programmation objet. Elle indique une dépendance entre deux classes.

L'une a besoin des services d'une autre pour réaliser sa fonction.

La composition se fait en déclarant des attributs dans la classe.



RÈGLES D'HÉRITAGE EN JAVA

Spécifique à Java :

- Une classe fille ne peut hériter que d'une seule classe mère : **Héritage simple**.
- Plusieurs classes filles peuvent hériter d'une même classe mère.
- Une classe fille peut elle-même être la classe mère d'une nouvelle classe. Il n'y a pas de limites dans les niveaux.
- On peut donc affirmer qu'en Java, toute instance de classe, quelle que soit sa classe d'appartenance est de type Object, en plus d'être du type de la classe avec laquelle elle a été instanciée.



LE CAST INDUIT PAR L'HÉRITAGE

En POO, une sous-classe peut être considéré comme une instance de la classe mère :

```
ClasseMere objMere = new ClasseFille();
```



L'inverse n'est pas vrai :

```
ClasseFille objFille = new ClasseMere();
```



- L'héritage définit un cast implicite de ClasseFille vers ClasseMere

Dans l'exemple suivant, on part du principe que methodeRedef() existe dans la classe Mère et la classe Fille

```
ClasseMere objMere = new ClasseFille();
objMere.methodeMere();
objMere.methodeRedef(); // Quelle méthode va être appelée ?
```

- Java considère toujours qu'il doit tenir compte de l'objet désigné par la référence : en conséquence, c'est bien la méthode de ClasseFille qui va être appelée. Ce comportement est logique : il est raisonnable que le traitement que l'on souhaite voir s'exécuter soit bien celui de l'objet désigné.

Mais :

```
ClasseMere objMere = new ClasseFille() ;
// Erreur de compilation
objMere.methodeFille() ;
// OK, le cast « rassure » le compilateur
( ( ClasseFille ) objMere ).methodeFille() ;
```

- Donc ici, les seules méthodes appelables directement sont celles de ClasseMere. Parmi ces méthodes, certaines peuvent avoir été redéfinies, auquel cas, ce sont les méthodes redéfinies qui seront appelées si l'on pointe sur une instance de la classe dérivée.



UTILISATION DE L'HÉRITAGE

PAR L'EXEMPLE

Imaginons que nous souhaitons créer un jeu MMORPG.

Dans notre désir de mettre en place, il nous faut créer nos différentes classes à jouer :

- Un guerrier
- Un voleur
- Un soigneur

On vous demande de mettre en place la structure :

1. Créer les 3 classes du jeu ?
2. Que constatons nous ?
3. Améliorez notre architecture de classes ?

Chaque classe vont posséder des caractéristiques soit :

- Son Nom
- Sa Race
- Son Niveau
- Ses Points de vie
- Ses Points de magie
- Ses Points de Force
- Ses Points d'habilité
- Sa classe
 - Le nom de sa classe
 - L'arme de sa classe
 - Le type d'armure de sa classe

L'HÉRITAGE EN JAVA : EXTENDS ET INSTANCEOF

```
Public class Professeur extends Personne {
// code
}
```

Chaque Professeur reçoit automatiquement les attributs et les méthodes du superclasse Personne

Vous ajoutez ensuite de nouveaux attributs et méthodes à la sous-classe nouvellement créée.

```
Professeur prof = new Professeur();
```

L'objet prof a accès à toutes les méthodes de la classe Personne, ainsi qu'aux méthodes de sa propre classe.

Vous pouvez utiliser l'opérateur instanceof pour déterminer si un objet est un membre ou un descendant d'une classe.

Si prof est un objet de Professeur :

- prof instanceof Professeur ➔ true
- prof instanceof Personne ➔ true

Si p1 est un objet Personne :

- p1 instanceof Personne ➔ true
- p1 instanceof Professeur ➔ false

APPEL DE CONSTRUCTEURS PENDANT L'HÉRITAGE

Ce qui se passe chez l'enfant

Lorsque vous instanciez un objet membre d'une sous-classe, vous appelez à la fois le constructeur de la superclasse et le constructeur de la sous-classe. Lorsque vous créez un objet de sous-classe, le constructeur de la superclasse doit d'abord s'exécuter, puis le constructeur de la sous-classe est exécuté.

Constructeur par défaut

Si aucune définition de constructeur alors Java fournit et utilise le constructeur par défaut qui ne nécessite pas d'arguments.

Lorsqu'on définit un constructeur (qu'il soit par défaut ou avec des paramètres), alors Java utilise votre constructeur.

Si une superclasse contient :	Alors ses sous-classes :
Aucun constructeur définit	Ne nécessite pas de constructeurs
Un constructeur par défaut définit	Ne nécessite pas de constructeurs
Seuls les constructeurs autres que ceux par défaut	Doit contenir un constructeur qui appelle celui de la superclasse

APPEL DE CONSTRUCTEURS PENDANT L'HÉRITAGE

`super(liste des paramètres)`

Le mot-clé `super` fait toujours référence à la super-classe de la classe dans laquelle vous l'utilisez.

Remarque ! L'instruction `super()` doit être la première instruction de tout constructeur de sous-classe qui l'utilise. Même les définitions des attributs ne peuvent la précéder. On construit d'abord l'ascendant avant de construire l'objet.

Exemple :

```
class Personne {
    String nom;
    String cin;

    public Personne(String nom, String cin) {
        this.nom = nom;
        this.cin = cin;
    }

    public void afficher() {
        System.out.println("Nom : " + nom + " - cin : " + cin);
    }
}

public class Professeur extends Personne {
    int PPR;
    double salaire;

    public Professeur(String nom, String cin, int ppr, double salaire) {
        super(nom, cin);
        this.PPR = ppr;
        this.salaire = salaire;
    }
}
```

Constructeur de la Mère

Constructeur de l'enfant avec appel du constructeur de la Mère



POO : COMPRENDRE LA SURCHARGE ET LA REDÉFINITION DE MÉTHODE EN JAVA

REDÉFINITION DE MÉTHODE

La programmation orientée objet repose sur le concept d'héritage, qui permet aux classes de dériver de classes existantes.

L'héritage permet aux classes de partager des propriétés et des méthodes communes, tout en offrant la possibilité de personnaliser et d'ajouter des fonctionnalités spécifiques à chaque classe.

Pour exploiter cette flexibilité et cette réutilisabilité, les développeurs ont recours à des concepts clés tels que [la redéfinition de méthode](#), [la surcharge](#) et [le polymorphisme](#).

La redéfinition de méthode, également appelée polymorphisme par substitution, se produit lorsque les sous-classes redéfinissent les méthodes héritées de leur classe parente. Cela permet aux sous-classes de personnaliser le comportement des méthodes héritées pour répondre à leurs propres besoins.

Prenons l'exemple d'une classe Voiture qui a une méthode "vitesseMaximale"

```
public class Voiture {  
  
    public void vitesseMaximale() {  
        System.out.println("Vitesse max voiture = 200 km/h");  
    }  
  
}
```

Maintenant, supposons que nous ayons une classe Enfant qui hérite de la classe Voiture et souhaite redéfinir la méthode "vitesseMaximale" pour sa propre utilisation.

```
public class Enfant extends Voiture {  
  
    @Override  
    public void vitesseMaximale() {  
        System.out.println("Vitesse max voiture enfant = 20 km/h");  
    }  
  
}
```


SURCHAGE DE MÉTHODE

La surcharge de méthode se produit lorsqu'une classe a plusieurs méthodes avec le même nom, mais des paramètres différents.

Cela permet à une classe de traiter différents types de données et de fournir une interface commune pour les méthodes qui ont des fonctionnalités similaires.

Prenons l'exemple d'une classe Voiture qui a une méthode "vitesseMaximale" surchargée pour accepter un argument de type String :

```
public class Voiture {  
  
    public void vitesseMaximale() {  
        System.out.println("La vitesse maximale de la voiture est de 200 km/h");  
    }  
  
    public void vitesseMaximale(String conducteur) {  
        System.out.println("La vitesse maximale de la voiture pour " + conducteur + " est de 250 km/h");  
    }  
}
```

Dans cet exemple, la classe Voiture a deux méthodes avec le même nom "vitesseMaximale", mais la deuxième méthode prend un argument de type String qui représente le nom du conducteur.

Lorsqu'un objet de la classe Voiture est créé et que la méthode "vitesseMaximale" est appelée avec un argument, la méthode surchargée sera exécutée plutôt que la méthode originale qui ne prend pas d'argument.



POLYMORPHISME

RÉ-ÉCRITURE DES MÉTHODES

L'HÉRITAGE EN JAVA : POLYMORPHISME

Redéfinition des méthodes de la Mère

Lorsque vous créez une sous-classe en étendant une classe existante, la nouvelle sous-classe contient des données et des méthodes définies dans la superclasse d'origine.

Parfois, cependant, les attributs et les méthodes de la superclasse ne conviennent pas entièrement aux objets de la sous-classe.

Dans ces cas, vous souhaitez redéfinir les membres de la classe mère.

Polymorphisme

Utiliser le même nom de méthode pour indiquer différentes implémentations est appelé **polymorphisme** : **Même signature mais implémentation différente.**

Lorsqu'on souhaite redéfinir une méthode de classe mère dans une classe enfant, on peut insérer l'annotation **@Override** juste avant la signature de la méthode dans la classe fille.

Cela force le compilateur à émettre un message d'erreur si cela n'est pas fait !

Les trois types de méthodes que vous ne pouvez pas redéfinir dans une sous-classe sont :

- Méthodes statiques
- Méthodes finales
- Méthodes dans les classes finales

POLYMORPHISME PAR L'EXEMPLE

Le polymorphisme se réfère à la capacité des objets d'une classe à prendre plusieurs formes.

En Java, le polymorphisme est réalisé grâce à la redéfinition de méthode et à l'utilisation d'une référence de type de classe parente pour faire référence à une classe fille.

Prenons l'exemple d'une classe `Animal` qui a une méthode "crier"

```
public class Animal {  
  
    public void crier() {  
        System.out.println("L'animal crie");  
    }  
  
}
```

Maintenant, supposons que nous ayons deux classes qui héritent de la classe `Animal`, la classe `Chien` et la classe `Chat` :

```
public class Chien extends Animal {  
  
    @Override  
    public void crier() {  
        System.out.println("Le chien aboie");  
    }  
  
}
```

```
public class Chat extends Animal {  
  
    @Override  
    public void crier() {  
        System.out.println("Le chat miaule");  
    }  
  
}
```

Dans cet exemple, les classes `Chien` et `Chat` ont redéfini la méthode "crier" héritée de la classe `Animal` pour représenter les sons qu'ils font.

POLYMORPHISME PAR L'EXEMPLE

Maintenant, supposons que nous ayons une méthode "faireCrier" dans une autre classe qui prend un objet de la classe Animal en paramètre et appelle sa méthode "crier" :

```
public class RefugedeAnimaux {  
  
    public void faireCrier(Animal animal) {  
        animal.crier();  
    }  
  
}
```

Maintenant, si nous créons un objet Chien et un objet Chat et appelons la méthode "faireCrier" sur chaque objet, la méthode "crier" correspondante de chaque objet sera exécutée :

```
public static void main(String[] args) {  
  
    RefugedeAnimaux refugedeAnimaux = new RefugedeAnimaux();  
    Chien chien = new Chien();  
    Chat chat = new Chat();  
  
    refugedeAnimaux.faireCrier(chien); // affiche "Le chien aboie"  
    refugedeAnimaux.faireCrier(chat); // affiche "Le chat miaule"  
  
}
```

Dans cet exemple, l'utilisation de la référence de type de classe parente "Animal" nous permet d'appeler la méthode "crier" de la sous-classe appropriée en fonction de l'objet passé en paramètre.



LA CLASSE OBJECT

LA CLASSE À L'ORIGINE DE TOUT CHOSES

LA CLASSE OBJECT

Java est un langage qui ne supporte que l'héritage simple. L'arborescence d'héritage est un arbre dont la racine est la classe Object.

Si le développeur ne précise pas de classe parente dans la déclaration d'une classe, alors la classe hérite implicitement de Object.

La classe Object fournit des méthodes communes à toutes les classes.

Certaines de ces méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement.

La méthode equals

- L'implémentation par défaut de equals fournie par Object compare les références entre elles. Si la simple égalité de référence ne suffit pas, il faut alors redéfinir la méthode.
- *Dans certaines classes, cette méthode est déjà redéfinie : pour la classe String par exemple.*

La méthode hashCode

- La méthode hashCode est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation de table de hachage. (Utilisation très technique)

La méthode toString

- C'est une méthode très utile, notamment pour le débogage et la production de log. Elle permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Elle est implicitement appelée par le compilateur lorsqu'on concatène une chaîne de caractères avec un objet. Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.

REDÉFINITION DE LA MÉTHODE EQUALS

L'implémentation par défaut de `equals` fournie par `Object` compare les références entre elles. L'implémentation par défaut est donc simplement :

```
public boolean equals(Object obj) { new *  
    return (this == obj);  
}
```

Parfois, l'implémentation par défaut peut suffire mais si on souhaite tester de l'égalité d'un objet au-delà de la notion de référence, il faut redéfinir la méthode `equals`.

L'implémentation de `equals` doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement :

- Son implémentation doit être réflexive :
 - Pour `x` non nul, `x.equals(x)` doit être vrai
- Son implémentation doit être symétrique :
 - Si `x.equals(y)` est vrai alors `y.equals(x)` doit être vrai
- Son implémentation doit être transitive :
 - Pour `x`, `y` et `z` non nuls
 - Si `x.equals(y)` est vrai
 - Et si `y.equals(z)` est vrai
 - Alors `x.equals(z)` doit être vrai
- Son implémentation doit être consistante
 - Pour `x` et `y` non nuls
 - Si `x.equals(y)` est vrai alors il doit rester vrai tant que l'état de `x` et de `y` est inchangé.
- Si `x` est non nul alors `x.equals(null)` doit être faux.

LA CLASSE OBJECT

La méthode finalize

La méthode finalize est appelée par le ramasse-miettes avant que l'objet ne soit supprimé et la mémoire récupérée.

Redéfinir cette méthode donne donc l'opportunité au développeur de déclencher un traitement avant que l'objet ne disparaisse.

La méthode clone

La méthode clone est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet.

Par défaut, elle est déclarée protected car toutes les classes ne désirent pas permettre de cloner une instance.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur Cloneable.

ATTENTION : il effectue un clonage simple et non en profondeur donc si l'objet à cloner contient des références à des objets alors les attributs de ces objets ne seront pas clonés.

LA CLASSE OBJECT

La méthode getClass

La méthode `getClass` permet d'accéder à l'objet représentant la classe de l'instance.

Cela signifie qu'un programme Java peut accéder par programmation à la définition de la classe d'une instance.

Cette méthode est notamment très utilisée dans des usages avancés impliquant la réflexivité.

Les méthodes de concurrence

La classe `Object` fournit un ensemble de méthodes qui sont utilisées pour l'échange de signaux (thread) dans la programmation concurrente.

Il s'agit des méthodes `notify`, `notifyAll` et `wait`.