




Formation

Initiation à la P.O.O
avec java



Réalisé par Mr Jacquot David

Fait le mardi 7 mai 2019

Tables des matières

1	Introduction à la programmation orientée objet (POO)	5
1.1	L'objet	5
1.2	La classe.....	5
2	Les trois fondamentaux de la POO	6
2.1	L'encapsulation.....	6
2.2	L'héritage	6
2.3	Le polymorphisme	7
3	Les classes (class).....	8
3.1	Déclaration de la classe	9
3.2	Les modificateurs d'accès (général).....	10
4	Visibilité	10
4.1	Le principe de la visibilité.....	10
4.2	Champs et méthodes publics	10
4.3	Champs et méthodes privés.....	10
4.4	Champs et méthodes protégés	10
5	La notion d'objet	11
5.1	Instanciation d'un objet.....	11
5.2	La durée de vie d'un objet	11
5.3	L'objet null.....	11
5.4	La comparaison d'objets.....	11
5.5	La variable this.....	12
5.6	Déclarations des propriétés	12
5.6.1	Les variables d'instances.....	12
5.6.2	Les variables de classes	13
5.6.3	Les constantes.....	13
5.7	Déclarations des méthodes	14
5.7.1	Les modificateurs des méthodes.....	14
5.7.2	L'instruction return	15
5.8	Utilisation d'une méthode	15
5.9	La surcharge d'une méthode.....	16
5.10	Le constructeur	17
5.11	Valeur par défaut pour les propriétés	17
5.12	Le destructeur.....	18
5.13	L'accesseur.....	18
5.14	Comparer deux objets	19
6	L'héritage.....	22
6.1	Quand doit-on mettre en place l'héritage ?	22
6.2	L'héritage d'une classe.....	22

6.3	Pour une classe mère.....	22
6.4	Pour une classe fille	23
7	L'instruction super	24
7.1	Utiliser une variable de la classe mère.....	24
7.2	Utiliser une méthode de la classe mère	24
7.3	Appeler le constructeur de la classe mère	24
8	L'interface	26
8.1	Déclaration d'une interface.....	26
9	Les classe Internes	27
10	Les classes locales.....	27
10.1	Les classes anonymes	27
11	La classe abstraite.....	28
11.1	Les méthodes abstraites	28

1 Introduction à la programmation orientée objet (POO)

La programmation orientée objet repose sur plusieurs principes :

- ✓ l'objet
- ✓ l'encapsulation
- ✓ la classe
- ✓ l'héritage
- ✓ le polymorphisme

1.1 L'objet

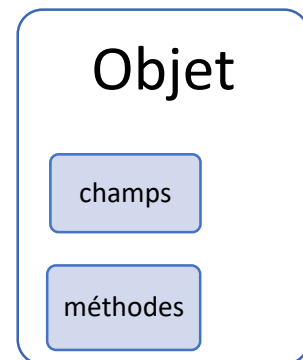
Un objet est une **structure de données**. Il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. L'objet **regroupe les données et les moyens de traitement de ces données**.

Un objet rassemble de fait deux éléments de la programmation procédurale.

- **Les champs :**
Les champs sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un champ peut posséder un type quelconque défini au préalable : nombre, caractère... ou même un type objet.
- **Les méthodes :**
Les méthodes sont les éléments d'un objet qui servent d'interface entre les données et le programme. Ce sont des procédures ou fonctions destinées à traiter les données.

Les champs et les méthodes d'un objet sont ses membres.

L'objet en lui-même est une **instance de classe**, plus simplement un exemplaire d'une classe, sa représentation en mémoire.



1.2 La classe

La classe regroupe des membres, méthodes et propriétés (attributs) communs à un ensemble d'objets.

La classe déclare, d'une part, des attributs représentant l'état des objets et, d'autre part, des méthodes représentant leur comportement.

Une classe représente donc une catégorie d'objets

2 Les trois fondamentaux de la POO

2.1 L'encapsulation

L'encapsulation permet de :

- Réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.
- Masquer l'implémentation d'un objet à un développeur extérieur et donc l'ensemble des procédures et fonctions destinées à la gestion interne de l'objet.
- Masquer un certain nombre de champs et méthodes tout en laissant visibles d'autres champs et méthodes.

Cela permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

2.2 L'héritage

L'héritage est un mécanisme parent/enfants, c'est une hiérarchie des classes, qui lie entre elles les classes composant un programme orienté objet. On appelle sous-classes les descendants d'une classe.

Une sous-classe hérite par défaut des mêmes variables et méthodes que la classe parent. Il est possible de définir de nouvelles variables ou méthodes ou de remplacer les éléments hérités par des éléments propres, de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter.

Il est possible de définir des classes abstraites. Ces classes ne sont pas destinées à être instanciées mais à fournir des éléments à hériter. Elles permettent aussi de factoriser du code.

Exemple d'un héritage :

Une classe parent Batiment avec comme attributs :

- les murs
- le toit
- une porte
- l'adresse
- la superficie

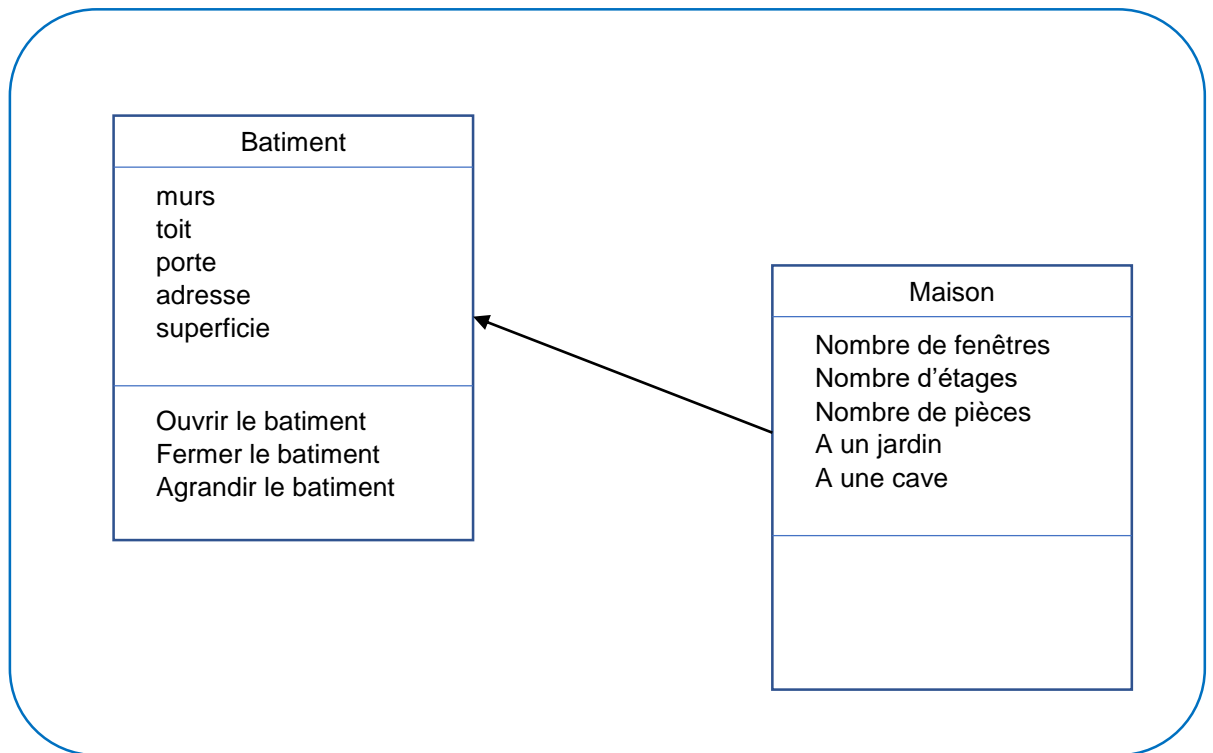
Et comme méthodes :

- ouvrir le Bâtiment
- fermer le Bâtiment
- agrandir le Bâtiment

Une sous-classe Maison qui hérite de Batiment avec comme attributs

- nombre de fenêtres ;
- nombre d'étages ;
- nombre de pièces ;
- possède ou non un jardin ;
- possède une cave.

Il est donc possible de créer un objet Maison. Ce nouvel objet est toujours considéré comme un Batiment, il possède donc toujours des murs, un toit, une porte, les champs Adresse ou Superficie et les méthodes destinées par exemple à Ouvrir le Bâtiment.



2.3 Le polymorphisme

Polymorphisme = poly comme plusieurs et morphisme comme forme

Un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter. C'est le concept de polymorphisme : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.

Le polymorphisme est la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet Véhicule et ses descendants Bateau, Avion, Voiture possédant tous une méthode avancer(), le système appellera la fonction avancer() spécifique suivant que le véhicule est un Bateau, un Avion ou bien une Voiture.

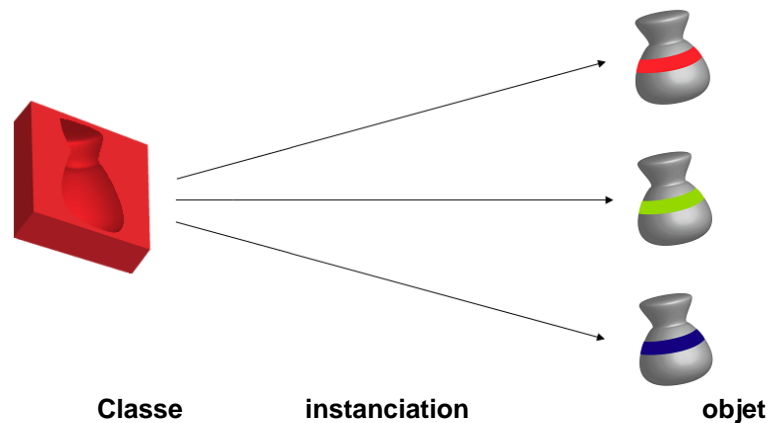
3 Les classes (class)

La classe est la structure de base de la Programmation Orienté Objet (POO).

Java nous permet d'utiliser des **objets**, entités regroupant des **propriétés** (données) et des **méthodes** (fonctions) au sein d'une **classe**, on parle d'**encapsulation**, rendant la programmation plus réutilisable et maintenable qu'en programmation classique (procédurale).

On appelle **classe** la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc "issu" d'une classe, c'est le produit qui sort d'un modèle. En réalité on dit qu'un objet est une instantiation d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'**objet** ou d'**instance** (éventuellement d'**occurrence**).

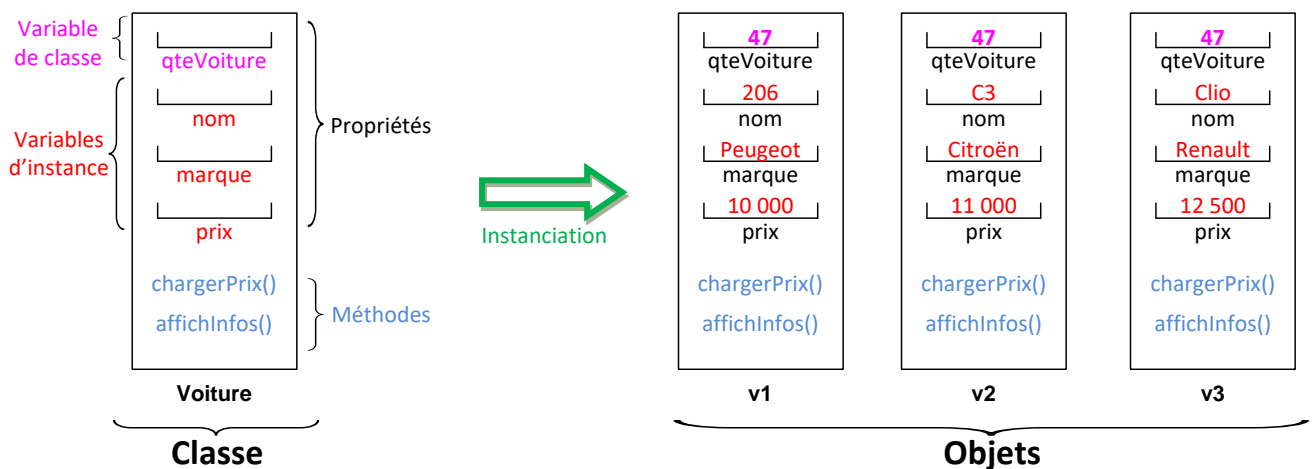
Une classe a en général, un nom qui commence par une **majuscule**.



Dans Java, tout appartient à une classe, sauf les variables de type primitif.

Une classe se compose

- Les **propriétés** (**données d'instance** ou **données membres**, car déclarées à l'intérieur de la classe), il s'agit des données représentant l'état de l'objet et/ou de classes.
- Les **méthodes** (**fonctions membres**, car déclarées à l'intérieur de la classe), il s'agit d'une collection de méthodes servant à manipuler les propriétés.



Les propriétés et les méthodes sont pourvues de **modificateurs d'accès**, aussi appelés **attributs de visibilité** qui gèrent leur accessibilité par les composants hors classes, ce qui signifie, que les propriétés et les méthodes ne sont pas toujours utilisables en tous points du programme.

3.1 Déclaration de la classe

Modificateur de classe	Description
abstract	La classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles .
private	La classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

- Les modificateurs **abstract** - **final**, et **public** - **private** sont **mutuellement exclusifs**.
- Le mot clé **extends** permet de spécifier une superclasse éventuelle, ce mot clé permet de préciser la classe mère dans une relation d'héritage.
- Le mot clé **implements** permet de spécifier une ou des **interfaces** que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

3.2 Les modificateurs d'accès (général)

Les modificateurs d'accès se placent avant le type de l'objet, ils s'appliquent aux classes et/ou aux propriétés et/ou aux méthodes, de plus, ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de **visibilité** des entités (méthodes ou attributs) qui permettent de définir des niveaux de protection différents, aussi appelé niveau d'accès.

Modificateur	Description
private	C'est le niveau de protection le plus fort . Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe prévue à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarée abstract car elles ne peuvent pas être redéfinies dans les classes filles.
protected	Si une méthode ou une variable est déclarée protected , seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder.
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique (il est préférable d'écrire des méthodes pour la consulter et la modifier).

Lorsque l'on ne précise **aucun modificateur**, une entité (classe, propriété ou méthode) est visible par toutes les classes se trouvant dans le même package (**package friendly**).

4 Visibilité

4.1 Le principe de la visibilité.

Afin de pouvoir garantir la protection des données (principe de l'encapsulation), il convient de pouvoir masquer certaines données et méthodes internes les gérant, et de pouvoir laisser visibles certaines autres devant servir à la gestion publique de l'objet.

4.2 Champs et méthodes publics

Les champs et méthodes dits **publics** sont accessibles depuis tous les descendants et dans tous les modules. Ils n'ont pas de restriction particulière.

Les méthodes permettant d'accéder aux champs d'ordre privé sont appelées accesseurs

Les accesseurs pour chaque champ privé ne sont pas obligatoires.

Les constructeurs et les destructeurs d'un objet doivent bénéficier de la visibilité public

4.3 Champs et méthodes privés

La visibilité **privée** restreint la portée d'un champ ou d'une méthode au module où il ou est déclaré.

Ainsi, si un objet est déclaré dans une unité avec un champ privé, alors ce champ ne pourra être accédé qu'à l'intérieur même de l'unité.

4.4 Champs et méthodes protégés

La visibilité **protégée** correspond à la visibilité privée excepté que tout champ ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où ils se situent

5 La notion d'objet

Les objets contiennent des attributs et/ou des méthodes. Les propriétés sont des variables élémentaires ou des objets nécessaires au fonctionnement de l'objet. En Java, une application est un objet. La classe est la description d'un objet. **Un objet est une instance d'une classe**. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes pour chaque objet.

En Java, tous les objets sont instanciés par allocation dynamique (en fonction des besoins et au moment de l'utilisation).

5.1 Instanciation d'un objet

Syntaxe

```
nom_classe    nomVariableObjet ;
```

Déclaration, puis instanciation	Déclaration et instanciation simultanée
MaClasse obj1; obj1 = new MaClasse();	MaClasse obj1 = new MaClasse();

- L'opérateur new crée une instance de la classe et l'associe à la variable. De plus, il appelle une méthode particulière appelée **constructeur**.
- **Chaque instance de classe nécessite sa propre variable**, par contre plusieurs variables peuvent désigner la même instance (objet).

5.2 La durée de vie d'un objet

La durée de vie de l'objet n'est pas forcément la durée de vie du programme.

L'objet existe à partir de son instanciation, jusqu'à la suppression de l'objet. Elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (**garbage collector**).

5.3 L'objet null

L'objet null est utilisable partout. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre.

Le fait d'initialiser une variable référant un objet à null permet au ramasse miette (**garbage collector**) de libérer la mémoire allouée à l'objet.

5.4 La comparaison d'objets

Les variables de type objet que l'on déclare **ne contiennent pas un objet** mais une **référence** vers cet objet.

- La méthode **equals** héritée de Object, permet de comparer l'égalité des variables de deux instances.
- La méthode **getClass()** de la classe Object, permet de s'assurer que deux objets sont de la même classe.

```
boolean identique;  
identique=obj1.getClass().equals(obj2.getClass());
```

- l'opérateur **instanceof** permet de déterminer la classe de l'objet qui est passé en paramètre.

5.5 La variable this

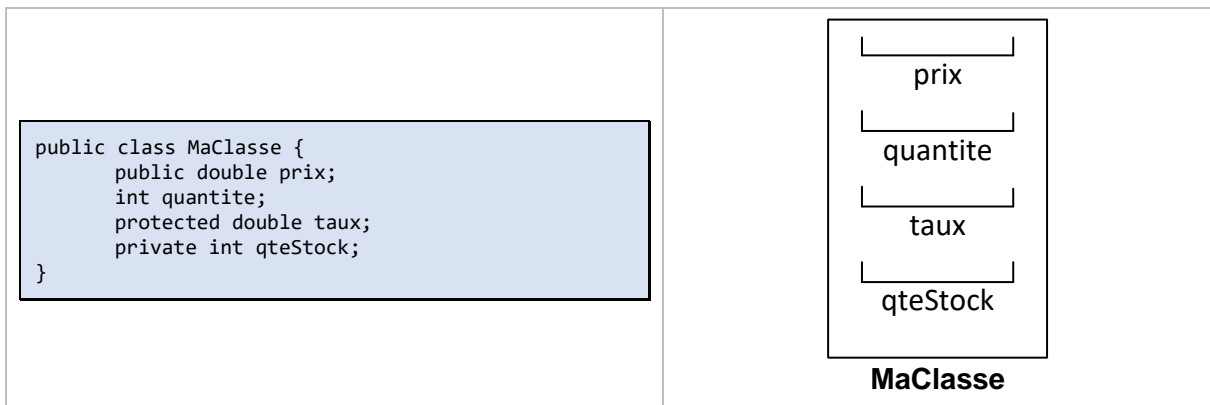
Cette variable référence dans une méthode l'instance de l'objet en cours d'utilisation. **this** est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

5.6 Déclarations des propriétés

Les données d'une classe sont contenues dans des variables nommées **propriétés** ou **attributs**. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

5.6.1 Les variables d'instances

Une variable d'instance est une simple variable déclarée dans le corps de la classe.



Déclaration d'un fichier qui contient la méthode main pour utiliser la classe MaClasse

```
package java_01;

import java.io.Console;

public class Debut {

    public static void main(String[] args)
    {
        // On instancie la classe MaClasse
        MaClasse m1 = new MaClasse();

        // On modifie la valeur de la propriété prix (public)
        m1.prix = 12;
        System.out.println("m1.prix=" + m1.prix);

        // On modifie la valeur de la propriété taux (protected -->
        même package)
        m1.taux = 5.5;
        System.out.println("m1.taux=" + m1.taux);

        // On tente de modifier la propriété qteStock (private -->
        erreur, seule la classe peut la modifier)
        // m1.qteStock = 3;

    }
}
```

5.6.2 Les variables de classes

Chaque instance de la classe **partage la même variable**, elle n'existe donc qu'une seule fois en mémoire et toutes les instances peuvent la modifier.

Syntaxe

```
modificateur    static type    variableClasse;
```

Exemple

```
public class MaClasse {  
  
    public static int compteur ;  
  
    public double prix;  
    int quantite;  
    protected double taux;  
    private int qteStock;  
  
}
```

5.6.3 Les constantes

Une constante est une propriété, qui une fois initialisée, ne peut pas être modifiée.

Syntaxe

```
modificateur    final  type    NOM_CONSTANTE= valeur;
```

Exemple

```
public class MaClasse {  
  
    private final double TAUX_STANDARD=19.6;  
  
    public static int compteur ;  
  
    public double prix;  
    int quantite;  
    protected double taux;  
    private int qteStock;  
  
}
```

- Pour les constantes, le nom est en majuscules avec les mots séparés par des tirets de soulignement (_).
- Le mot-clé **final** attribué à une variable **empêche** sa modification, par contre si c'est un objet on peut modifier ses données internes. Si l'on déclare un tableau final, on ne peut pas le remplacer par un autre tableau, mais on peut modifier ses données.

5.7 Déclarations des méthodes

Les méthodes sont des fonctions qui implémentent (permettent de coder) les traitements de la classe. Une méthode reçoit (éventuellement) des paramètres (**données**) et peut renvoyer une valeur (**type de retour**).

- **type de retour** : Le type de retour peut être un type classique (int, double, String ...), le nom d'une classe ou **void** (void → aucun type, dans ce cas la méthode ne renvoie rien, donc pas de return dans la méthode).
- **Le type et le nombre d'arguments** transmis à la méthode doit être **identique** à la **déclaration**.
- On ne peut **pas** donner **de valeurs par défaut** aux paramètres.
- Les arguments sont passés **par valeur**. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine (c'est une copie de la variable). Il est possible de modifier l'objet grâce à ces méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre (ce changement n'aura lieu que localement à la méthode).

Syntaxe

```
modificateur    type_retour    nomMéthode(type param1, ...)  
{  
    Contenu de la méthode  
}
```

L'ordre de déclaration des méthodes dans une classe n'a pas d'importance (une méthode ne doit pas être déclarée avant une autre, pour pouvoir être appelée par cette dernière).

5.7.1 Les modificateurs des méthodes

Modificateur	Description
public	La méthode est accessible aux méthodes des autres classes
private	L'usage de la méthode est réservé aux autres méthodes de la même classe
protected	La méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes.
final	La méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	La méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	La méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	Le code source de la méthode est écrit dans un autre langage

5.7.2 L'instruction return

Le mot return permet à une méthode de renvoyer une valeur (toutes les instructions qui suivent return sont ignorées).

Exemple

```
public int somme(int a, int b) {  
    int total;  
    total= a + b;  
    return total;  
}
```

5.8 Utilisation d'une méthode

Les appels de méthodes sont appelés « échanges de message ».

Syntaxe

```
nom_objet.nomMéthode(parametre, ... );
```

Exemple

```
...  
public class MaClasse {  
    ...  
    int somme(int a, int b) {  
        int total;  
        total= a + b;  
        return total;  
    }  
}  
...  
  
int resultat;  
MaClasse c=new MaClasse();  
resultat=c.somme(5,6);
```

5.9 La surcharge d'une méthode

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Exemple

```
...  
  
public class MaClasse {  
    int somme(int a, int b) {  
        int total;  
        total= a + b;  
        return total;  
    }  
  
    double somme(double a, double b) {  
        double total;  
        total= a + b;  
        return total;  
    }  
}  
  
...  
  
int resultat1;  
double resultat2;  
  
MaClasse c=new MaClasse();  
resultat1=c.somme(5,6);  
resultat2=c.somme(5.4,6.7);  
  
System.out.println (resultat1);  
  
// Pour un affichage propre pensez au format  
java.text.DecimalFormat df = new java.text.DecimalFormat("0.##");  
System.out.println (df.format(resultat2));
```

Vous ne pouvez pas avoir deux méthodes de même nom dont tous les paramètres sont identiques, mais qui auraient un type de retour différent.

5.10 Le constructeur

Le constructeur, est une méthode qui permet de créer les objets (réserver la place en mémoire) et éventuellement d'initialiser les variables (donner une valeur de départ aux variables). Il est appelé systématiquement lors de l'instanciation de la classe (création d'un objet). Si vous n'avez pas défini de constructeur, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement.

Le **constructeur** est une **méthode** comme une autre, excepté que **son nom doit obligatoirement correspondre à celui de la classe**, il n'est pas typé, pas même void. On peut surcharger un constructeur.

Dès que l'on crée un constructeur, Java considère que **le programmeur prend en charge la création des constructeurs** et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètre...

Exemple

```
public class MaClasse {
    int a;

    public MaClasse() { // Constructeur sans paramètre
    }

    public MaClasse(int valeur) { // Constructeur avec paramètre
        this.a = valeur; // Pour donner une valeur à notre variable
    }
}

...

MaClasse c1 = new MaClasse(); // Appel du constructeur sans paramètre
MaClasse c2 = new MaClasse(5); // Appel du constructeur avec paramètre
```

5.11 Valeur par défaut pour les propriétés

Deux écritures possibles

```
public class MaClasse {
    String nom="inconnu";
    int age=25;
    // Constructeur sans paramètre
    public MaClasse() {
    }
}
```

```
public class MaClasse {
    String nom;
    int age;
    // Constructeur sans paramètre
    public MaClasse() {
        this.nom="inconnu";
        this.age=25;
    }
}
```

5.12 Le destructeur

Le destructeur (finalizer en anglais) est une méthode qui est appelée lors de la libération de la mémoire par le garbage collector.

Pour créer un destructeur, il faut définir la méthode `finalize()` dans la classe.

L'appel au destructeur n'est pas garanti, il n'est en plus, pas possible, de connaître le moment où un objet sera traité par le garbage collector

5.13 L'accesseur

L'accesseur est une méthode publique qui donne l'accès à une variable d'instance privée.

Accesneur en lecture, il permet de lire une propriété, **par convention**, on le définit avec le préfixe **get** (on utilise parfois, le préfixe **is** pour un attribut de type booléen).

Accesneur en écriture (appelé aussi **mutateur**), il permet de modifier une propriété, **par convention**, on le définit avec le préfixe **set**.

Rappel

L'encapsulation permet de sécuriser l'accès aux données. Ainsi, les données déclarées **private** à l'intérieur d'une classe ne peuvent être lues et modifiées que par des méthodes définies dans la classe. Si une autre classe veut accéder aux données de la classe, elle devra appeler une méthode de la classe prévue à cet effet.

Exemple

```
public class MaClasse {
    private int valeur=5;

    public int getValeur() {    // Déclaration accesneur en lecture
        return this.valeur;
    }

    public void setValeur(int d) {    // Déclaration accesneur en écriture
        this.valeur = d;    // Pour donner une valeur à notre variable
    }
}

...

int donnee;
MaClasse c = new MaClasse();
donnee = c.getValeur();    // On récupère la valeur de la variable privée
                             valeur.
c.setValeur(20);
```

5.14 Comparer deux objets

Il faut faire attention à la comparaison de deux objets.

- Si l'on utilise l'opérateur **==** pour comparer deux objets, il renvoie **true** (vrai), si les deux objets **sont les mêmes**.
- Si on les compare avec la méthode **equals()**, celle-ci **doit** retourner **true** (vrai), si les deux objets ont un contenu équivalent.

Exemple

```
public class UneClasse {  
  
    private String nom;  
    private int age;  
  
    // Constructeur  
    public UneClasse(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
  
    // *****  
    //   Les getters et les setters  
    // *****  
  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}  
  
...  
  
UneClasse a1 = new UneClasse("DUPOND",30);  
UneClasse a2 = new UneClasse("DUPOND",30);  
  
System.out.println(a1 == a2); // Retourne false  
System.out.println(a1.equals(a2)); // Retourne false
```

- Pour que **equals()** soit égal à **true**, si les contenus sont les mêmes, **on doit implémenter** la méthode, méthode **equals()** (on la surcharge) en **comparant tous les attributs de la classe**.
- Par défaut, dans **Object** (la super classe de tous les objets Java), la méthode **equals()** utilise **==** pour comparer deux objets.

Exemple

```
public class UneClasse {

    private String nom;
    private int age;

    // Constructeur
    public UneClasse(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // *****
    // Les getters et les setters
    // *****

    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

...

UneClasse a1 = new UneClasse("DUPOND",30);
UneClasse a2 = new UneClasse("DUPOND",30);

System.out.println(a1 == a2); // Retourne false
System.out.println(a1.equals(a2)); // Retourne false

// On crée une variable de type UneClasse (null)
UneClasse a3;

// On fait pointer (référence) a3 vers l'objet de a1
a3=a1;
System.out.println(a1 == a3); // Retourne true (même objet)
System.out.println(a1.equals(a3)); // Retourne true (même objet)
// Vérification de la dernière comparaison
a3.setAge(99);
// La modification de la variable a3 a bien modifié l'objet
référéncé par a1
System.out.println("age de a1 = " + a1.getAge());
```

Pour comparer deux classes, il faut utiliser la méthode equals() et l'implémenter pour tous nos objets

On surcharge la méthode equals() pour notre classe.

Exemple

```
public class UneClasse {

    private String nom;
    private int age;

    // Constructeur
    public UneClasse(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Les getters et les setters

    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    // On surcharge la méthode equals() pour qu'elle fonctionne avec les objets de notre classe
    @Override
    public boolean equals(Object obj) {
        // Si les deux variables objet font référence à la même instance de l'objet
        if (this == obj) {
            return true;
        }

        // Si l'objet est null,
        if (obj == null) {
            return false;
        }

        // Si la classe des deux objets est différente
        if (getClass() != obj.getClass()) {
            return false;
        }

        // On crée un nouvel objet en castant la classe de l'objet passé en paramètre
        // pour comparer leur contenu
        UneClasse other = (UneClasse) obj;

        // Comparaison de l'age
        if (age != other.age) {
            return false;
        }

        // comparaison d'une chaîne de caractère (nom) on teste null et l'égalité des chaînes avec equals
        if (nom == null) {
            if (other.nom != null) return false;
        } else if (!nom.equals(other.nom)) {
            return false;
        }
        return true;
    }

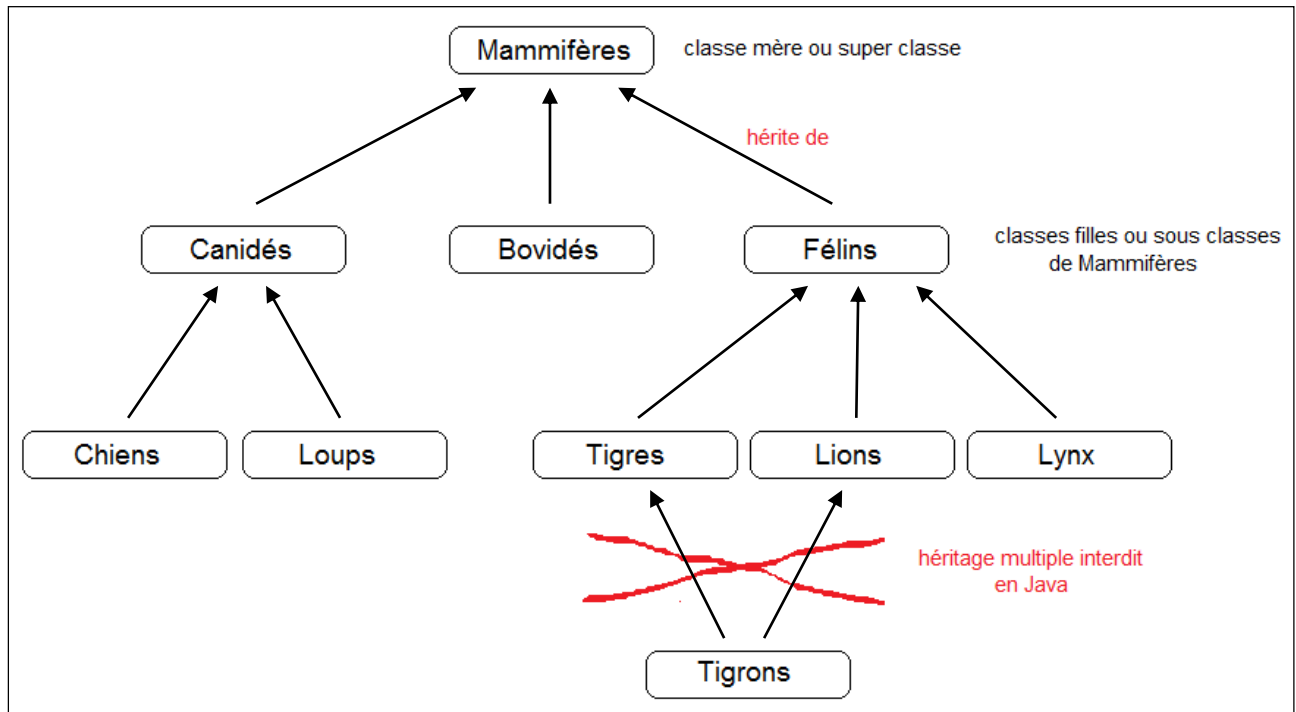
    ...
    UneClasse a1 = new UneClasse("DUPOND",30);
    UneClasse a2 = new UneClasse("DUPOND",30);

    System.out.println(a1 == a2); // Retourne false
    System.out.println(a1.equals(a2)); // Retourne true
}
```

6 L'héritage

L'héritage permet la réutilisation du code et améliore son évolution.

Une classe qui hérite d'une autre classe est appelée **classe fille** ou **sous-classe**, tandis que la classe dont elle a hérité, est appelée **classe mère** ou **super classe**. L'héritage multiple est interdit en Java.



Notions d'héritage

- La classe **Tigris** **hérite** de la classe **Félin**, la classe **Félin** est la **classe de base** ou **classe mère** de **Tigris**.
- Les classes **Tigris**, **Lion** et **Lynx** sont des **classes filles** ou **classes dérivées** de la classe **Félin**.
- La classe **Mammifères** est la **super classe**, car toutes les classes héritent d'elle.

6.1 Quand doit-on mettre en place l'héritage ?

Soit A et B deux classes. Pour qu'un héritage soit possible, **il faut que l'on puisse dire que A est un B**. On peut dire qu'un **Félin** est un **Mammifère**, donc héritage. Une voiture est un véhicule, donc héritage.

6.2 L'héritage d'une classe

Une classe fille, récupère les propriétés et les méthodes de la classe mère, elle peut les utiliser telles quelles, mais elle peut aussi les redéfinir.

Le constructeur d'une classe, n'est pas hérité par une classe fille.

6.3 Pour une classe mère

- ✓ Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et à toutes les autres classes.
- ✓ Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais via les méthodes héritées.
- ✓ Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur **protected**. La variable ainsi définie sera héritée dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

1. Pour empêcher la redéfinition d'une méthode (surcharge), il faut la déclarer avec le modificateur final.

6.4 Pour une classe fille

- ✓ Si la méthode héritée convient partiellement (du fait de la spécialisation apportée par la classe fille), il faut la redéfinir, ou la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via **super**).
- ✓ Pour garantir l'évolution du code, si une méthode héritée ne convient pas, il faut la **redéfinir** ou la **surcharger** sans appeler la méthode héritée lors de la redéfinition (si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge).

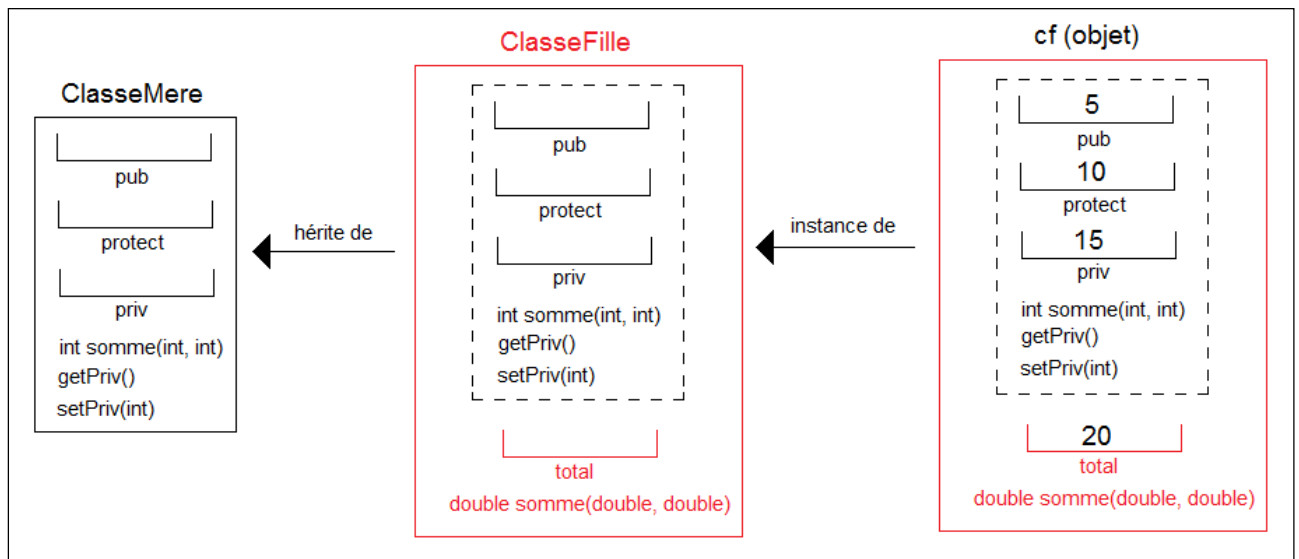
Syntaxe

```
modificateur    class  nom_classe_fille    extends nom_classe_mère {  
... }
```

Exemple

```
public class ClasseMere {  
    public int pub= 5;  
    protected int protect=10;  
    private int priv=15;  
  
    public int getPriv() {           // Déclaration accesseur en lecture  
        return priv;  
    }  
  
    public void setPriv(int valeur) { // Déclaration accesseur en écriture  
        priv = valeur; // Pour donner une valeur à notre variable privée  
    }  
  
    public int somme(int a, int b){           // Une méthode somme  
        return (a + b);  
    }  
  
}  
...  
  
public class ClasseFille extends ClasseMere{  
  
    int total=20; // Total est une variable propre à ClasseFille  
  
    // Ceci est une surcharge de la fonction  
    public double somme(double a, double b){  
        return (a + b);           // Somme,n'est pas une redéfinition  
    }  
}  
...  
  
// A écrire dans une classe différente  
  
int d1, d2, d3, res1; // 4 variables de type int
```

En image



La classe ClasseFille **hérite** de ClasseMere, elle récupère donc toutes ses propriétés et méthodes, et déclare une nouvelle propriété (total) et surcharge la méthode somme. **L'instanciation** de ClasseFille crée un objet cf. La machine virtuelle Java **réserve une place en mémoire** pour y stocker ses propriétés et méthodes, et initialise les propriétés.

7 L'instruction super

Le mot clé **super** permet de faire référence à l'objet parent.

Le mot-clé **super** permet de désigner des membres de la superclasse, et manipuler ces derniers à partir d'une sous-classe.

super() sert aussi à appeler un constructeur de la classe parente d'une classe.

Ceci est rendu nécessaire lorsque qu'on déclare une classe étendant une autre classe, et que celle-ci ne possède pas de constructeur avec les mêmes arguments.

7.1 Utiliser une variable de la classe mère

Syntaxe

```
super.nomVariable;
```

7.2 Utiliser une méthode de la classe mère

Syntaxe

```
super.nomMethode();
```

7.3 Appeler le constructeur de la classe mère

Ce mot clé peut également être **utilisé avec une liste d'arguments optionnels pour appeler le constructeur** de la superclasse.

Syntaxe

```
super(argument1, ...);
```


Exemple d'utilisation d'un constructeur et de l'héritage

```
public class ClasseMere {
    int a;
    int info=10;
    public ClasseMere() { // Constructeur sans paramètres
    }

    public ClasseMere(int valeur) { // Constructeur avec paramètre
        this.a = valeur; // Pour donner une valeur à notre variable
    }
}
```

```
public class ClasseFille extends ClasseMere {
    int b;
    int info=77;

    public ClasseFille() { // Constructeur sans paramètres
        super();
    }

    public ClasseFille(int v1,int v2 ) { // Constructeur avec
paramètre
        super(v1); // Appel du constructeur de la classe mère
        this.b = v2; // Pour donner une valeur à notre variable
    }

    public int recupInfo() {
        return super.info;
    }
}
```

```
ClasseFille c1 = new ClasseFille(); // Appel du constructeur sans
paramètre
ClasseFille c2 = new ClasseFille(5,7); // Appel du constructeur avec
paramètre
System.out.println("info (classe fille) =" + c1.info);
System.out.println("info (classe mère)=" + c1.recupInfo() );
```

8 L'interface

Une interface (**classe abstraite**) permet de définir un ensemble de constantes et de déclaration de méthodes (c'est-à-dire sans écrire de code pour ces méthodes) qui **devront être implémentés dans les classes filles**. Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par l'interface.

Les interfaces permettent de mettre en œuvre un mécanisme de remplacement de l'héritage multiple (interdit en Java) et du polymorphisme (les objets héritent des mêmes caractéristiques mais leurs actions peuvent être différentes).

Ce mécanisme d'interface permet de créer des classes, qui héritent, de plusieurs interfaces simultanément.

8.1 Déclaration d'une interface

Une interface est **implicitement** déclarée avec le modificateur **abstract** (abstrait), mais on peut utiliser le modificateur d'accès public.

Syntaxe

```
public] interface nomInterface [implements nomInterface1, ... ] { ... }
```

Exemple

```
interface TVA{  
    static double tauxTva=20.6;  
    double prixTtc();  
}
```

Exemple d'utilisation d'une interface

```
public interface TVA{ // On crée une interface  
    static double TAUX_TVA=20.6;  
    double prixTtc();  
}
```

```
// On hérite de l'interface TVA  
public class Article implements TVA {  
    public double prix=15;  
    public double prixTtc(){ // On définit la fonction prixTtc()  
        double prTtc;  
        prTtc = prix + (prix* TAUX_TVA)/100;  
        return prTtc;  
    }  
}
```

```

// On hérite de l'interface Tva
public class Location implements Tva {
    public double prixLoc=100;
    public double charges = 50;
    public double prixTtc() { // On définit la fonction prixTtc()
        double prTtc;
        prTtc = prixLoc + (prixLoc* TAUX_TVA)/100 + charges;
        return prTtc;
    }
}

```

Dans cet exemple, nous avons défini une interface Tva, dont les classes Articles et Location héritent, dans chacune d'elles, nous avons défini la méthode prixTtc dans chaque classe, ces deux **méthodes** sont différentes.

9 Les classe Internes

La définition d'une classe à l'intérieur d'une autre classe, définit une **classe interne**.
La classe interne à **accès aux méthodes et aux attributs de la classe englobante**.

10 Les classes locales

Lorsqu'une classe est définie dans un bloc d'une classe, la portée est limitée à ce seul bloc, on parle de classe locale.

Une classe locale peut accéder aux attributs de la classe englobante ainsi qu'aux paramètres et variables locales de la méthode où elle est définie, à condition que ceux-ci soit spécifiés **final**.

Une classe locale ne peut pas être **static**.

10.1 Les classes anonymes

Une classe peut être déclarée au moment de l'instanciation de sa classe parente, sans lui donner de nom par dérivation d'une super classe, ou par implémentation d'une interface, on parle alors de classe anonyme.

Dérivation de super classe	Implémentation d'interface
<pre> public class MaClasse ... SuperClasse c = new SuperClasse() { // méthodes redéfinies }; ... }; </pre>	<pre> public class MaClasse ... Interface c = new Interface() { // implementation des méthodes de l'Interface }; ... }; </pre>

Les classes anonymes sont (surtout) utilisées pour implémenter les méthodes d'une interface Listener (la gestion des événements, action sur un bouton par exemple).

11 La classe abstraite

Une **classe abstraite** est une classe normale, dans laquelle, on peut définir des propriétés et des méthodes communes (pour les classes filles), mais on pourra aussi déclarer des méthodes **qui ne seront pas implémentées** (c'est-à-dire sans écrire de code de ces méthodes), elles **devront obligatoirement être implémentées dans les classes filles**.

Une classe abstraite **n'est pas instanciable** (on ne peut pas créer d'objet de cette classe), elle sert à **factoriser du code (réunir un code commun à plusieurs classes)**, permet de définir un squelette identique pour toutes les classes filles.

Une classe qui hérite d'une classe abstraite doit **obligatoirement** implémenter les méthodes non définies (déclarées **abstract** dans la classe parente), mais elle n'est pas obligée de réimplémenter les méthodes déjà implémentées dans la classe parente (ce qui facilite la maintenance du code).

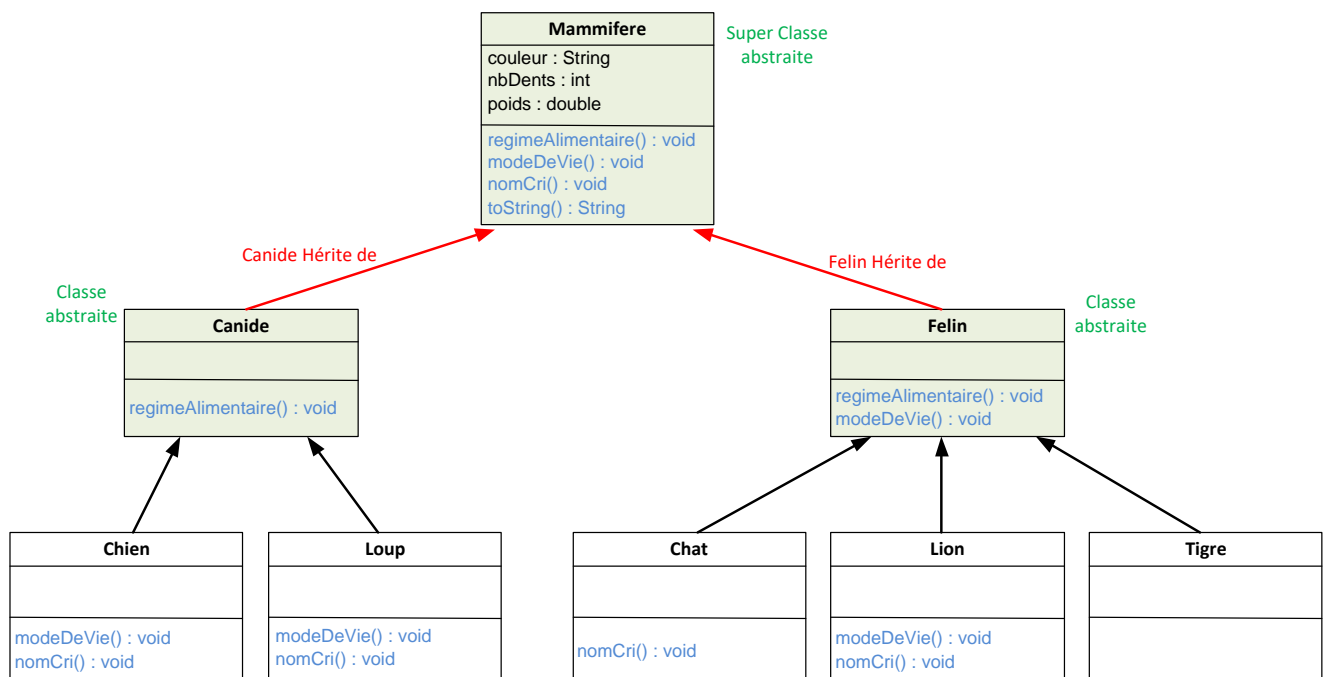
Toute sous-classe d'une classe abstraite doit :

- ✓ Implémenter toutes les méthodes abstraites héritées (on dit que c'est une classe **concrète**).
- Ou
- ✓ Elle doit être déclarée **abstract**.

11.1 Les méthodes abstraites

Certains comportements n'étant pas identiques dans tous les objets, ils seront déclarés (**abstract**) dans la classe et définis dans les classes filles.

Dès qu'une classe contient une **méthode abstraite**, elle doit elle aussi être **déclarée abstraite**.



Exemple

```
public abstract class Mammifere {

    protected String couleur;
    protected int nbDents;
    protected double poids;
    protected double taille;

    // Méthodes abstraites à définir dans chaque classe fille
    abstract void regimeAlimentaire();
    abstract void modeDeVie ();

    // Méthodes standards
    protected void nomCri()
    {
        System.out.println("Cri indéterminé.");
    }

    public String toString(){
        String ch = " \rJe suis un objet de la " + this.getClass() +
        ", je suis " + this.couleur + ", j'ai " + this.nbDents + " dents et
        je pèse " + this.poids + " Kg.";
        return ch;
    }
}
```

```
public abstract class Canide extends Mammifere {
    void modeDeVie() {
        System.out.println("Je vis en meute.");
    }

    void regimeAlimentaire(){
        System.out.println("Je suis carnivore.");
    }
}
```

```

public abstract class Felin extends Mammifere {

    void modeDeVie() {
        System.out.println("Je suis solitaire.");
    }

    void regimeAlimentaire(){
        System.out.println("Je suis carnivore.");
    }
}

```

```

public class Loup extends Canide {

    public Loup() {
    }

    public Loup(String couleur, int nbDents, double poids) {
        this.couleur = couleur;
        this.nbDents = nbDents;
        this.poids = poids;
    }

    void modeDeVie() {
        System.out.println("Je vis en meute.");
    }

    protected void nomCri(){
        System.out.println("Je hurle, la nuit");
    }
}

```

```

public class Chien extends Canide {

    public Chien() {
    }

    public Chien(String couleur, int nbDents, double poids) {
        this.couleur = couleur;
        this.nbDents = nbDents;
        this.poids = poids;
    }

    void modeDeVie() {
        System.out.println("Je vis souvent seul (par obligation, je ne suis plus sauvage)");
    }

    protected void nomCri(){
        System.out.println("J'aboie (trop souvent) !");
    }
}

```

```
public class Chat extends Felin {  
  
    public Chat() {  
    }  
  
    public Chat(String couleur, int nbDents, double poids) {  
        this.couleur = couleur;  
        this.nbDents = nbDents;  
        this.poids = poids;  
    }  
  
    protected void nomCri(){  
        System.out.println("Je miaule ou feule");  
    }  
}
```

```
public class Lion extends Felin {  
  
    public Lion() {  
    }  
  
    public Lion(String couleur, int nbDents, double poids) {  
        this.couleur = couleur;  
        this.nbDents = nbDents;  
        this.poids = poids;  
    }  
  
    void modeDeVie() {  
        System.out.println("Je vis en groupe et parfois seul.");  
    }  
  
    protected void nomCri(){  
        System.out.println("Je rugit.");  
    }  
}
```

```

public class Tigre extends Felin {

    public Tigre() {
    }

    public Tigre(String couleur, int nbDents, double poids) {
        this.couleur = couleur;
        this.nbDents = nbDents;
        this.poids = poids;
    }
}

```

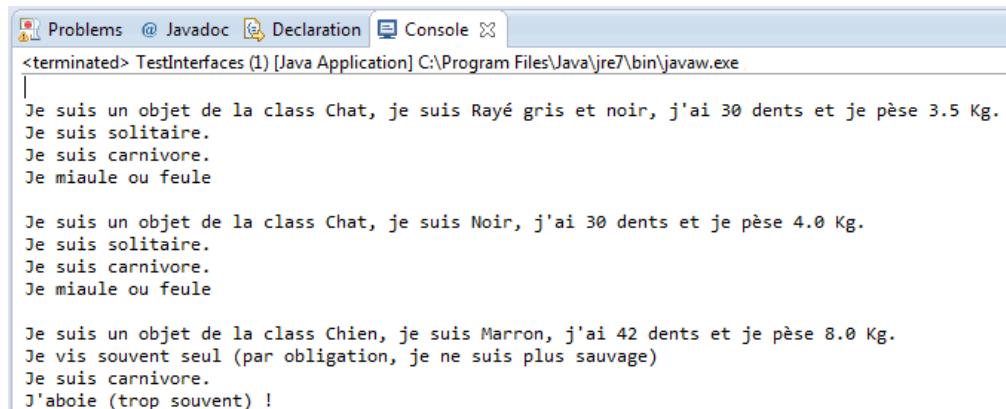
```

...
Chat ch = new Chat("Rayé gris et noir",30,3.5);
System.out.println(ch.toString());
ch.modeDeVie();
ch.regimeAlimentaire();
ch.nomCri();

Mammifere c = new Chat("Noir",30,4);
System.out.println(c.toString());
c.modeDeVie();
c.regimeAlimentaire();
c.nomCri();

Mammifere paf = new Chien("Marron",42,8);
System.out.println(paf.toString());
paf.modeDeVie();
paf.regimeAlimentaire();
paf.nomCri();

```



```

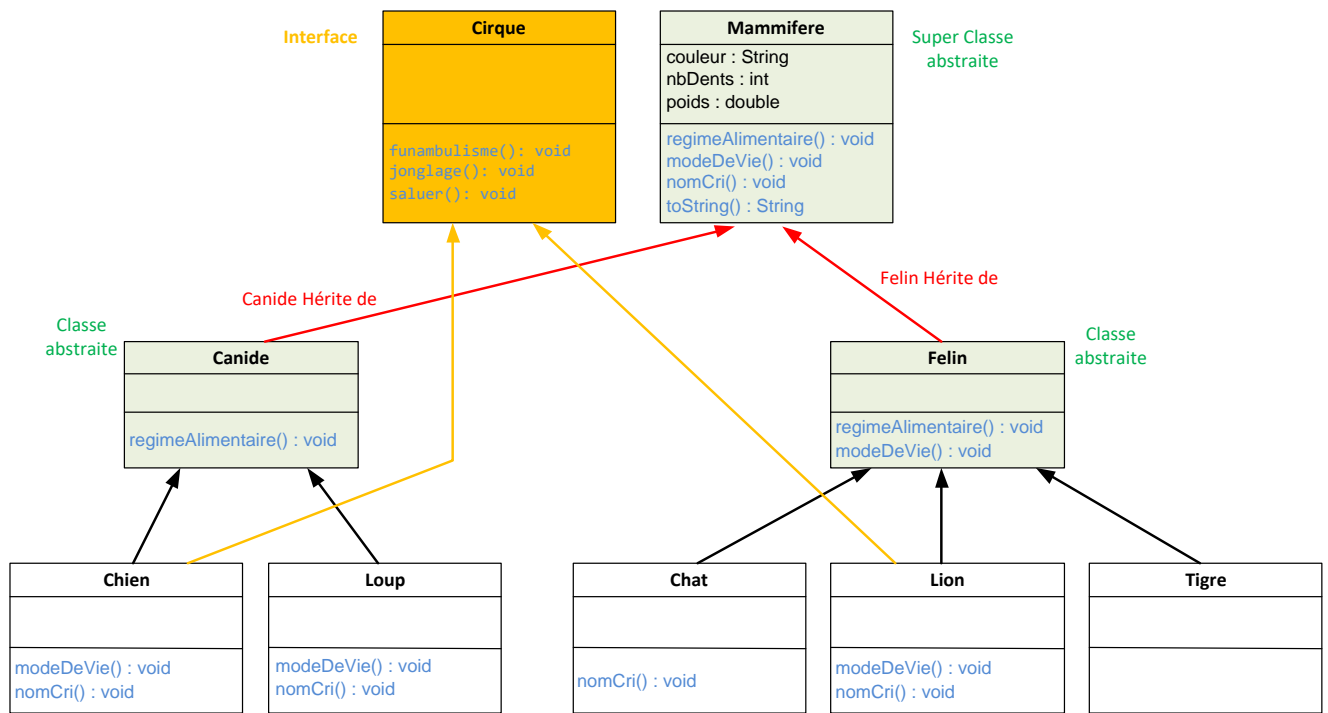
<terminated> TestInterfaces (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
Je suis un objet de la class Chat, je suis Rayé gris et noir, j'ai 30 dents et je pèse 3.5 Kg.
Je suis solitaire.
Je suis carnivore.
Je miaule ou feule

Je suis un objet de la class Chat, je suis Noir, j'ai 30 dents et je pèse 4.0 Kg.
Je suis solitaire.
Je suis carnivore.
Je miaule ou feule

Je suis un objet de la class Chien, je suis Marron, j'ai 42 dents et je pèse 8.0 Kg.
Je vis souvent seul (par obligation, je ne suis plus sauvage)
Je suis carnivore.
J'aboie (trop souvent) !

```


Utilisons une interface pour simuler un héritage multiple.



L'interface Cirque.

Exemple

```
public interface Cirque{           // On crée l'interface Cirque
    public void funambulisme();

    public void jonglage();

    public void saluer();
}
```

Les classes Chien et Lion, vont implémenter l'interface Cirque.

```
public class Chien extends Canide implements Cirque {

    public Chien() {
    }

    public Chien(String couleur, int nbDents, double poids) {
        this.couleur = couleur;
        this.nbDents = nbDents;
        this.poids = poids;
    }

    void modeDeVie() {
        System.out.println("Je vis souvent seul (par obligation, je ne suis plus sauvage)");
    }

    protected void nomCri(){
        System.out.println("J'aboie (trop souvent) !");
    }

    // Implémentation des méthodes de l'interface Cirque
    @Override
    public void funambulisme() {
        System.out.println("Je marche avec les pattes avant sur un cable.");
    }

    @Override
    public void jonglage() {
        System.out.println("Je ne suis pas une otarie");
    }

    @Override
    public void saluer() {
        System.out.println("Je fais le beau et la révérence");
    }
}
```

```

public class Lion extends Felin implements Cirque{

    public Lion() {
    }

    public Lion(String couleur, int nbDents, double poids) {
        this.couleur = couleur;
        this.nbDents = nbDents;
        this.poids = poids;
    }

    void modeDeVie() {
        System.out.println("Je vis en groupe et parfois seul.");
    }

    protected void nomCri(){
        System.out.println("Je rugit.");
    }
    // Implémentation des méthodes de l'interface Cirque
    @Override
    public void funambulisme() {
        System.out.println("J'ai autre chose à faire !");
    }

    @Override
    public void jonglage() {
        System.out.println("Je suis pas un clown.");
    }

    @Override
    public void saluer() {
        System.out.println("Saluer, Moi le Roi des animaux ?");
    }
}

```