

Langage de programmation

**Java**

**La POO**

MARECAT Gaëtan

**CGI**

# Sommaire



- |   |                    |
|---|--------------------|
| 1 | Les base de la POO |
| 2 | Les classes        |
| 3 | Héritage           |
| 4 | Interface          |
| 5 | Les collections    |

# Les bases de la POO

# Introduction à la POO

La programmation orientée objet (POO) est un paradigme de programmation qui encourage la modélisation des problèmes informatiques en utilisant des objets et des classes pour représenter des entités du monde réel. Elle offre de nombreux avantages, ce qui explique pourquoi elle est largement utilisée.

Voici quelques raisons pour lesquelles la POO est utilisée :

- Modélisation du monde réel => facilite la compréhension et la maintenance
- Abstraction => simplifier la complexité
- Encapsulation => protéger les données
- Réutilisation du code => réduire la duplication
- Maintenance facilitée
- Héritage et polymorphisme => traitement généraliste des objets

En résumé, la POO est utilisée en raison de sa capacité à faciliter la modélisation, la réutilisation, la maintenance et la gestion de la complexité des systèmes logiciels. Elle encourage également une approche structurée du développement logiciel qui améliore la collaboration en équipe et la sécurité.

# Quelques définitions

## Objet

Un objet est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. C'est le concept central de la programmation orientée objet (POO).<sup>1</sup>

## Héritage

L'héritage est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe.<sup>1</sup>

## Encapsulation

L'encapsulation est un concept qui vise à protéger les données d'un système, en général d'un classe. Ce principe engendre un effet de boîte noir autour du système protégeant ainsi ce dernier.

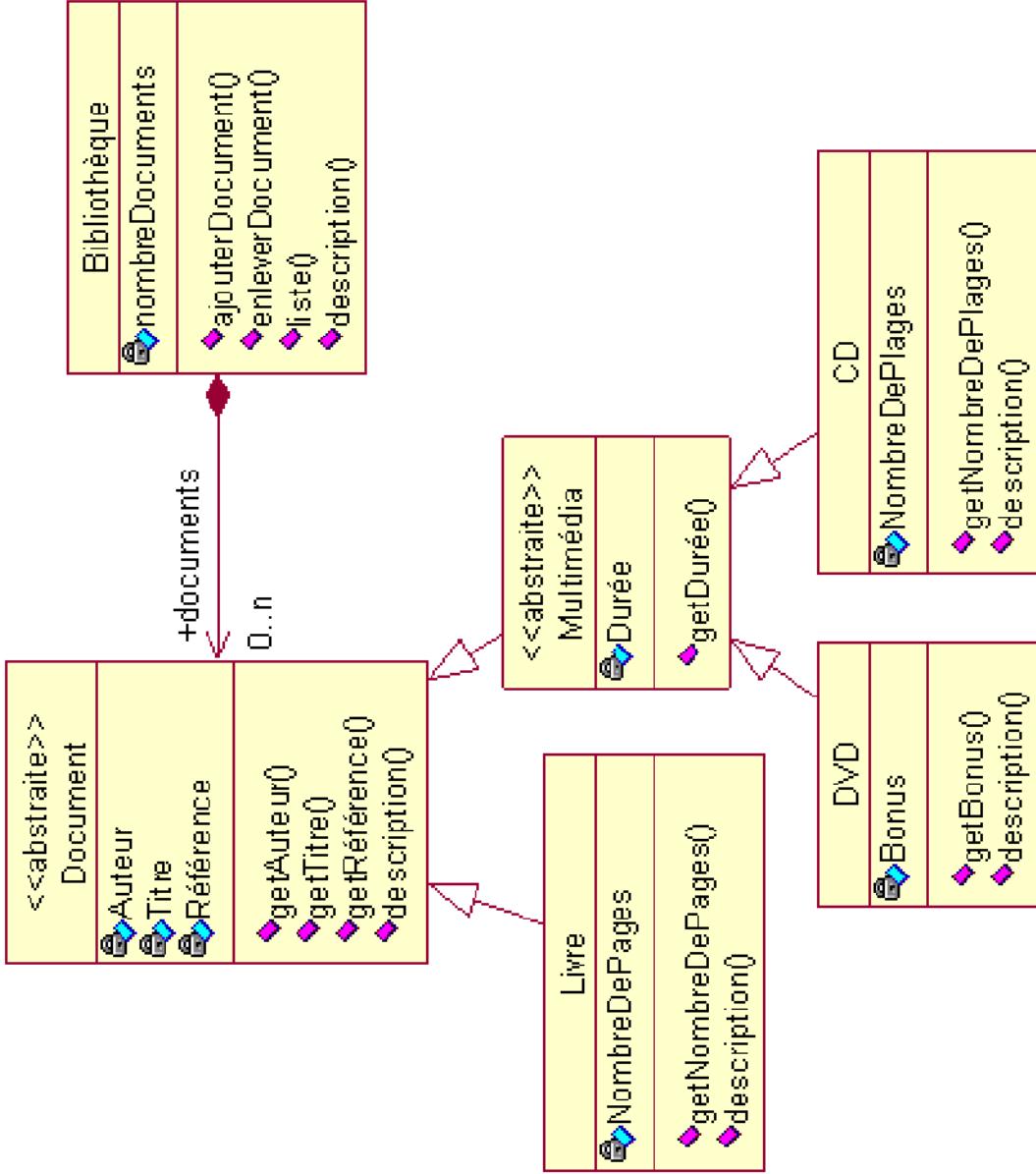
## Polymorphisme

Concept qui permet à une méthode pour une signature donnée d'avoir des comportement différent.

## Composition

En programmation informatique, la composition est une technique qui permet de combiner plusieurs éléments de programmation entre eux pour obtenir un élément ayant une fonctionnalité plus complexe.<sup>1</sup>

# Un petit exemple



# Les classes

# Définition

Une classe est un modèle qui définit les propriétés et les comportements d'un type d'objet. Elle est utilisée pour créer des instances d'objets, qui sont des copies individuelles du modèle.

```
class NomDeClasse {  
    // Définition des propriétés  
    // Définition des méthodes  
}
```

Les propriétés sont des variables qui stockent les données de l'objet.

```
private String nom;
```

# Définition

Les méthodes sont des fonctions qui exécutent des actions sur l'objet.

```
public void afficherNom() {  
    System.out.println(nom);  
}
```

Pour créer une instance d'une classe, on utilise le mot-clé **new**.

Cette instruction crée une nouvelle instance de la classe Personne et stocke son adresse dans la variable p.

On peut ensuite accéder aux propriétés et aux méthodes de l'objet via la référence

```
p.nom = "John Doe";  
p.afficherNom(); // Affiche "John Doe"
```

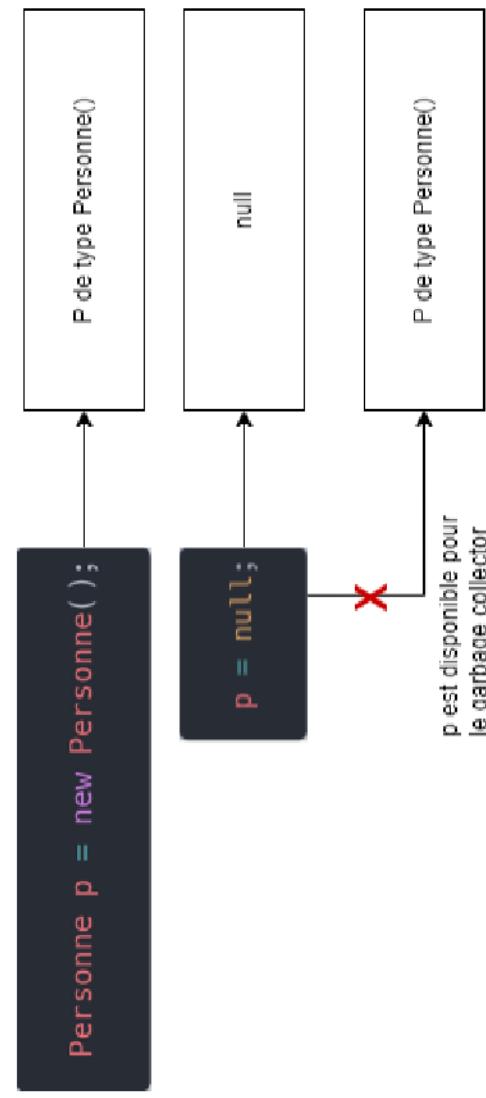
# Référence

En Java, les références sont des variables qui stockent l'adresse d'un objet. Elles sont utilisées pour accéder aux propriétés et aux méthodes de cet objet.

Les références sont différentes des variables primitives, qui stockent directement la valeur d'un type primitif.

Par exemple, la variable « int a = 10; » stocke directement la valeur 10.

La variable « Personne p = new Personne(); » stocke l'adresse de l'objet Personne créé par l'opérateur new.



Le **heap** est une zone de mémoire dynamique qui est utilisée pour stocker les objets. Il est différent de la pile ou (stack), qui est utilisée pour stocker les variables locales et les paramètres des méthodes.

# Comprendre le garbage collector

*Garbage collection* désigne le processus de libération automatique de la mémoire *on the heap* en supprimant les objets qui ne sont plus accessibles dans votre programme.

Cela ne signifie pas forcément qu'un objet éligible au garbage collector sera immédiatement supprimé.

Un objet n'est plus joignable lorsqu'une des deux situations suivantes se produit :

- L'objet n'a plus aucune référence pointant vers lui.
- Toutes les références à l'objet sont sorties du champ d'application.

# Comprendre les déclarations de package et les imports

## Packages

Les classes Java sont regroupées en packages.

L'instruction `import` indique au compilateur dans quel package il doit chercher une classe.

Le fonctionnement est similaire à celui de l'envoi d'une lettre.

Imaginez que vous envoyez une lettre au 123 Main Street, appartement 9. Le facteur apporte d'abord la lettre au 123 Main Street.

Ensuite, le facteur cherche la boîte aux lettres de l'appartement numéro 9.

L'adresse est comme le nom du package en Java. Le numéro d'appartement est comme le nom de la classe en Java.

De la même manière que le facteur ne cherche que les numéros d'appartement dans le bâtiment, Java ne cherche que les noms de classe dans le paquet.

# Comprendre les déclarations de package et les imports

## Wildcards

Les classes d'un même package sont souvent importées ensemble. Vous pouvez utiliser un raccourci pour importer toutes les classes d'un package.

L'astérisque \* est un caractère de remplacement qui correspond à toutes les classes du package.

Chaque classe du package java.util est disponible pour ce programme lorsque Java le compile.

L'instruction import ne fait pas intervenir les packages, fields ou methods enfants ; elle importe uniquement les classes directement dans le package.

```
import java.util.*; // imports java.util.Random among other things
public class NumberPicker {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10));
    }
}
```

# Comprendre les déclarations de package et les imports

## Wildcards

Imaginons que vous souhaitez utiliser la classe AtomicInteger dans le package java.util.concurrent. Quel(s) import(s) le supporte(nt) ?

```
import java.util.*;  
import java.util.concurrent.*;  
import java.util.concurrent.atomic.*;
```

# Ordonner les éléments d'une classe

Elément	Exemple	Requis ?	Où doit-il aller ?
Déclaration de package	package abc;	Non	Première ligne du fichier (à l'exclusion des commentaires ou des lignes vides)
Déclarations d'importation	import java.util.*;	Non	Immédiatement après le package (s'il est présent)
Déclaration de type de premier niveau	public class C	Oui	Immédiatement après l'importation (le cas échéant)
Déclarations de champs	int value;	Non	Tout élément de premier niveau dans une classe
Déclarations de méthodes	void method()	Non	Tout élément de premier niveau d'une classe

# Ordonner les éléments d'une classe

```
package structure; // package must be first non-comment
import java.util.*; // import must come after package
public class Meerkat { // then comes the class
    double weight; // fields and methods can go in either order
    public double getWeight() {
        return weight; } // another field - they don't need to be together
    double height; // 
```

```
/* header */
package structure;
// class Meerkat
public class Meerkat { }
```

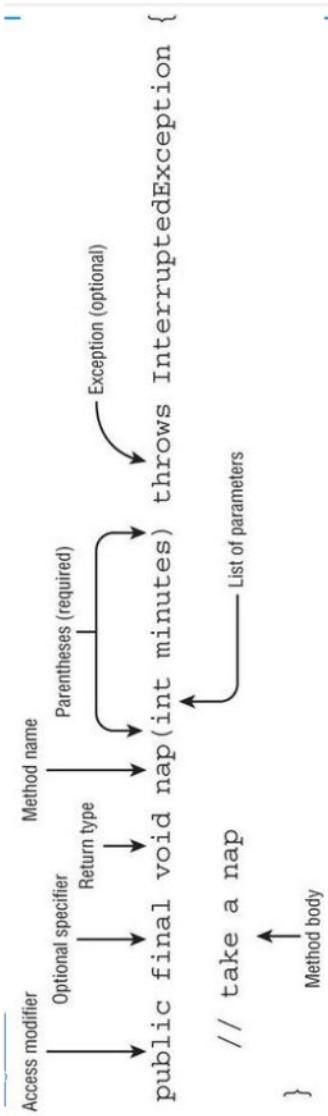
```
import java.util.*; // DOES NOT COMPILE
package structure; // DOES NOT COMPILE
String name; // DOES NOT COMPILE
public class Meerkat { } // DOES NOT COMPILE
```

# This

En Java, le mot-clé `this` fait référence à l'instance actuelle de la classe dans laquelle il est utilisé. Il est principalement utilisé pour éviter les ambiguïtés entre les variables d'instance et les paramètres ou les variables locales ayant le même nom.

```
public class Exemple {  
    private int valeur;  
  
    public void setValeur(int valeur) {  
        // Utilisation de 'this' pour accéder à la variable d'instance  
        this.valeur = valeur;  
    }  
  
    public Exemple() {  
        this(0); // Appelle le constructeur avec un argument  
    }  
  
    public Exemple(int valeur) {  
        this.valeur = valeur;  
    }  
}
```

# Conception de méthodes



## Valeur dans l'exemple nap()

Element	Valeur dans l'exemple nap()	Requis
Access modifier	public	Non
Optional specifier	final	Non
Return type	void	Oui
Method name	nap	Oui
Parameter list	(int minutes)	Oui, mais peut être des parenthèses vides
Method signature	nap(int minutes)	Oui
Exception list	throws InterruptedException	Non
Method body	{ // take a nap }	Oui, sauf pour les méthodes abstraites

# Portée

- **Private**
  - Le modificateur private signifie que la méthode ne peut être appelée qu'à partir de la même classe.
- **Package access**
  - Avec le package access, la méthode ne peut être appelée qu'à partir d'une classe du même package. Cette méthode est délicate car il n'y a pas de mot-clé. Il suffit d'omettre le modificateur d'accès. L'accès par paquetage est parfois appelé accès privé par paquetage ou accès par défaut
- **Protected**
  - Le modificateur protected signifie que la méthode ne peut être appelée qu'à partir d'une classe du même paquet ou d'une sous-classe.
- **Public**
  - Le modificateur public signifie que la méthode peut être appelée de n'importe où.

# Revue de la portée

	<b>private</b>	<b>package</b>	<b>protected</b>	<b>public</b>
Même classe	Oui	Oui	Oui	Oui
Autre class du même package	Non	Oui	Oui	Oui
Une sous classe d'un package différent	Non	Non	Oui	Oui
Une class dans un package différent	Non	Non	Non	Oui

# Spécificateurs optionnels

Contrairement aux modificateurs d'accès, vous pouvez avoir plusieurs spécificateurs dans la même méthode (bien que toutes les combinaisons ne soient pas légales). Dans ce cas, vous pouvez les spécifier dans n'importe quel ordre.

Modifieur	Description
static	Indique que la méthode est un membre de l'objet de la classe partagée
abstract	Utilisé dans une classe abstraite ou une interface lorsque le corps de la méthode est exclu.
final	Spécifie que la méthode ne peut pas être surchargée dans une sous-classe
default	Utilisé dans une interface pour fournir une implémentation par défaut d'une méthode pour les classes qui implémentent l'interface
synchronized*	Utilisé avec un code multithread
native*	Utilisé pour interagir avec du code écrit dans un autre langage, tel que le C++.
strictfp*	Utilisé pour rendre les calculs en virgule flottante portables

# Les Varargs

Règles de création d'une méthode avec un paramètre varargs

- Une méthode peut avoir au maximum un paramètre varargs.
- Si une méthode contient un paramètre varargs, celui-ci doit être le dernier de la liste.

Lesquelles compile ?

```
public class VisitAttractions {  
    public void walk1(int... steps) {}  
    public void walk2(int start, int... steps) {}  
    public void walk3(int... steps, int start) {}  
    public void walk4(int... start, int... steps) {}  
}
```

# Appeler une méthode avec des Varargs

Lorsque vous appelez une méthode avec un paramètre varargs, vous avez le choix. Vous pouvez passer un tableau, ou vous pouvez lister les éléments du tableau et laisser Java le créer pour vous.

Quel que soit la méthode utilisée, la fonction recevra un tableau contenant les éléments.

Vous pouvez même omettre les valeurs varargs dans l'appel de la méthode, et Java créera un tableau de longueur zéro pour vous.

```
// Pass an array
int[] data = new int[] {1, 2, 3};
walk1(data);

// Pass a list of values
walk1(1, 2, 3);

walk1();
```

Accès aux éléments d'un Vararg : L'accès à un paramètre varargs est identique à l'accès à un tableau. Il utilise l'indexation des tableaux.

# Utiliser les Varargs avec d'autres paramètre de méthode

```
1: public class DogWalker {  
2:     public static void walkDog(int start, int... steps) {  
3:         System.out.println(steps.length);  
4:     }  
5:     public static void main(String[] args) {  
6:         walkDog(1); // 0  
7:         walkDog(1, 2); // 1  
8:         walkDog(1, 2, 3); // 2  
9:         walkDog(1, new int[] {4, 5}); // 2  
10:    } }
```

Attention le code compile mais plante à l'exécution :)

`walkDog(1, null); // Triggers NullPointerException in walkDog()`

# Accéder à une donnée *static*

Les méthodes et variables déclarées statiques ne nécessitent pas d'instance de la classe. Elles sont partagées par tous les utilisateurs de la classe.

```
public class Penguin {  
    String name;  
    static String nameOfTallestPenguin;  
}
```

```
public static void main(String[] unused) {  
    var p1 = new Penguin();  
    p1.name = "Lilly";  
    p1.nameOfTallestPenguin = "Lilly";  
    var p2 = new Penguin();  
    p2.name = "Willy";  
    p2.nameOfTallestPenguin = "Willy";  
  
    System.out.println(p1.name);  
    System.out.println(p1.nameOfTallestPenguin);  
    System.out.println(p2.name);  
    System.out.println(p2.nameOfTallestPenguin);  
}  
  
// Lilly  
// Willy  
// Lilly  
// Willy  
// Willy
```

# Accéder à une donnée *static*

Les méthodes statiques ont deux objectifs principaux :

- Pour les méthodes utilitaires ou d'aide qui ne nécessitent pas d'état sur l'objet. Comme il n'est pas nécessaire d'accéder aux variables d'instance, les méthodes statiques évitent à l'appelant d'avoir à instancier un objet uniquement pour appeler la méthode.
- Pour l'état partagé par toutes les instances d'une classe, comme un compteur. Toutes les instances doivent partager le même état. Les méthodes qui utilisent simplement cet état doivent également être statiques.

# Cast

Le "cast" est l'opération de conversion d'une valeur d'un type de données en un autre type de données. Le cast est nécessaire lorsque vous avez une valeur d'un certain type de données et que vous souhaitez la convertir en un type différent, par exemple, lorsque vous devez effectuer des opérations sur des types incompatibles.

Il existe deux types de cast en Java : explicite et implicite.

Un cast implicite se produit lorsque Java convertit automatiquement un type de données en un autre sans perte d'information. Cela se produit principalement lorsque le type source est plus petit ou plus précis que le type de destination.

```
int entier = 5;  
double reel = entier; // Conversion implicite, int vers double
```

Un cast explicite se produit lorsque vous spécifiez explicitement que vous voulez convertir un type de données en un autre, même si cela peut entraîner une perte d'information.

```
double reel = 7.5;  
int entier = (int) reel; // Conversion explicite, double vers int avec perte de décimales
```

# Autoboxing et Unboxing

Le langage Java intègre des gestionnaires qui effectuent automatiquement la conversion entre les primitives et les classes enveloppantes et vice-versa.

```
int quack = 5;                                // Autoboxing
Integer quackquack = quack;                     // Unboxing
int quackquackquack = quackquack;               // Unboxing

Short tail = 8;                                // Autoboxing
Character p = Character.valueOf('p');           // Unboxing
char paw = p;                                    // Unboxing
Boolean nose = true;                            // Autoboxing
Integer e = Integer.valueOf(9);                 // Unboxing, then implicit casting
long ears = e;
```

```
10: Character elephant = null;                // NullPointerException
11: char badElephant = elephant;             // NullPointerException
```

# Autoboxing et Unboxing

L'autoboxing et l'unboxing s'appliquent aussi aux appels de méthode.

Java va soit caster la valeur, soit l'autoboxer automatiquement, mais pas les deux en même temps.

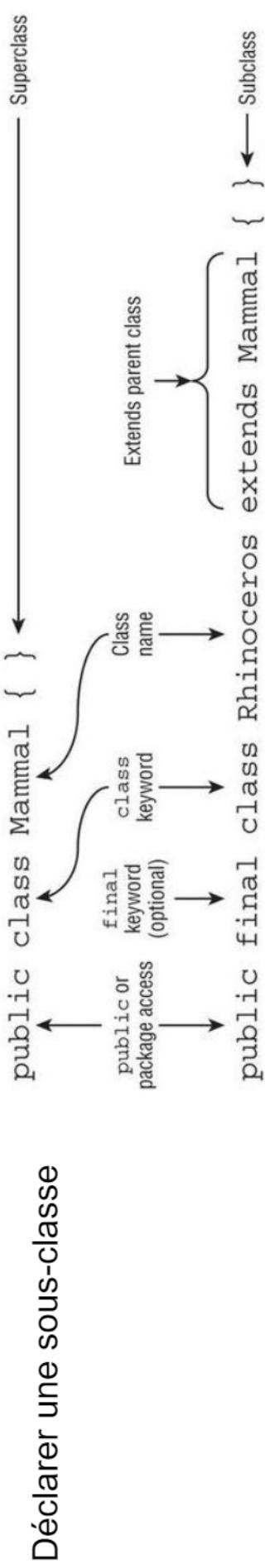
Est-ce que ces deux exemples sont valide ?

```
public class Chimpanzee {  
    public void climb(Long t) {}  
    public void swing(Integer u) {}  
    public void jump(int v) {}  
    public static void main(String[] args) {  
        var c = new Chimpanzee();  
        c.climb(123);  
        c.swing(123);  
        c.jump(123L);  
    }  
}
```

```
public class Gorilla {  
    public void rest(Long x) {  
        System.out.print("long");  
    }  
    public static void main(String[] args) {  
        var g = new Gorilla();  
        g.rest(8);  
    }  
}
```

# Héritage

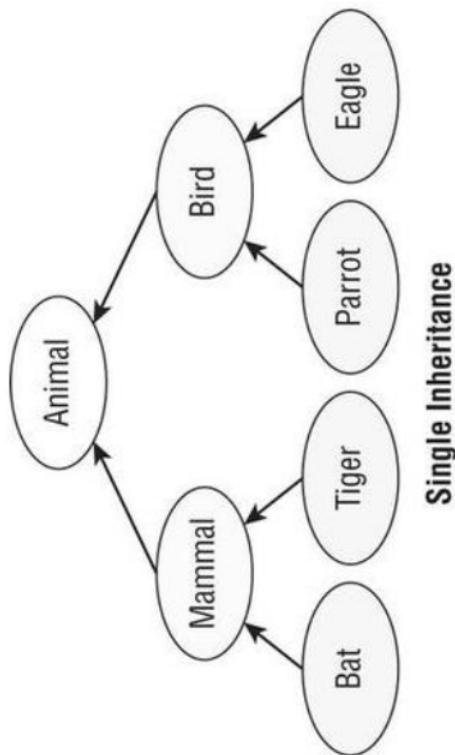
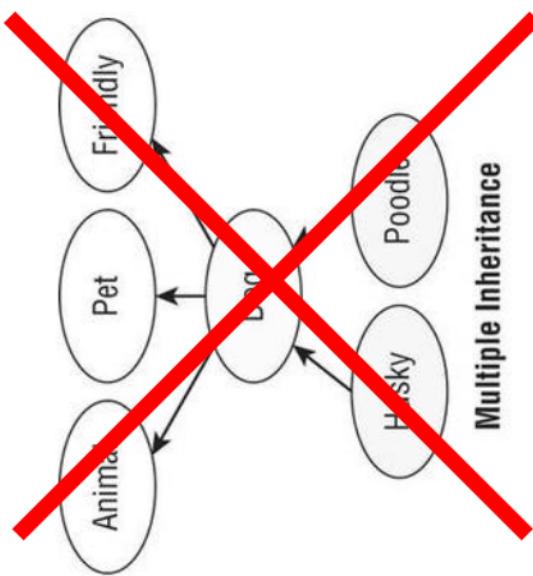
# Comprendre l'héritage



Modifier	Description
final	La classe ne peut pas être étendue.
abstract	La classe est abstraite, peut contenir des méthodes abstraites et nécessite une sous-classe concrète pour être instanciée.
Sealed*	La classe ne peut être étendue que par une liste spécifique de classes.
non-sealed*	Une sous-classe d'une classe scellée autorise des sous-classes potentiellement sans nom.
Static	Utilisé pour les classes statiques imbriquées définies à l'intérieur d'une classe.

# Héritage unique ou multiple

Java prend en charge l'héritage unique, selon lequel une classe ne peut hériter que d'une seule classe parentale directe.

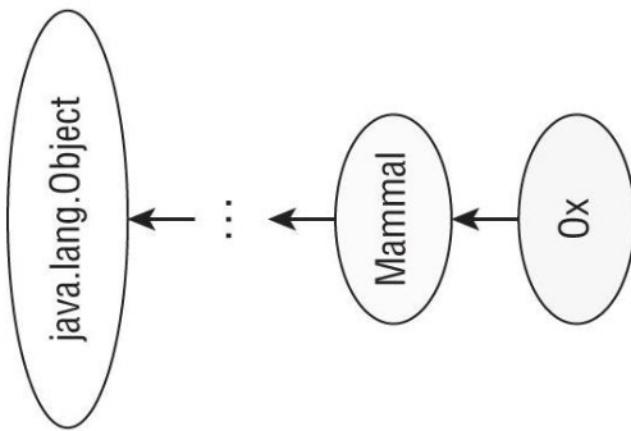


# java.lang.Object

En Java, toutes les classes héritent d'une seule classe : `java.lang.Object`, ou `Object` en abrégé. De plus, `Object` est la seule classe qui n'a pas de classe mère.

Le compilateur insère automatiquement du code dans toutes les classes que vous écrivez et qui n'étendent pas une classe spécifique.

Les types primitifs tels que `int` et `boolean` n'héritent pas de `Object`, puisqu'ils ne sont pas des classes.



All objects inherit `java.lang.Object`

# Les méthodes importantes de Object

La méthode « `boolean equals(Object obj)` » retourne si oui ou non l'objet passé en paramètre est égal à `this`.

Le méthode « `String toString()` » retourne un string qui représente l'objet.

La méthode « `int hashCode()` » return un hash qui représente l'objet. Soit un int unique qui identifie l'objet.

```
import java.util.Objects;

public class Book {
    private String name;
    private String author;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Book book = (Book) o;
        return Objects.equals(name, book.name)
            && Objects.equals(author, book.author);
    }

    @Override
    public String toString() {
        return "Book{" +
            "name=" + name + '\'' +
            ", author='" + author + '\'' +
            '}';
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, author);
    }
}
```

# Déclaration des constructeurs

Ce constructeur créé par Java est appelé constructeur par défaut et est ajouté chaque fois qu'une classe est déclarée sans constructeur.

Le nom du constructeur, Bunny, correspond au nom de la classe, Bunny, et il n'y a pas de type de retour, pas même void.

```
public class Bunny {  
    public Bunny() {  
        System.out.print("hop");  
    }  
}
```

```
public class Bunny {  
    public bunny() {} // DOES NOT COMPILE  
    public void Bunny() {} //not a constructor  
}
```

Comme les paramètres de méthode, les paramètres de constructeur peuvent être n'importe quelle classe valide, n'importe quel tableau ou n'importe quel type primitif.

```
public class Bunny {  
    public Bunny(int size, Object param) {  
    }  
}
```

# Initialisation des objets

Que va afficher cet exemple ?

```
public class Animal {  
    static { System.out.print("A"); }  
}  
  
public class Hippo extends Animal {  
    public static void main(String[] grass) {  
        System.out.print("C");  
        new Hippo();  
        new Hippo();  
        new Hippo();  
    }  
    static { System.out.print("B"); }  
}
```

## Initialiser la classe X

- S'il existe une super-classe Y de X, initialiser d'abord la classe Y.
- Traiter toutes les déclarations de variables statiques dans l'ordre dans lequel elles apparaissent dans la classe.
- Traiter tous les initialiseurs statiques dans l'ordre dans lequel ils apparaissent dans la classe.

# Initialisation des champs *final*

Contrairement aux membres *static* de la classe, les champs d'instance *final* peuvent être définis dans un constructeur. Le constructeur fait partie du processus d'initialisation, il est donc autorisé à assigner des variables d'instance finales. Au moment où le constructeur se termine, toutes les variables d'instance finales doivent se voir attribuer une valeur exactement une fois.

```
public class MouseHouse {  
    private final int volume;  
    private final String type;  
    {  
        this.volume = 10;  
    }  
    public MouseHouse(String type) {  
        this.type = type;  
    }  
    public MouseHouse() {  
        // DOES NOT COMPILE  
        this.volume = 2; // DOES NOT COMPILE  
    }  
}
```

```
public class MouseHouse {  
    private final int volume;  
    private final String name;  
    public MouseHouse() {  
        this.name = "Empty House"; // Constructor assignment  
    }  
    {  
        volume = 10; // Instance initializer assignment  
    }  
}
```

# Surcharge d'une méthode

Surcharge classique :

- La méthode de la classe enfant doit avoir la même signature que la méthode de la classe mère.
  - La méthode de la classe enfant doit être au moins aussi accessible que la méthode de la classe mère.
  - Si la méthode renvoie une valeur, celle-ci doit être la même ou un sous-type de la méthode de la classe mère, ce que l'on appelle les types de retour covariants.
  - La méthode de la classe enfant ne peut pas déclarer une checked exception qui est nouvelle ou plus large de toute les exceptions déclarées dans la méthode de la classe mère.\*
- ```
public class Marsupial {  
    public double getAverageWeight() {  
        return 50;  
    }  
}  
public class Kangaroo extends Marsupial {  
    public double getAverageWeight() {  
        return super.getAverageWeight() + 20;  
    }  
}  
public static void main(String[] args) {  
    System.out.println(new  
        Marsupial().getAverageWeight()); // 50.0  
    System.out.println(new  
        Kangaroo().getAverageWeight()); // 70.0  
}
```

# Annotation @Override

Vous pouvez utiliser l'annotation @Override pour indiquer au compilateur que vous tentez de surcharger une méthode.

```
public class Fish {  
    public void swim() {};  
}  
public class Shark extends Fish {  
    @Override  
    public void swim(int speed) {}; // DOES NOT COMPILE  
}
```

```
public class Fish {  
    public void swim() {};  
}  
public class Shark extends Fish {  
    @Override  
    public void swim(int speed) {}; // DOES NOT COMPILE  
}
```

# Écrire des méthodes *final*/

Les méthodes finales ne peuvent pas être surchargées.

```
public class Bird {  
    public final boolean hasFeathers() {  
        return true;  
    }  
    public final static void flyAway() {}  
  
}  
  
public class Penguin extends Bird {  
    public final boolean hasFeathers() { // DOES NOT COMPILE  
        return false;  
    }  
    public final static void flyAway() {} // DOES NOT COMPILE  
}
```

# Création d'une classe abstraite

Une classe abstraite est une classe déclarée avec le modificateur abstrait qui ne peut pas être instanciée directement et qui peut contenir des méthodes abstraites.

- Seules les méthodes d'instance peuvent être marquées comme abstraites dans une classe, et non les variables, les constructeurs ou les méthodes statiques.
- Une méthode abstraite ne peut être déclarée que dans une classe abstraite.
- Une classe non abstraite qui étend une classe abstraite doit implémenter toutes les méthodes abstraites héritées.
- La surcharge d'une méthode abstraite suit les règles existantes de surcharge des méthodes vu plus tôt.

```
public abstract class Canine {}  
  
public class Wolf extends Canine {}  
  
public class Fox extends Canine {}  
  
public class Coyote extends Canine {}
```

# Déclarer des méthodes abstraites

Une méthode abstraite est toujours déclarée sans corps.

Elle comprend également un point-virgule ( ; ) après la déclaration de la méthode.

Une classe abstraite peut inclure des méthodes non abstraites.

En fait, une classe abstraite peut inclure tous les mêmes membres qu'une classe non abstraite, y compris les variables, les méthodes statiques et d'instance, les constructeurs, etc.

```
public abstract class Mammal {  
    abstract void showHorn();  
    abstract void eatLeaf();  
}  
  
public abstract class Rhino extends Mammal {  
    void showHorn() {} // Inherited from Mammal  
}  
  
public class BlackRhino extends Rhino {  
    void eatLeaf() {} // Inherited from Mammal  
}
```

```
public abstract class Animal {  
    abstract String getName();  
}  
  
public abstract class BigCat extends Animal {  
    protected abstract void roar();  
}  
  
public class Lion extends BigCat {  
    public String getName() {  
        return "Lion";  
    }  
    public void roar() {  
        System.out.println("The Lion lets out a loud ROAR!");  
    }  
}
```

# Modificateurs `abstract` et `final`

Terme incompatible ! Java ne permet pas à une classe ou à une méthode d'être marquée à la fois comme abstraite et comme finale. Par exemple, l'extrait de code suivant ne sera pas compilé :

```
public abstract final class Tortoise { // DOES NOT COMPILE
    public abstract final void walk(); // DOES NOT COMPILE
}
```

# Modificateurs `abstract` et `final`

Une méthode ne peut pas être à la fois abstraite et privée.

```
public abstract class Whale {  
    private abstract void sing(); // DOES NOT COMPILE  
}  
  
public class HumpbackWhale extends Whale {  
    private void sing() {  
        System.out.println("Humpback whale is singing");  
    }  
}
```

# Modificateur `abstract` et `static`

Une méthode statique ne peut être que cachée, et non surchargée.

Elle est définie comme appartenant à la classe, et non comme une instance de la classe.

Si une méthode statique ne peut pas être remplacée, il s'ensuit qu'elle ne peut pas non plus être qualifiée d'abstraite puisqu'elle ne peut jamais être implémentée

```
abstract class Hippopotamus {  
    abstract static void swim(); // DOES NOT COMPILE  
}
```

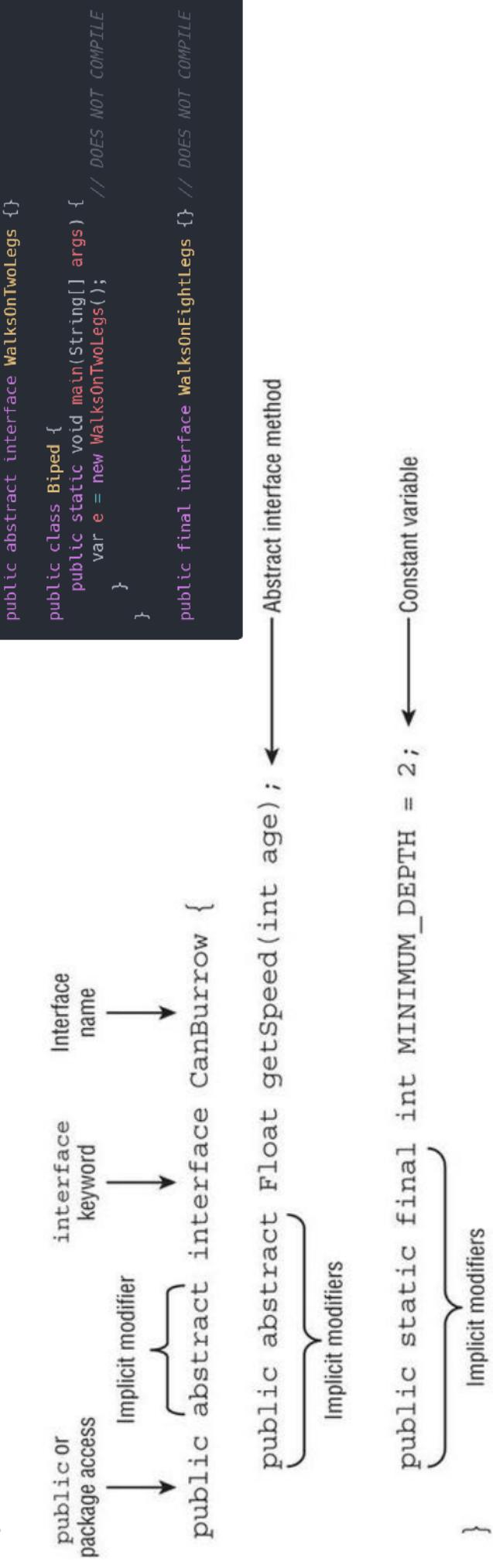
# Interfaces

# Implémenter des interfaces

Comme les classes ne peuvent étendre qu'une seule classe, leur utilisation pour l'héritage est limitée.

En revanche, une classe peut implémenter un nombre quelconque d'interfaces.

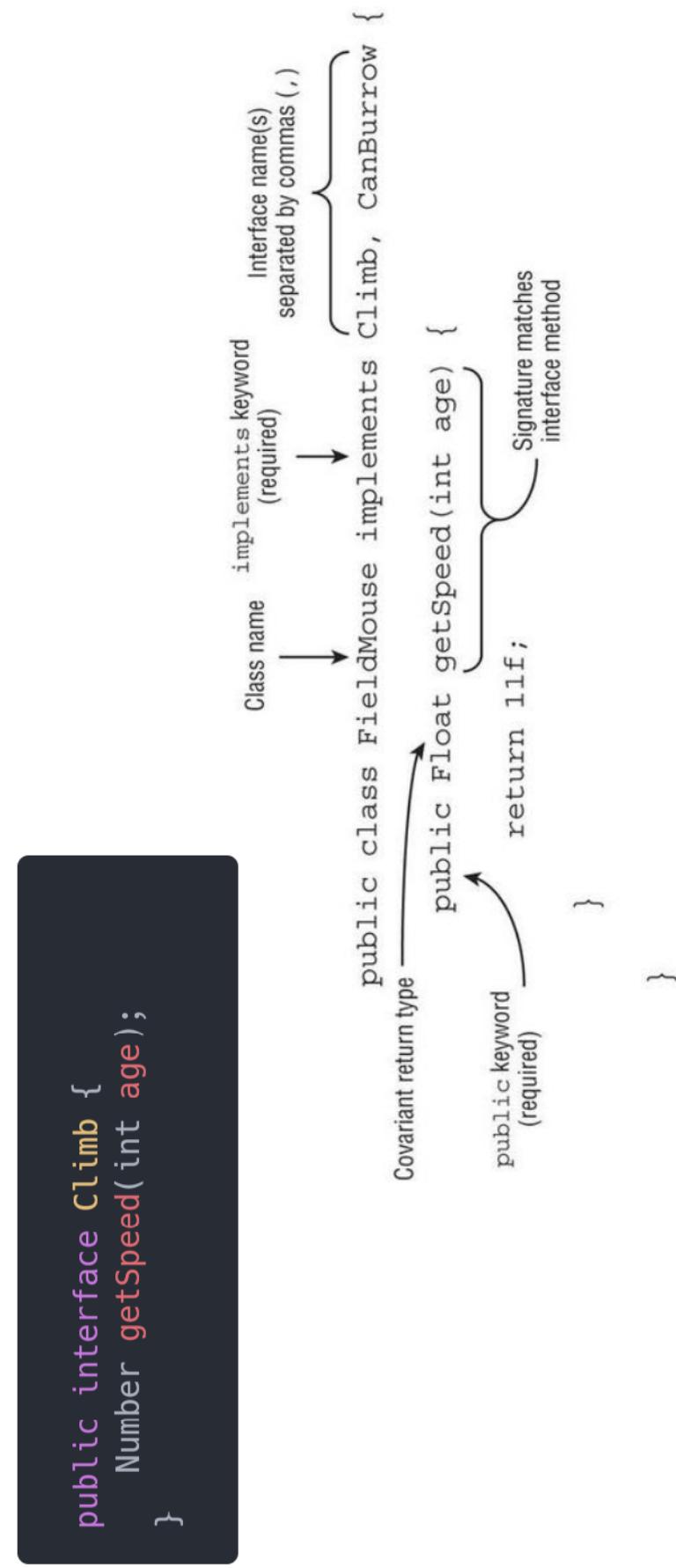
Une interface est un type de données abstraites qui déclare une liste de méthodes abstraites que toute classe implémentant l'interface doit fournir.



# Exemple

La signature de la méthode `getSpeed()` correspond exactement, et le type de retour est covariant, puisqu'un `Float` peut être implicitement casté en `Number`.

```
public interface Climb {  
    Number getSpeed(int age);  
}
```



# Héritage entre interfaces

Comme une classe, une interface peut étendre une autre interface à l'aide du mot-clé `extends`.

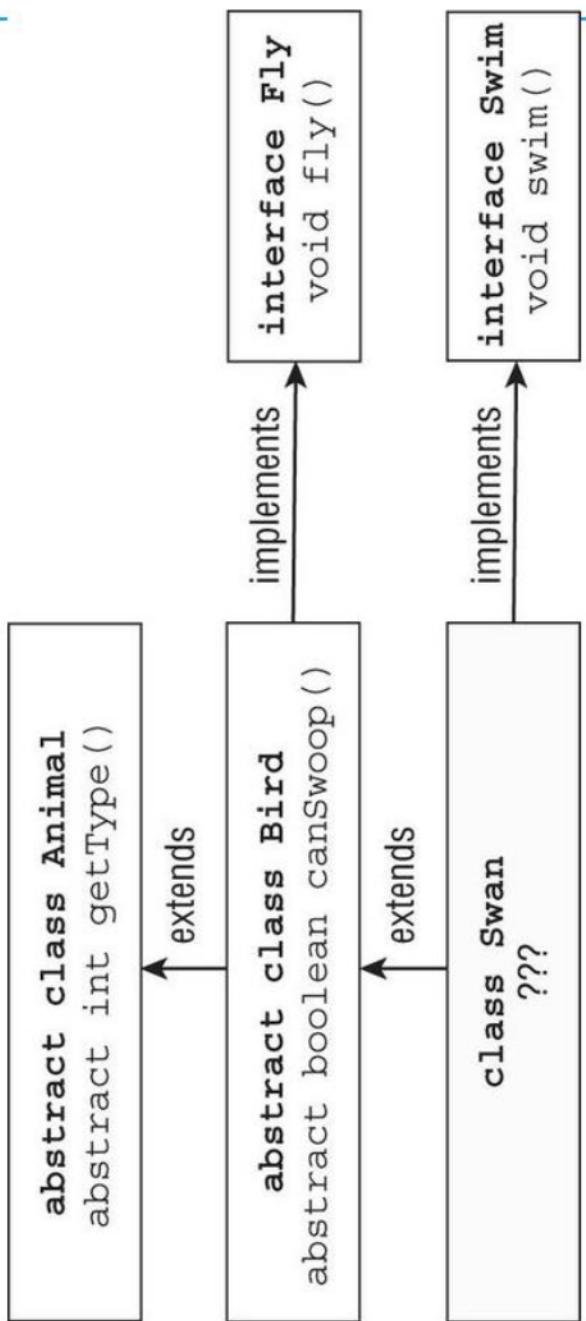
```
public interface Nocturnal {}  
  
public interface HasBigEyes extends Nocturnal {}
```

Contrairement à une classe, qui ne peut étendre qu'une seule classe, une interface peut étendre plusieurs interfaces.

```
public interface Nocturnal {  
    public int hunt();  
}  
  
public interface CanFly {  
    public void flap();  
}  
  
public interface HasBigEyes extends Nocturnal, CanFly {}  
  
public class Owl implements HasBigEyes {  
    public int hunt() { return 5; }  
    public void flap() { System.out.println("Flap!"); }  
}
```

# Héritage entre interfaces

Comme pour une classe abstraite, lorsqu'une classe concrète hérite d'une interface, toutes les méthodes abstraites héritées doivent être implémentées.



La classe concrète `Swan` hérite de quatre méthodes abstraites qu'elle doit mettre en œuvre : `getType()`, `canSwoop()`, `fly()` et `swim()`.

# Insertion des modificateurs implicites

Comme indiqué précédemment, un modificateur implicite est un modificateur que le compilateur insère automatiquement.

Cela rappelle le fait que le compilateur insère un constructeur sans argument par défaut si vous ne définissez pas de constructeur.

La liste suivante comprend les modificateurs implicites pour les interfaces :

- Les interfaces sont implicitement abstraites.
- Les variables d'interface sont implicitement publiques, statiques et finales.
- Les méthodes d'interface sans corps sont implicitement abstraites.
- Les méthodes d'interface sans modificateur private sont implicitement publiques.

# Insertion des modificateurs implicites

Exemple d'équivalence

```
public interface Soar {  
    int MAX_HEIGHT = 10;  
    final static boolean UNDERWATER = true;  
    void fly( int speed );  
    abstract void takeoff();  
    public abstract double dive();  
}
```

```
public abstract interface Soar {  
    public static final int MAX_HEIGHT = 10;  
    public final static boolean UNDERWATER = true;  
    public abstract void fly( int speed );  
    public abstract void takeoff();  
    public abstract double dive();  
}
```

# Différence entre classe abstraite et interface

Bien que les classes abstraites et les interfaces soient toutes deux considérées comme des types abstraits, seules les interfaces utilisent des modificateurs implicites.

```
abstract class Husky { // abstract required in class declaration
    abstract void play(); // abstract required in method declaration
}

interface Poodle { // abstract optional in interface declaration
    void play(); // abstract optional in method declaration
}
```

# Les collections

# Les collections

Les collections sont des objets qui permettent de gérer des ensemble d'objet. Elles permettent de faire les opérations les plus courantes sur les ensembles comme par exemple : insérer, récupérer, trier, filtrer, classer...

L'API Java collection regroupe plusieurs interfaces et classes. Voici les 4 grandes familles de collection :

- List : collection d'éléments ordonnés qui accepte les doublons
- Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
- Map : collection sous la forme d'une association de paires clé/valeur
- Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement\*

# ArrayList (implémentation de List)

- ArrayList est l'implémentation la plus courante de List en Java. Elle est optimisée pour les accès aux éléments individuels, ce qui la rend idéale pour les applications qui nécessitent un accès aléatoire aux éléments d'une liste.
- ArrayList est une implémentation basée sur un tableau. Cela signifie que les éléments d'une ArrayList sont stockés dans un tableau. Lorsque l'on ajoute un élément à une ArrayList, le tableau est automatiquement agrandi si nécessaire.

```
// Création d'une ArrayList de chaînes de caractères
ArrayList<String> listeDeMots = new ArrayList<>();

// Ajout d'éléments à la liste
listeDeMots.add("Pomme");
listeDeMots.add("Banane");
listeDeMots.add("Fraise");
listeDeMots.add("Orange");

// Affichage de la liste
System.out.println("Contenu de la liste : " + listeDeMots);

// Accès à un élément par index
String premierMot = listeDeMots.get(0);
System.out.println("Le premier mot est : " + premierMot);

// Modification d'un élément
listeDeMots.set(2, "Cerise"); // Remplace le troisième mot par "Cerise"
System.out.println("Contenu de la liste après modification : " + listeDeMots);
```

# ArrayList (implémentation de List)

```
// Suppression d'un élément par valeur
ListeDeMots.remove("Banane"); // Supprime "Banane"
System.out.println("Contenu de la liste après suppression : " + listeDeMots);

// Taille de la liste
int taille = listeDeMots.size();
System.out.println("Taille de la liste : " + taille);

// Vérifier si un élément existe dans la liste
boolean existe = listeDeMots.contains("Fraise");
System.out.println("Est-ce que 'Fraise' existe dans la liste ? " + existe);

// Parcourir et afficher tous les éléments de la liste
System.out.println("Parcours de la liste :");
for (String mot : listeDeMots) {
    System.out.println(mot);
}
```

# LinkedList (implémentation de List)

```
// Créeation d'une LinkedList de chaînes de caractères
LinkedList<String> listeChainee = new LinkedList<>();

// Ajout d'éléments à la liste
listeChainee.add("Pomme");
listeChainee.add("Banane");
listeChainee.add("Fraise");
listeChainee.add("Orange");

// Affichage de la liste
System.out.println("Contenu de la liste chaînée : " + listeChainee);

// Accès et modification d'éléments
String premierElement = listeChainee.getFirst();
System.out.println("Le premier élément est : " + premierElement);

listeChainee.set(2, "Cerise"); // Remplace le troisième élément par "Cerise"
System.out.println("Contenu de la liste après modification : " + listeChainee);

// Suppression d'éléments
listeChainee.removeLast(); // Supprime le dernier élément
System.out.println("Contenu de la liste après suppression : " + listeChainee);

// Taille de la liste
int taille = listeChainee.size();
System.out.println("Taille de la liste : " + taille);

// Parcourir et afficher tous les éléments de la liste
System.out.println("Parcours de la liste :");
for (String element : listeChainee) {
    System.out.println(element);
}
```

- LinkedList est une structure de données linéaire dans Java qui représente une liste chaînée.
- Contrairement à un tableau (ArrayList), une LinkedList ne stocke pas les éléments consécutifs en mémoire, mais chaque élément est lié au suivant par un lien. Cela permet des opérations d'insertion et de suppression rapides, mais l'accès à un élément par index est généralement moins efficace que dans un ArrayList.

# Set

Set est une interface faisant partie du framework de collections, qui représente un ensemble d'éléments uniques, c'est-à-dire une collection d'objets où chaque élément ne peut apparaître qu'une seule fois. Les classes qui implémentent l'interface Set garantissent qu'aucun doublon ne sera stocké, ce qui en fait un choix idéal pour la suppression des éléments en double ou pour vérifier l'unicité des éléments.

Les principales classes qui implémentent l'interface Set dans Java sont HashSet, LinkedHashSet, et TreeSet :

- HashSet : Cette classe stocke les éléments dans un ordre non spécifique. Elle offre une recherche et une insertion très rapides, mais l'ordre des éléments n'est pas garanti.
- LinkedHashSet : Cette classe stocke les éléments dans l'ordre d'insertion. Elle combine les avantages d'un HashSet (recherche rapide) avec un suivi de l'ordre d'insertion.
- TreeSet : Cette classe stocke les éléments dans un ordre trié (par ordre naturel ou en utilisant un comparateur fourni). Elle est adaptée lorsque vous avez besoin des éléments triés.

# Exemple avec HashSet

```
// Création d'un ensemble HashSet de chaînes de caractères
Set<String> ensemble = new HashSet<>();

// Ajout d'éléments à l'ensemble
ensemble.add("Pomme");
ensemble.add("Banane");
ensemble.add("Fraise");
ensemble.add("Orange");

// Affichage de l'ensemble (l'ordre n'est pas garanti)
System.out.println("Contenu de l'ensemble : " + ensemble);

// Vérification de l'appartenance d'un élément
boolean contientPomme = ensemble.contains("Pomme");
System.out.println("L'ensemble contient Pomme : " + contientPomme);

// Suppression d'un élément
ensemble.remove("Banane");
System.out.println("Contenu de l'ensemble après suppression : " + ensemble);

// Taille de l'ensemble
int taille = ensemble.size();
System.out.println("Taille de l'ensemble : " + taille);
```

# Map

En Java, Map est une interface faisant partie du framework de collections qui représente une structure de données permettant de stocker des paires clé-valeur. Chaque élément dans une Map est une association d'une clé (key) et d'une valeur (value). Les clés sont uniques, c'est-à-dire qu'il ne peut y avoir qu'une seule entrée pour chaque clé dans une Map, mais les valeurs peuvent être en double.

L'interface Map est largement utilisée pour stocker, rechercher et récupérer des données basées sur une clé. Voici les classes les plus couramment utilisées pour implémenter l'interface Map :

- **HashMap** : Cette classe stocke les paires clé-valeur sans garantie d'ordre. Elle est efficace pour la plupart des opérations courantes.
- **LinkedHashMap** : Cette classe stocke les paires clé-valeur dans l'ordre d'insertion. Elle est utile lorsque vous avez besoin de conserver l'ordre d'insertion.
- **TreeMap** : Cette classe stocke les paires clé-valeur dans un ordre trié, basé sur l'ordre naturel des clés ou un comparateur fourni. Elle est appropriée lorsque vous avez besoin de maintenir les clés triées.
- **Hashtable** : Il s'agit d'une version plus ancienne de HashMap. Elle est thread-safe, mais son utilisation est déconseillée au profit de ConcurrentHashMap dans les applications multithreadées.
- **ConcurrentHashMap** : Cette classe est similaire à HashMap, mais elle est conçue pour la concurrence, ce qui signifie qu'elle peut être utilisée en toute sécurité en toute concurrence, ce qui signifie qu'elle peut être utilisée en toute sécurité par plusieurs threads simultanément.

# Exemple avec HashMap

```
// Création d'une HashMap de noms (clé) et âges (valeur)
Map<String, Integer> map = new HashMap<>();

// Ajout d'éléments à la Map
map.put("Alice", 25);
map.put("Bob", 30);
map.put("Eve", 28);

// Accès aux éléments par clé
int ageAlice = map.get("Alice");
System.out.println("Âge d'Alice : " + ageAlice);

// Vérification de l'existence d'une clé
boolean contientBob = map.containsKey("Bob");
System.out.println("La Map contient Bob : " + contientBob);

// Parcours de la Map
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Clé : " + entry.getKey() + ", Valeur : " + entry.getValue());
}

// Suppression d'un élément par clé
map.remove("Eve");
System.out.println("Contenu de la Map après suppression : " + map);

// Taille de la Map
int taille = map.size();
System.out.println("Taille de la Map : " + taille);
```



# Exercices