Langage de programmation JAVA

AMMOR Fatimazahra




Programme

I Introduction & structure

- **Environnement Java** : compilation & interprétation, JDK, JVM, JRE
- **Structure d'une classe Java** : attributs, méthodes, classes et fichier source, écriture d'une fonction main
- **Packages** : déclaration, importation
- **Création des objets :** constructeurs, lectures & écriture des attributs, exécution des instances, ordre d'initialisation
- **Variables :** types de données, déclaration des variables, initialisations des variables, suppression des variables
- **Opérateurs :** types, affectation des valeurs, caste
- **Méthodes :** design d'une méthode, déclaration des variables locales et d'instance, Varargs, accès au données statiques, passage des paramètres, surcharge des méthodes (overloading)

II Programmation O. O. avec Java

- **Héritage** : déclaration des sous-classes, héritage unique vs l'héritage multiple, déclarations des constructeurs, initialisation des objets, héritage des attributs
- **Classes abstrait:** création, méthodes abstraits, utilisation
- **Classes immutables**
- **Interfaces :** déclaration, héritage, méthodes, implémentations des interfaces
- **Enum :** déclaration, utilisation dans switch, insertion des constructeurs, méthodes et attributs
- **Polymorphisme** : objet vs référence, caste des objets, polymorphisme vs overiding, overiding vs hiding attributs
- **Tableau** : déclaration, passage des paramètres, accès, utilisation
- **Collections** : list, set, queu, map, trie des données

III Exceptions

- **Introduction au concept :**types des exceptions, throw & throws, méthodes(appel, redéfinition), affichage d'une exception
- **Les classes d'exception**
- **Gestion des exceptions**
- **Exceptions personnalisées**

IV Expressions LAMBDA

- **Ecriture d'une expression lambda**
- **Coder une interface fonctionnelle**
- **Les références des méthodes**
- **Utilisation des interfaces fonctionnelles intégrées**

Programme

V Collections Vs streams

- **Obtaining a Stream From a Collection**
- **Stream Processing Phases**
- **Stream.filter()**
- **Stream.map()**
- **Stream.collect()**
- **Stream.min() and Stream.max()**
- **Stream.count()**
- **Stream.reduce()**

VI Entrées / sorties

- **Référencements des fichiers & répertoires**
- **Introduction au I/O streams**
- **Lecture & écriture**
- **Sérialisation des données**
- **Interaction avec l'utilisateur**

VII Généricité

- **Introduction à la généricité en Java**
- **Apprendre à coder une classe générique**
- **Définition de méthodes génériques**

VIII Interfaces graphiques

- **introduction,**
- **Création de fenêtres et composants de base :** Jframe, Jpanel, JButton (bouton), JLabel (étiquette), JTextField (champ de texte), JTextArea (zone de texte)
- **Disposition des composants (Layouts) :** FlowLayout, BorderLayout ,GridLayout
- **Gestion des événements**
- **Composants avancés et boîtes de dialogue**
- **Gestion de l'apparence et personnalisation**
- **Accès à une base de données**

Introduction et structure du langage JAVA




Historique JAVA

•**Lancement officiel (1995) par Sun Microsystems** : Java a été officiellement lancé en 1995 avec la sortie de **Java 1.0**. À cette époque, l'une des fonctionnalités les plus novatrices était l'introduction de la **machine virtuelle Java (JVM)**, qui permettait aux programmes d'être écrits une seule fois et exécutés n'importe où (concept de **"Write Once, Run Anywhere"**).
•Évolution : Java 1, Java 2, Java SE, Java EE, etc.
•**Java 5 (2004)** : Java 5 a introduit de nombreuses fonctionnalités importantes telles que les **génériques**, les **annotations**, l'**énumération (enum)**, et la **boucle for-each**.
•**Java 8 (2014)** : Considérée comme l'une des versions les plus importantes. Elle a introduit les **expressions lambda**, les **Streams**, et l'API **Date and Time**.
•**Java 17 (2021)** : Une autre version LTS qui a renforcé la stabilité et la performance, en introduisant de nouvelles fonctionnalités comme les **patterns pour instanceof**, et l'**aperçu des classes scellées**.

Historique JAVA

Java a un écosystème vaste et diversifié. Voici quelques composantes majeures :

- **JDK (Java Development Kit)** : Le JDK inclut la JVM, le compilateur javac, et d'autres outils de développement. Il est nécessaire pour développer des applications Java.
- **JRE (Java Runtime Environment)** : Le JRE inclut la JVM et les bibliothèques standard nécessaires pour exécuter des programmes Java, mais pas pour les développer.
- **Java EE (Enterprise Edition)** : Une extension de Java SE pour le développement d'applications d'entreprise, incluant des outils pour le développement web, la gestion des transactions, la messagerie, etc.
- **Frameworks populaires** :
    - **Spring** : Framework pour le développement d'applications web et d'entreprise.
    - **Hibernate** : Framework pour la persistance des données (ORM).
    - **Apache Maven/Gradle** : Outils de gestion de projets et de dépendances.
- **Utilisation** : applications web, mobiles, serveurs

Avantages JAVA

1. Portabilité (Write Once, Run Anywhere) :
Code Java compilé en bytecode, exécuté sur n'importe quelle plateforme avec la JVM.
Le code Java est d'abord compilé en un **bytecode** qui est interprété par la **JVM**. Ce bytecode est indépendant de l'architecture matérielle, ce qui rend Java **portable** sur différentes plateformes.
Exemples : applications desktop, web, mobiles.
2. Programmation Orientée Objet (POO) :
Facilite la réutilisation du code, la modularité, et l'organisation.
Concepts comme l'héritage, le polymorphisme, et l'encapsulation favorisent une bonne conception logicielle.
3. Large Écosystème et Bibliothèques :
Richesse de bibliothèques et frameworks (Spring, Hibernate) qui simplifient le développement.
Accès à des outils de développement puissants et des communautés actives.
4. Facilité d'Apprentissage :
Syntaxe claire et similaire à d'autres langages populaires (C++, C#), ce qui facilite l'apprentissage pour les débutants.
Documentation riche et nombreuses ressources d'apprentissage disponibles.

Avantages JAVA

5. Sécurité

5.1. Modèle de Sécurité :

- Java utilise un modèle de sécurité basé sur les permissions. Chaque application Java s'exécute dans un environnement contrôlé (sandbox) qui limite ses capacités.
- Permet de restreindre l'accès aux ressources critiques (fichiers, réseau).

5.2. Cryptographie :

- Bibliothèques robustes (Java Cryptography Architecture) pour la gestion de la cryptographie, du hachage et de la sécurisation des données.

5.3. Sécurité des Applets :

- Les applets Java, lorsqu'elles étaient populaires, fonctionnaient dans un environnement sécurisé pour protéger les utilisateurs.

5.4. Gestion des Exceptions :

- Gestion des erreurs et exceptions intégrée, permettant de capturer les anomalies sans compromettre la sécurité du système.
  Exemple : Accès aux fichiers
  Code : Utilisation de java.nio.file pour lire un fichier en toute sécurité.

```
import java.nio.file.*;
import java.io.IOException;

public class SecureFileRead {
    public static void main(String[] args) {
        Path path = Paths.get("secret.txt");
        try {
            String content = Files.readString(path);
            System.out.println(content);
        } catch (IOException e) {
            System.err.println("Erreur d'accès au fichier : " +
e.getMessage());
        }
    }
}
```

Avantages JAVA

6. Multi-threading

6.1. Support intégré :

- Java offre un support intégré pour le multi-threading via la classe Thread et l'interface Runnable.
- Permet la création de plusieurs threads qui s'exécutent simultanément, améliorant l'efficacité des applications.

6.2. Exécution Réactive :

- Les applications peuvent rester réactives même lors de tâches longues, comme le téléchargement ou le traitement de données, car le travail peut être délégué à des threads distincts.

6.3. Gestion des Threads :

- Java fournit des mécanismes de synchronisation (synchronized, wait, notify) pour éviter les conflits d'accès aux ressources partagées.
- Exemples : applications serveurs, jeux en temps réel.

Exemple : Téléchargement Simultané

Code : Création de plusieurs threads pour simuler des téléchargements.

```
public class DownloadTask extends Thread {
    private String fileName;

    public DownloadTask(String fileName) {
        this.fileName = fileName;
    }

    public void run() {
        System.out.println("Téléchargement de " + fileName + " démarré...");
        // Simuler le temps de téléchargement
        try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {}
        System.out.println("Téléchargement de " + fileName + " terminé.");
    }

    public static void main(String[] args) {
        DownloadTask task1 = new DownloadTask("fichier1.zip");
        DownloadTask task2 = new DownloadTask("fichier2.zip");
        task1.start();
        task2.start();
    }
}
```

Avantages JAVA

7. Performances

7.1. Compilation Just-In-Time (JIT) :

- La JVM compile le bytecode en code machine natif lors de l'exécution, ce qui améliore considérablement les performances.
- Cette compilation à la volée optimise l'exécution en analysant le code en temps réel.

7.2. Garbage Collection :

- Java gère la mémoire automatiquement via un ramasse-miettes (Garbage Collector), réduisant le risque de fuites de mémoire et simplifiant le développement.
- Différentes stratégies de GC disponibles (ex. G1, ZGC) pour optimiser les performances selon les besoins.

7.3. Bibliothèques Optimisées :

- Utilisation de bibliothèques hautement optimisées (par exemple, les collections Java) qui sont conçues pour des performances élevées dans des scénarios courants.

7.4. Performances Comparées :

- Bien que Java soit généralement moins performant que les langages compilés natifs (comme C/C++), les améliorations de la JVM et des algorithmes optimisés permettent de réduire cet écart.

Exemple : Compilation JIT

•**Code :** Comparaison de l’exécution de deux boucles pour montrer l'effet de la compilation JIT.

```
public class PerformanceTest {
    public static void main(String[] args) {
        long startTime = System.nanoTime();
        for (int i = 0; i < 1_000_000; i++) {
            Math.sqrt(i);
        }
        long duration = System.nanoTime() - startTime;
        System.out.println("Durée d'exécution : " + duration + "
nanosecondes");
    }
}
```

Structure d’un programme JAVA

Modificateurs d’accès : **public**, **private**, **protected** et **default**
Public signifie que notre classe ainsi que notre méthode sont accessibles de partout.

String[] args représente un tableau de chaînes de caractères : passer des arguments au programme lors de son exécution depuis la ligne de commande pour une exécution personnalisée.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Méthode main() : point d’entrée de tout programme, c’est là où commence l’exécution

Le mot-clé **static** signifie que cette méthode appartient à la classe elle-même et non à une instance particulière de la classe.
C’est important que la méthode soit static pour que la JVM puisse l’appeler dès le départ sans instanciation (avant création d’objets).
Si la méthode n’est pas static, il faudrait instancier un objet pour l’appeler

Void : ne retourne aucune valeur


*
* *
* *
* *
****
* *
* *
* *
Process finished with exit code 0

Structure d’un programme JAVA

Voici ce qui se passe lorsqu'un programme Java est exécuté :

1. Chargement de la classe : La JVM charge la classe contenant la méthode main().
2. Appel de la méthode main() : La JVM appelle la méthode main(), qui exécute le code qu'elle contient.
3. Exécution : Le programme s'exécute ligne par ligne à partir de la méthode main(), jusqu'à ce qu'il atteigne la fin de cette méthode.

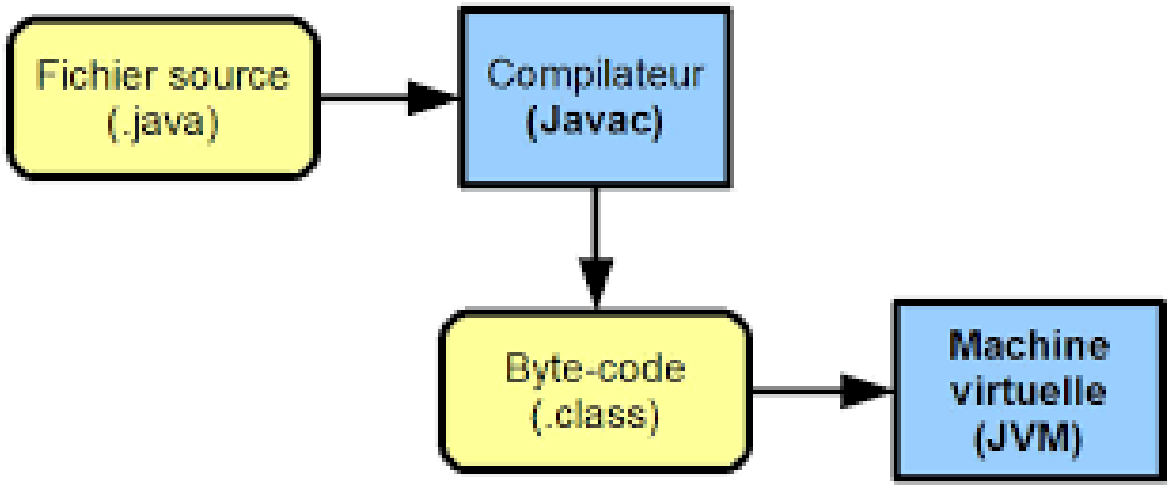
Si la méthode main() ne peut pas être trouvée (par exemple, si elle n’est pas correctement définie ou si elle est manquante), une erreur est générée, car la JVM ne sait pas par où commencer l’exécution.

Ici, la méthode main() est simple. Elle appelle la méthode System.out.println() pour afficher du texte dans la console.
Lors de l'exécution de ce programme, la JVM appelle la méthode main(), et le message "Hello, World!" est affiché à l'écran.

Dans les applications plus complexes, la méthode main() sert souvent à initialiser le programme. Une fois initialisé, elle appelle d'autres méthodes ou objets pour réaliser les différentes fonctionnalités.

Compilation et exécution

- **Compilation :** Utilisation du JDK (javac)
- **Exécution :** Utilisation de la JVM (java)



Environnement et développement

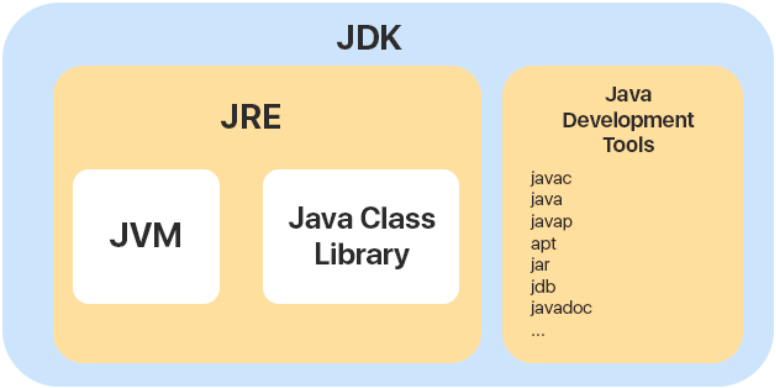
- Un environnement de développement intégré (IDE) est une application logicielle qui aide les programmeurs à développer efficacement le code logiciel.
  - La vérification de la syntaxe du langage ;
  - La détection des erreurs ;
  - L'auto-complétion ;
  - Un compilateur ;
  - Un serveur.

Atom

IntelliJ

Visual Studio Code

- Installation et configuration du JDK



Conventions de Nommage

Classes
Style : CamelCase
Règle : Chaque mot commence par une majuscule.
Exemples : Student, EmployeeDetails, CarModel

Variables
Style : camelCase
Règle : Comme pour les méthodes, le premier mot est en minuscule.
Exemples : studentCount, firstName, totalAmount

Constantes
Style : UPPER_SNAKE_CASE
Règle : Tous les caractères sont en majuscules, les mots sont séparés par des underscores
Exemples : MAX_VALUE, DEFAULT_TIMEOUT, PI_VALUE

Méthodes
Style : camelCase
Règle : Le premier mot commence par une minuscule, les mots suivants par une majuscule.
Exemples : calculateTotal, getEmployeeName, setAge

Package
Style : lettres minuscules
Règle : Les noms de package doivent être courts et sans espaces, souvent basés sur le nom de domaine inversé.
Exemples : com.example.project, org.mycompany.utilities

Interfaces
Style : CamelCase
Règle : Souvent, les noms d'interfaces se terminent par "able" ou "ible".
Exemples : Runnable, Serializable, Comparable

Programmation Orientée Objet avec JAVA




Qu’est-ce que la POO?

Définitions	Principes fondamentaux	Vs Programmation Impérative
- La Programmation Orientée Objet est un paradigme de programmation qui utilise "objets" pour modéliser des concepts du monde réel. - Chaque objet est une instance d'une classe, qui définit ses caractéristiques (attributs) et comportements (méthodes).	**- Modularité :** Le code est organisé en modules (classes) indépendants, facilitant la gestion et la compréhension. **- Réutilisabilité :** Les classes peuvent être réutilisées dans différents programmes, réduisant le besoin de réécriture de code. **- Abstraction :** La POO permet de cacher les détails d'implémentation, ne montrant que l'interface publique des objets. Cela simplifie l'interaction avec des systèmes complexes.	- En programmation impérative, le code est structuré autour de fonctions et de séquences d'instructions. - La POO se concentre sur les objets, rendant le code plus intuitif et proche de la réalité.

Exemples de la Vie Réelle :

- Modéliser des objets comme Voiture, Animal, CompteBancaire.
- Chaque objet a des propriétés (ex. couleur, taille) et des actions (ex. conduire, manger).

Programmation orientée objet avec JAVA

1- Classes Vs Objets




Classes Vs Objets

- Une classe est une blueprint (plan ou modèle) pour créer des objets. - Elle définit les attributs (variables d'instance) et les méthodes (comportements) que les objets de ce type auront.	**- Un objet est une instance d'une classe. - Chaque objet a ses propres valeurs pour les attributs définis dans la classe.**

// Définition d'une classe
public class Voiture {
  // Attributs (variables d'instance)
  String marque;
  String couleur;
  int vitesseMax;
 	
  // Méthodes (comportements)
  void demarrer() {
  System.out.println("La voiture démarre.");
  }
  void accelerer() {
  System.out.println("La voiture accélère.");
  }
}
	// Création d'un objet à partir de la classe Voiture
	Voiture maVoiture = new Voiture();
	
	// Assignation des valeurs aux attributs
	maVoiture.marque = « Citroen »;
	maVoiture.couleur = « Rouge »;
	maVoiture.vitesseMax = 180;
	
	// Appel des méthodes sur l'objet
	maVoiture.demarrer(); // Affiche : La voiture démarre.
	maVoiture.accelerer(); // Affiche : La voiture accélère.

Voiture est une **classe** qui définit des attributs comme marque, couleur, et vitesseMax, ainsi que des méthodes comme demarrer() et accelerer().	maVoiture est un **objet** de type Voiture avec des valeurs spécifiques pour ses attributs.

Classes Vs Objets

```
public class Personne {
    // Attributs
    String nom;
    int age;

    // Méthodes
    void direBonjour() {
        System.out.println("Bonjour, je m'appelle " + nom
+ " et j'ai " + age + " ans.");
    }
}
```

```
Personne personne1 = new Personne();
personne1.nom = "Alice";
personne1.age = 30;
personne1.direBonjour();  // Affiche : Bonjour, je
m'appelle Alice et j'ai 30 ans.

Personne personne2 = new Personne();
personne2.nom = "Bob";
personne2.age = 25;
personne2.direBonjour();  // Affiche : Bonjour, je
m'appelle Bob et j'ai 25 ans.
```

Personne1 et **Personne2** sont des objets différents, chacun ayant des valeurs différentes pour nom et age.

Programmation orientée objet avec JAVA

2- Constructeurs




Constructeurs

- Définition : méthode spéciale pour initialiser les objets
- Exemple :

```
public class Person {
    String name;
    int age;

    // Constructeur
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Constructeurs

1. Définition du Constructeur

- Un **constructeur** est une méthode spéciale utilisée pour initialiser les objets lors de leur création.
- Il a le même nom que la classe et ne retourne aucune valeur (pas même void).
- Son rôle principal est de **donner des valeurs initiales** aux attributs de la classe.

2. Caractéristiques d'un Constructeur

- **Pas de valeur de retour :** Contrairement aux méthodes ordinaires, un constructeur ne spécifie pas de type de retour.
- **Même nom que la classe :** Le nom du constructeur doit correspondre exactement au nom de la classe.
- **Appel automatique lors de la création de l'objet :** Le constructeur est automatiquement invoqué avec l'instruction new.

3. Types de Constructeurs

- **Constructeur par défaut (Default Constructor) :**
  - Si aucun constructeur n'est défini par l'utilisateur, le compilateur génère un constructeur par défaut.
  - Il ne prend aucun paramètre et initialise les attributs à leurs valeurs par défaut (0, null, etc.).
- **Constructeur avec paramètres (Parameterized Constructor) :**
  - Permet d'initialiser les attributs avec des valeurs spécifiques lors de la création de l'objet.

Constructeurs

```
public class Personne {
    // Attributs
    String nom;
    int age;

    // Méthodes
    void direBonjour() {
        System.out.println("Bonjour, je m'appelle " + nom + " et j'ai " + age + " ans.");
    }
}
```

Exemple Pratique : Supposons que nous avons une classe Voiture avec un constructeur qui initialise la marque et l'année du modèle

Ecrivez cette classe contenant
Un constructeur
Une méthode pour afficher la marque et l'année du modèle
La méthode main pour lancer notre programme

```
Personne personne1 = new Personne();
personne1.nom = "Alice";
personne1.age = 30;
personne1.direBonjour();  // Affiche : Bonjour, je m'appelle Alice et j'ai 30 ans.

Personne personne2 = new Personne();
personne2.nom = "Bob";
personne2.age = 25;
personne2.direBonjour();  // Affiche : Bonjour, je m'appelle Bob et j'ai 25 ans.
```

```
public class Voiture {
    String marque;
    int annee;

    // Constructeur
    public Voiture(String marque, int annee) {
        this.marque = marque;
        this.annee = annee;
    }

    public void afficherDetails() {
        System.out.println("Marque: " + marque + ", Année: " +
annee);
    }

    public static void main(String[] args) {
        // Création d'un objet Voiture avec le constructeur
        Voiture voiture1 = new Voiture("Toyota", 2020);
        voiture1.afficherDetails();  // Marque: Toyota, Année: 2020
    }
}
```

Constructeurs

- **Explication du code :**
  - Le constructeur Voiture(String marque, int annee) est utilisé pour initialiser les attributs marque et annee.
  - Lors de la création de l'objet voiture1, les valeurs "Toyota" et 2020 sont passées au constructeur pour initialiser les attributs.
  - L'objet voiture1 est maintenant prêt à être utilisé avec ces valeurs.

5. Surcharge des Constructeurs (Constructor Overloading)

- Java permet la **surcharge des constructeurs**, c'est-à-dire la création de plusieurs constructeurs avec des signatures différentes (nombre ou type de paramètres différents

Constructeurs

```
public class Voiture {
    String marque;
    int annee;

    // Constructeur sans paramètres
    public Voiture() {
        this.marque = "Inconnue";
        this.annee = 0;
    }

    // Constructeur avec paramètres
    public Voiture(String marque, int annee) {
        this.marque = marque;
        this.annee = annee;
    }
}
```

Constructeurs

- **Explication :**
  - Le constructeur sans paramètres initialise les valeurs par défaut.
  - Le constructeur avec paramètres permet de spécifier les valeurs lors de la création de l'objet.

Appel de Constructeur avec this()

- On peut utiliser l'instruction this() pour appeler un autre constructeur de la même classe.

```
public class Voiture {
    String marque;
    int annee;

    // Constructeur principal
    public Voiture() {
        this("Inconnue", 0);  // Appel d'un autre constructeur
    }

    public Voiture(String marque, int annee) {
        this.marque = marque;
        this.annee = annee;
    }
}
```

Constructeurs

Exemple Pratique : Supposons que nous avons une classe Etudiant avec un constructeur qui initialise le nom, l'âge et le niveau scolaire

Ecrivez cette classe contenant
Un constructeur qui définit le nom, l'âge et le niveau scolaire
Un constructeur sans paramètres qui affecte les valeurs par défaut en utilisant this()
Une méthode pour afficher les informations sur l'étudiant
La méthode main pour lancer notre programme

Programmation orientée objet avec JAVA

3- Héritage




Héritage

Définition : mécanisme permettant de créer une nouvelle classe à partir d'une classe existante
Syntaxe : En Java, l'héritage est exprimé avec le mot-clé extends :

```
public class Employee extends Person {
   double salary;
}
```

- L'héritage permet à une classe (la **classe fille** ou **sous-classe**) de **hériter** des propriétés (attributs) et des comportements (méthodes) d'une autre classe (la **classe parent** ou **super-classe**).
- Cela favorise la réutilisation du code et permet d'organiser les classes en hiérarchies logiques.

```
class ClasseParent {
   // Propriétés et méthodes de la classe parent
}

class ClasseFille extends ClasseParent {
   // Propriétés et méthodes spécifiques à la classe fille
}
```

Héritage

Prenons l'exemple d'une classe Animal et de ses sous-classes Chien et Chat :

```
// Super-classe
public class Animal {
    String nom;

    public void manger() {
        System.out.println(nom + " est en train de manger.");
    }
}
// Sous-classe
public class Chien extends Animal {
    public void aboyer() {
        System.out.println(nom + " aboie.");
    }
}
// Sous-classe
public class Chat extends Animal {
    public void miauler() {
        System.out.println(nom + " miaule.");
    }
}
```

Héritage

Explication :

- La classe Animal contient un attribut nom et une méthode manger().
- La classe Chien hérite de Animal et possède une méthode spécifique aboyer().
- La classe Chat hérite également de Animal et a sa propre méthode miauler().

Héritage et Réutilisation du Code

- En utilisant l'héritage, les méthodes communes à plusieurs classes peuvent être définies dans une super-classe. Cela évite de dupliquer le code.
- Par exemple, la méthode manger() est définie dans la classe Animal, et toutes les sous-classes Chien et Chat l'héritent automatiquement.

Appel du Constructeur Parent avec super()

- Lorsqu'une sous-classe hérite d'une classe parent, elle peut appeler le **constructeur** de la classe parent à l'aide du mot-clé super() pour initialiser les propriétés héritées.
- Exemple :

Héritage

Constructeurs et Héritage

•Les constructeurs ne sont **pas hérités** par les sous-classes, mais une sous-classe peut appeler le constructeur de la classe parente via super().

•Exemple :

```
public class Voiture {
    String marque;
    public Voiture(String marque) {
        this.marque = marque;
    }
}
public class VoitureElectrique extends Voiture {
    int autonomie;
    public VoitureElectrique(String marque, int autonomie) {
        super(marque);  // Appel du constructeur parent
        this.autonomie = autonomie;
    }
}
```

Héritage

```
public class Animal {
    String nom;
    public Animal(String nom) {
        this.nom = nom;
    }
    public void manger() {
        System.out.println(nom + " est en train de manger.");
    }
}
public class Chien extends Animal {
    public Chien(String nom) {
        super(nom);  // Appel du constructeur de la classe parent
    }
    public void aboyer() {
        System.out.println(nom + " aboie.");
    }
}
```

Héritage

Exemple Pratique : Supposons que nous avons une classe Personne, Etudiant et Professeur

La classe Personne possède l'attribut nom et prénom. Et le constructeur qui initialise le nom, la méthode afficherDétails().
La classe Etudiant possède l'attribut : niveauScolaire. Et la méthode afficherNiveauScolaire() qui afficher aussi le nom et prénom de l'étudiant.
La classe Professeur possède l'attribut : niveauEnseigné. Et la méthode afficherNiveauEnseigné() qui afficher aussi le nom et prénom du professeur.
Ecrivez la classe personne et ses sous-classes ainsi qu'une classe main pour lancer le programme qui affiche les différentes informations