LA PROGRAMMATION ORIENTÉE OBJET

Résumé complet avec exemples Java — style David J. Malan ■

■ ■ Ce document est conçu pour être lu lentement, un concept à la fois. Chaque idée est expliquée avec une analogie du monde réel, puis traduite en code Java commenté pas à pas.

---

1. Pourquoi la Programmation Orientée Objet ?

Avant la POO, les programmes ressemblaient à une grande liste d'instructions. Les données et les actions étaient séparées, sans lien clair entre elles. Résultat : du code difficile à lire, à modifier, et à réutiliser.

La POO change ça avec une idée simple :

■ Le monde réel est fait d'objets → le code aussi

Une voiture a des propriétés (couleur, vitesse) et des actions (démarrer, rouler). En POO, on crée un objet **Voiture** dans le code qui fait exactement pareil.

Les 3 piliers de la POO

- ■ **Encapsulation** → regrouper données + actions dans une boîte, et cacher ce qui est interne
- ■ **Héritage** → une classe peut hériter d'une autre pour réutiliser son code
- ■ **Polymorphisme** → le même message peut déclencher des comportements différents selon l'objet

2. L'Objet — La brique de base

Un objet est la représentation informatique d'une chose réelle ou abstraite. Cette chose possède des caractéristiques (son **état**) et sait faire des choses (son **comportement**).

OBJET = identité + état + comportement

Décomposition avec l'exemple de la voiture

- ■ **Identité** : ce qui distingue cet objet de tous les autres — géré automatiquement par Java
- ■ **État** : les attributs (variables) qui décrivent l'objet à un instant T
- ■■ **Comportement** : les méthodes (fonctions) que l'objet peut exécuter

■ **Représentation d'un objet en mémoire**

```
// Un objet 'voiture' en mémoire — ce que Java stocke
uneVoiture {
    // --- ÉTAT (attributs) ---
    couleur       = "bleue"
    vitesse       = 50        // km/h
    niveauEssence = 30        // litres
    conducteur    = "Dupont"

    // --- COMPORTEMENT (méthodes) ---
    démarrer()       // démarre le moteur
    rouler()         // fait avancer la voiture
    faireLePlein()   // remplit le réservoir
}
```

■ **Retiens ça : un objet = données + fonctions dans la même boîte. C'est tout.**

3. La Classe — Le moule à objets

Une classe est le **plan de construction**. Elle définit ce que tous les objets de ce type auront en commun. L'objet, lui, est ce qu'on fabrique avec ce plan.

Analogie de l'usine

■ ■ La classe Voiture = les plans de l'usine Renault (écrits une seule fois). Chaque voiture produite = un objet (une instance). Toutes les voitures ont les mêmes attributs définis dans le plan, mais chacune a ses propres valeurs.

Écriture d'une classe en Java

■ **Voiture.java — Définition de la classe**

```
// █████████████████████████████████████████████████████
// CLASSE = le plan. On l'écrit une seule fois.
// █████████████████████████████████████████████████████
public class Voiture {

    // ██ ATTRIBUTS (l'état) ██████████████████████████
    private String couleur;       // chaque voiture a sa couleur
    private double vitesse;       // vitesse actuelle en km/h
    private double niveauEssence; // litres restants

    // ██ CONSTRUCTEUR (comment créer un objet) ███████
    public Voiture(String couleur, double niveauEssence) {
        this.couleur        = couleur;
        this.vitesse        = 0;          // démarre à l'arrêt
        this.niveauEssence  = niveauEssence;
    }

    // ██ MÉTHODES (le comportement) ███████████████████
    public void demarrer() {
        this.vitesse = 10; // on démarre doucement
        System.out.println("La voiture démarre !");
    }

    public void rouler(double km) {
        this.niveauEssence -= km * 0.07; // 7L aux 100km
        System.out.println("Parcouru : " + km + " km");
    }
}
```

Créer des objets depuis la classe (instanciation)

■ **Main.java — Création et utilisation d'objets**

```
// █████████████████████████████████████████████████████
// INSTANCIATION = créer un objet depuis le plan (la classe)
// █████████████████████████████████████████████████████
public class Main {
    public static void main(String[] args) {

        // 'new' = fabriquer un objet en mémoire
        //  ↓ type  ↓ nom de l'objet       ↓ valeurs initiales
```

```
        Voiture maVoiture   = new Voiture("bleue",  50.0);
        Voiture tonneVoiture = new Voiture("rouge", 30.0);

        // Ces deux objets sont INDÉPENDANTS
        // Modifier l'un ne change pas l'autre
        maVoiture.demarrer();     // → "La voiture démarre !"
        maVoiture.rouler(100);    // → "Parcouru : 100 km"
    }
}
```

■■ **Classe = le plan. Objet = ce qu'on construit. new = le mot magique pour construire.**

4. L'Encapsulation — La boîte noire

L'encapsulation consiste à **cacher les détails internes** d'un objet et à n'exposer que ce qui est nécessaire. C'est exactement comme un téléphone : tu appuies sur un bouton, tu n'as pas besoin de savoir comment le circuit fonctionne.

---

private vs public — les modificateurs d'accès

- ■ **private** → accessible uniquement DANS la classe. Personne d'autre ne peut y toucher directement.
- ■ **public** → accessible de partout. C'est l'interface visible de l'objet.

■ **CompteBancaire.java — Encapsulation avec getters/setters**

```
public class CompteBancaire {

    // PRIVÉ : personne ne peut modifier ça directement
    private double solde;
    private String titulaire;

    public CompteBancaire(String titulaire, double soldeInitial) {
        this.titulaire = titulaire;
        this.solde     = soldeInitial;
    }

    // PUBLIC : la seule façon d'accéder au solde
    // C'est un 'getter' — il retourne une valeur
    public double getSolde() {
```

```
        return this.solde;
    }

    // PUBLIC : la seule façon de déposer de l'argent
    // C'est un 'setter avec logique' — il VÉRIFIE avant de modifier
    public void deposer(double montant) {
        if (montant <= 0) {
            System.out.println("Montant invalide !");
            return; // on arrête tout
        }
        this.solde += montant; // on modifie l'état interne
        System.out.println("Dépôt OK. Nouveau solde : " + this.solde);
    }

    public void retirer(double montant) {
        if (montant > this.solde) {
            System.out.println("Fonds insuffisants !");
            return;
        }
        this.solde -= montant;
    }
}
```

■ Main.java — Démonstration de l'encapsulation

```
// CE QU'ON PEUT FAIRE (via l'interface publique)
CompteBancaire compte = new CompteBancaire("Alice", 1000.0);
compte.deposer(500);         // ■ OK — méthode publique
double s = compte.getSolde(); // ■ OK — getter public

// CE QU'ON NE PEUT PAS FAIRE (privé = protégé)
// compte.solde = 999999;    // ■ ERREUR DE COMPILATION
// compte.solde += 100;      // ■ ERREUR DE COMPILATION

// Pourquoi c'est bien ?
// → impossible de mettre un solde négatif par accident
// → la logique de vérification est TOUJOURS appliquée
```

■ L'encapsulation protège les données. On passe toujours par les méthodes, jamais directement.

5. L'Héritage — La famille de classes

L'héritage permet à une classe (sous-classe) de **récupérer automatiquement** tout ce qu'une autre classe (super-classe) possède, et d'y ajouter ses propres spécificités.

■ C'est la génétique du code. Un chien hérite des caractéristiques d'un mammifère. Un mammifère hérite des caractéristiques d'un animal. Et le chien peut en plus aboyer — ce que les autres animaux ne font pas.

La règle d'or pour l'héritage

'Un X EST UN Y' → X peut hériter de Y

■ Un CompteEpargne EST UN Compte → ■ héritage logique

■ Un Chien EST UN Animal → ■ héritage logique

■ Une Voiture EST UN Moteur → ■ non (une voiture A un moteur, pas IS un moteur)

Exemple complet : comptes bancaires

■ **Compte.java — Super-classe (parent)**

```
// ■■■ SUPER-CLASSE (classe parent) ■■■■■■■■■■■■■■■■■■■■■
public class Compte {
    // Ces attributs seront hérités par TOUTES les sous-classes
    protected double solde;     // 'protected' = visible par les enfants
    protected String titulaire;

    public Compte(String titulaire, double solde) {
        this.titulaire = titulaire;
        this.solde     = solde;
    }

    public void deposer(double montant) {
        this.solde += montant;
    }

    public void retirer(double montant) {
        if (montant <= this.solde) {
```

```
            this.solde -= montant;
        }
    }

    public double getSolde() { return this.solde; }
}
```

■ **CompteEpargne.java — Sous-classe (enfant)**

```
// ■■■ SOUS-CLASSE (classe enfant) ■■■■■■■■■■■■■■■■■■■■■■
//  'extends' = "hérite de"
public class CompteEpargne extends Compte {

    // Attribut AJOUTÉ (spécifique à CompteEpargne)
    private double taux; // taux d'intérêt annuel

    // Le constructeur appelle d'abord celui du parent avec 'super()'
    public CompteEpargne(String titulaire, double solde, double taux) {
        super(titulaire, solde); // ← appel du constructeur de Compte
        this.taux = taux;
    }

    // Méthode AJOUTÉE (spécifique à CompteEpargne)
    public void calculerInterets() {
        // 'this.solde' vient de Compte (hérité via 'protected')
        double interets = this.solde * this.taux / 100;
        this.solde += interets;
        System.out.println("Intérêts ajoutés : " + interets + " €");
    }
}
```

■ **Main.java — Utilisation de l'héritage**

```
public class Main {
    public static void main(String[] args) {

        CompteEpargne livretA = new CompteEpargne("Bob", 2000.0, 3.0);

        // Méthodes HÉRITÉES de Compte — disponibles sans rien faire
        livretA.deposer(500);        // héritée ✓
        livretA.retirer(200);        // héritée ✓
        System.out.println(livretA.getSolde()); // héritée ✓ → 2300.0
```

```
        // Méthode PROPRE à CompteEpargne
        livretA.calculerInterets();  // → "Intérêts ajoutés : 69.0 €"
    }
}
```

■■ L'héritage évite la duplication. On écrit le code commun une fois dans le parent, les enfants en profitent automatiquement.

6. Le Polymorphisme — Même message, résultats différents

Poly = plusieurs. Morphe = forme. Le polymorphisme c'est la capacité d'envoyer le **même message** à des objets différents et d'obtenir des comportements adaptés à chacun.

Analogie musicale ■

■ Tu dis 'joue une note' à un pianiste, un guitariste, et un trompettiste. Le message est identique. Chacun réagit à sa façon. C'est exactement ça le polymorphisme.

Exemple : les formes géométriques

■ Figure.java — Classe abstraite

```
// ■■■ CLASSE ABSTRAITE (template) ■■■■■■■■■■■■■■■■■■■■■■■
// 'abstract' = on ne peut pas créer un objet Figure directement
// Elle sert juste de base commune
public abstract class Figure {

    // Attribut commun à toutes les figures
    protected String couleur;

    public Figure(String couleur) {
        this.couleur = couleur;
    }

    // Méthode abstraite = CHAQUE sous-classe DOIT la définir
    // On ne peut pas écrire le code ici car on ne sait pas
    // encore quelle forme c'est !
```

```
    public abstract double calculerSurface();

    // Méthode concrète (commune à toutes les figures)
    public void afficher() {
        System.out.println("Figure " + couleur + ", surface = "
                           + calculerSurface());
    }
}
```

■ **Cercle.java & Rectangle.java — Sous-classes avec @Override**

```
// ■■■ SOUS-CLASSES CONCRÈTES ■■■■■■■■■■■■■■■■■■■■■■■■■■■

public class Cercle extends Figure {
    private double rayon;

    public Cercle(String couleur, double rayon) {
        super(couleur);  // appel du constructeur de Figure
        this.rayon = rayon;
    }

    // REDÉFINITION (override) de calculerSurface()
    @Override
    public double calculerSurface() {
        return Math.PI * rayon * rayon; // π × r²
    }
}

public class Rectangle extends Figure {
    private double largeur, hauteur;

    public Rectangle(String couleur, double l, double h) {
        super(couleur);
        this.largeur = l;
        this.hauteur = h;
    }

    @Override
    public double calculerSurface() {
        return largeur * hauteur; // l × h
    }
}
```

■ Main.java — Polymorphisme en action

```
public class Main {
    public static void main(String[] args) {

        // On crée un tableau de Figure — peut contenir N'IMPORTE quelle sous-classe
        Figure[] formes = {
            new Cercle("rouge", 5.0),
            new Rectangle("bleu", 4.0, 6.0),
            new Cercle("vert", 3.0),
        };

        // ■ MAGIE DU POLYMORPHISME ■
        // Une seule boucle, mais chaque objet répond à SA façon
        for (Figure f : formes) {
            f.afficher(); // Java appelle la BONNE version selon l'objet réel
        }

        // Résultat :
        // → Figure rouge, surface = 78.54
        // → Figure bleu, surface = 24.0
        // → Figure vert, surface = 28.27
    }
}
```

■ @Override = on redéfinit une méthode héritée. Java appellera toujours la version la plus spécifique.

■ Super pouvoir : si tu ajoutes Triangle demain, la boucle fonctionne sans modification. Zéro changement ailleurs !

7. Récapitulatif — Tout en un coup d'œil

Concept	En une phrase	Mot-clé Java	Analogie
■ **Classe**	Le plan de construction	class	Les plans d'architecte
■ **Objet**	Une chose concrète créée depuis une classe	new	La maison construite
■ **Instance**	Un objet spécifique d'une classe	new	Cette maison-ci, pas une autre
■ **Encapsulation**	Cacher les détails, exposer une interface claire	private / public	Boîte noire d'un avion
■ **Héritage**	Une classe reprend tout d'une autre + ses spécificités	extends	Enfant hérite des parents
■ **Polymorphisme**	Même message = comportements différents selon l'objet	@Override	Chaque musicien joue à sa façon
■ **Abstraction**	Classe qu'on ne peut pas instancier, juste hériter	abstract	La recette générale, pas le plat

8. Checklist — Comment reconnaître les concepts

Tu vois	Concept	Exemple
`class NomClasse { ... }`	**Définition d'une classe**	`class Voiture { ... }`
`new NomClasse(...)`	**Création d'un objet (instanciation)**	`new Voiture("bleue", 50)`
`private attribut`	**Encapsulation (donnée cachée)**	`private double solde`
`public méthode()`	**Interface publique de l'objet**	`public void deposer()`
`extends ParentClasse`	**Héritage**	`class CompteEpargne extends Compte`
`super(...)`	**Appel du constructeur parent**	`super(titulaire, solde)`

Tu vois	Concept	Exemple
`@Override`	**Redéfinition polymorphe**	`@Override public double surface()`
`abstract class / méthode`	**Classe ou méthode abstraite**	`abstract double calculerSurface()`

Pour aller plus loin

■ La POO, c'est organiser ton code comme le monde réel : des objets qui ont des propriétés et savent faire des choses. Ce document couvre les bases — la suite c'est les interfaces, les collections, et les design patterns. ■

Les prochaines étapes naturelles en Java :

- Les **interfaces** (interface) → définir un contrat sans implémentation
- Les **collections** (ArrayList, HashMap) → gérer des groupes d'objets
- La gestion d'**exceptions** (try/catch) → gérer les erreurs proprement
- Les **design patterns** → recettes éprouvées pour des problèmes classiques

---