Développement en langage objet

Guide complet sur la programmation orientée objet : concepts fondamentaux, principes avancés (encapsulation, héritage, polymorphisme, abstraction), design patterns, principes SOLID, anti-patterns et environnement de développement. Exemples multi-langages (Java, Python, C#, PHP).

Développement Web Programmation 40 min de lecture Niveau Intermédiaire

Document généré le 11/07/2026 à 21h27 · nouvy.fr/wiki/developpement-langage-objet

Sommaire

17 section(s) · 40 min de lecture

Utiliser la programmation orientée objet avec le langage sélectionné — Développement avancé — Mise en pratique

Ce guide couvre l'ensemble des concepts de la **programmation orientée objet (POO)**, des fondamentaux aux techniques avancées. Chaque notion est illustrée dans **plusieurs langages** (Java, Python, C#, PHP) pour montrer l'universalité des concepts.

Introduction à la programmation orientée objet (POO)

Concepts fondamentaux

Définition et historique de la POO

La **programmation orientée objet** est un paradigme qui organise le code autour d'**objets** — des entités regroupant des **données** (attributs) et des **comportements** (méthodes) — plutôt que de séquences d'instructions.

Repères historiques :

- **1967 — Simula** : premier langage à introduire les concepts de classes et d'objets (Ole-Johan Dahl et Kristen Nygaard).
- **1972 — Smalltalk** : popularise la POO avec un modèle purement objet (Alan Kay, Xerox PARC).
- **1983 — C++** : ajoute la POO au langage C (Bjarne Stroustrup).
- **1995 — Java** : langage objet portable et sécurisé (Sun Microsystems).
- **1995 — PHP 3/4** puis **PHP 5** (2004) : introduction progressive de la POO dans le web.
- **2000 — C#** : langage objet de Microsoft pour la plateforme .NET.
- **2000+ — Python, TypeScript, Kotlin, Swift…** : la POO est intégrée dans quasiment tous les langages modernes.

Pourquoi utiliser la POO ? Comparaison avec la programmation procédurale

Critère	Procédurale	Orientée objet
Organisation	Fonctions et variables globales	Classes et objets
Réutilisabilité	Copier-coller de fonctions	Héritage et composition
Maintenabilité	Difficile sur gros projets	Modulaire et extensible
Encapsulation	Pas native	Intégrée (public, private, protected)
Modélisation	Centrée sur les traitements	Centrée sur les entités du domaine

Avantages concrets de la POO :

- **Modularité** : chaque classe a une responsabilité claire.
- **Réutilisabilité** : l'héritage et la composition permettent de partager du code sans le dupliquer.

- **Maintenabilité** : les modifications sont localisées dans des classes spécifiques.
- **Extensibilité** : ajout de fonctionnalités sans modifier le code existant (principe Open/Closed).
- **Collaboration** : les équipes peuvent travailler sur des classes différentes en parallèle.

Principes de base

Les quatre piliers de la POO

La POO repose sur quatre grands principes. Pensez à eux comme les **règles fondamentales** qui guident la façon dont on organise le code :

Pilier	En une phrase	Analogie du monde réel
Encapsulation	Cacher les données internes et n'exposer que des méthodes contrôlées	Comme une voiture : on utilise le volant et les pédales (interface publique), sans toucher au moteur directement (données internes)
Héritage	Réutiliser et spécialiser le comportement d'une classe existante	Comme une recette de famille : on reprend la base et on y ajoute sa touche personnelle
Polymorphisme	Un même appel de méthode, des comportements différents selon le type réel	Comme le bouton « play » : il lance une vidéo, une musique ou un podcast selon le contexte
Abstraction	Ne montrer que l'essentiel, masquer la complexité d'implémentation	Comme une télécommande : on appuie sur un bouton sans connaître l'électronique interne

Ces quatre piliers ne sont pas indépendants : ils se **complètent**. Par exemple, l'encapsulation permet de cacher les détails que l'abstraction ne veut pas montrer, et le polymorphisme s'appuie sur l'héritage pour fonctionner.

Les notions de classes et d'objets

- **Classe** : un **modèle** (ou plan) qui décrit les attributs et les méthodes d'un type d'objet. C'est comme un **plan d'architecte** : il décrit comment construire une maison, mais ce n'est pas encore une maison.
- **Objet** (ou **instance**) : une réalisation concrète d'une classe, avec ses propres valeurs d'attributs. C'est la **maison construite** à partir du plan. On peut construire plusieurs maisons (objets) à partir du même plan (classe), chacune avec ses propres caractéristiques (couleur, nombre d'étages...).

> ***Exemple concret** : la classe `Voiture` décrit qu'une voiture a une marque et une vitesse. L'objet `maVoiture` est une Peugeot qui roule à 50 km/h — c'est **une voiture précise** créée à partir du modèle.*

Comparaison dans plusieurs langages :

Java :

```
public class Voiture {
    private String marque;
    private int vitesse;

    public Voiture(String marque) {
        this.marque = marque;
        this.vitesse = 0;
    }

    public void accelerer(int kmh) {
        this.vitesse += kmh;
    }
}

// Création d'un objet
Voiture maVoiture = new Voiture("Peugeot");
maVoiture.accelerer(50);
```

Copier

Python :

```
class Voiture:
    def __init__(self, marque):
        self.__marque = marque   # attribut privé (convention __)
        self.__vitesse = 0

    def accelerer(self, kmh):
        self.__vitesse += kmh

# Création d'un objet
ma_voiture = Voiture("Peugeot")
ma_voiture.accelerer(50)
```

Copier

C# :

```
public class Voiture
{
    private string marque;
    private int vitesse;

    public Voiture(string marque)
    {
        this.marque = marque;
        this.vitesse = 0;
    }

    public void Accelerer(int kmh)
    {
        this.vitesse += kmh;
    }
}

// Création d'un objet
Voiture maVoiture = new Voiture("Peugeot");
maVoiture.Accelerer(50);
```

Copier

PHP :

```
class Voiture {
    private string $marque;
    private int $vitesse;

    public function __construct(string $marque) {
        $this->marque = $marque;
        $this->vitesse = 0;
    }

    public function accelerer(int $kmh): void {
        $this->vitesse += $kmh;
    }
}

// Création d'un objet
$maVoiture = new Voiture("Peugeot");
$maVoiture->accelerer(50);
```

Copier

Syntaxe et structure en langage objet

Structure d'une classe : propriétés (attributs) et méthodes

Une classe se compose de :

- **Attributs (propriétés)** : les données de l'objet.
- **Méthodes** : les comportements / actions de l'objet.

```
┌──────────────────────────────┐
│        Produit               │  ← Nom de la classe
├──────────────────────────────┤
│ - nom : String               │
│ - prix : Double              │  ← Attributs (propriétés)
│ - quantite : Int             │
├──────────────────────────────┤
│ + getNom() : String          │
│ + setPrix(p: Double)         │  ← Méthodes
│ + getValeurStock(): Double   │
└──────────────────────────────┘
```

Copier

Java :

```
public class Produit {
    private String nom;
    private double prix;
    private int quantite;

    public Produit(String nom, double prix, int quantite) {
        this.nom = nom;
        this.prix = prix;
        this.quantite = quantite;
    }

    public String getNom() { return nom; }
    public void setPrix(double prix) {
        if (prix >= 0) this.prix = prix;
    }

    public double getValeurStock() {
        return prix * quantite;
    }
}
```

Copier

Python :

```
class Produit:
    def __init__(self, nom: str, prix: float, quantite: int):
        self.__nom = nom
        self.__prix = prix
        self.__quantite = quantite

    @property
    def nom(self) -> str:
        return self.__nom

    @property
    def prix(self) -> float:
        return self.__prix

    @prix.setter
    def prix(self, valeur: float):
        if valeur >= 0:
            self.__prix = valeur

    def get_valeur_stock(self) -> float:
        return self.__prix * self.__quantite
```

Copier

C# :

```
public class Produit
{
    public string Nom { get; private set; }
    private double _prix;
    public double Prix
    {
        get => _prix;
        set { if (value >= 0) _prix = value; }
    }
    public int Quantite { get; set; }

    public Produit(string nom, double prix, int quantite)
    {
        Nom = nom;
        Prix = prix;
        Quantite = quantite;
    }

    public double GetValeurStock() => Prix * Quantite;
}
```

Copier

PHP :

```
class Produit {
    private string $nom;
    private float $prix;
    private int $quantite;

    public function __construct(string $nom, float $prix, int $quantite) {
        $this->nom = $nom;
        $this->prix = $prix;
        $this->quantite = $quantite;
    }

    public function getNom(): string { return $this->nom; }

    public function setPrix(float $prix): void {
        if ($prix >= 0) $this->prix = $prix;
    }

    public function getValeurStock(): float {
        return $this->prix * $this->quantite;
    }
}
```

Copier

Constructeurs et destructeurs

Le **constructeur** est une méthode spéciale appelée **automatiquement** au moment où l'on crée un objet avec `new`. Son rôle est d'**initialiser** l'objet : donner des valeurs de départ aux attributs, ouvrir une connexion, préparer des ressources, etc. Sans constructeur, on devrait appeler manuellement plusieurs setters après chaque création — ce qui serait répétitif et source d'erreurs.

Le **destructeur** est l'inverse : il est appelé automatiquement quand l'objet est **détruit** (quand il sort de la portée, ou que le programme se termine). Il sert à **libérer les ressources** (fermer un fichier, une connexion réseau ou une base de données). En pratique,

tous les langages n'en ont pas besoin : Java et Python utilisent un **ramasse-miettes** (*garbage collector*) qui gère la mémoire automatiquement.

Langage	Constructeur	Destructeur
Java	`public NomClasse() {}`	`finalize()` (déprécié, le GC gère)
Python	`def __init__(self):`	`def __del__(self):`
C#	`public NomClasse() {}`	`~NomClasse() {}` ou `IDisposable`
PHP	`public function __construct() {}`	`public function __destruct() {}`
C++	`NomClasse() {}`	`~NomClasse() {}`

Exemple en Python (constructeur + destructeur) :

```
class ConnexionBDD:
    def __init__(self, hote: str, port: int):
        self.hote = hote
        self.port = port
        self.connexion = self._connecter()
        print(f"Connecté à {hote}:{port}")

    def _connecter(self):
        # Logique de connexion...
        return True

    def __del__(self):
        # Nettoyage : fermer la connexion
        print(f"Déconnexion de {self.hote}:{self.port}")
```

Copier

Exemple en PHP :

```
class ConnexionBDD {
    private string $hote;
    private PDO $connexion;

    public function __construct(string $hote, string $base, string $user, string $pass) {
        $this->hote = $hote;
        $this->connexion = new PDO("mysql:host=$hote;dbname=$base", $user, $pass);
        echo "Connecté à $hote\n";
    }

    public function __destruct() {
        $this->connexion = null;
        echo "Déconnexion de {$this->hote}\n";
    }
}
```

Copier

Modificateurs d'accès : public, privé, protégé

Les modificateurs d'accès (ou **visibilité**) déterminent **qui a le droit** d'accéder à un attribut ou une méthode. C'est le mécanisme concret de l'encapsulation : on choisit ce qui est visible de l'extérieur et ce qui reste interne.

Imaginez une entreprise : certaines informations sont **publiques** (le nom de l'entreprise), d'autres sont réservées aux **employés et filiales** (les procédures internes = `protected`), et d'autres sont strictement **confidentielles** (les mots de passe = `private`).

Modificateur	Description	Accessible depuis
public	Accessible partout	Classe, sous-classes, extérieur
private	Accessible uniquement dans la classe	Classe uniquement
protected	Accessible dans la classe et ses sous-classes	Classe et sous-classes

Syntaxe par langage :

Langage	public	private	protected
Java	`public`	`private`	`protected`
Python	`nom` (convention)	`__nom` (name mangling)	`_nom` (convention)
C#	`public`	`private`	`protected`
PHP	`public`	`private`	`protected`

> ***Note Python** : il n'y a pas de vrai modificateur d'accès. Le préfixe `_` est une convention pour indiquer « protégé » et `__` déclenche le name mangling pour simuler le « privé ».*

Types de méthodes : statiques vs instance

- **Méthode d'instance** : opère sur un **objet spécifique**, accède à `this` / `self`. Elle a besoin qu'un objet existe pour fonctionner, car elle travaille avec les données de cet objet précis.
- **Méthode statique** : appartient à la **classe elle-même**, pas à un objet. On peut l'appeler sans créer d'instance. Elle est utile pour des opérations utilitaires qui ne dépendent pas de l'état d'un objet (ex. : calculer un maximum, convertir une unité, générer un identifiant).

> ***Analogie** : une méthode d'instance, c'est comme demander à **une personne précise** son nom (`alice.getNom()`). Une méthode statique, c'est comme consulter **l'annuaire de la classe** (`Personne.compter()`) — on n'a pas besoin d'une personne en particulier.*

Java :

```
public class MathUtils {
    private static int compteur = 0; // Attribut statique partagé

    public static int max(int a, int b) { // Méthode statique
        return (a > b) ? a : b;
    }

    public MathUtils() { compteur++; } // Constructeur d'instance

    public int getCompteur() { return compteur; } // Méthode d'instance
}

// Appel statique : pas besoin d'objet
int resultat = MathUtils.max(10, 20);
```

Copier

Python :

```
class MathUtils:
    compteur = 0  # Attribut de classe

    @staticmethod
    def maximum(a, b):
        return a if a > b else b

    @classmethod
    def incrementer(cls):
        cls.compteur += 1

# Appel statique
resultat = MathUtils.maximum(10, 20)
```

Copier

PHP :

```
class MathUtils {
    private static int $compteur = 0;

    public static function max(int $a, int $b): int {
        return ($a > $b) ? $a : $b;
    }

    public function __construct() { self::$compteur++; }
}

// Appel statique
$resultat = MathUtils::max(10, 20);
```

Copier

Méthodes abstraites et interfaces

- **Méthode abstraite** : une méthode **déclarée mais pas implémentée** (elle n'a pas de corps / pas de code). Elle existe dans une **classe abstraite** et dit : « toute classe qui hérite de moi **doit** fournir une implémentation de cette méthode ». C'est une façon de **forcer** les classes filles à définir un comportement spécifique.
- **Interface** : un **contrat** (ou « cahier des charges ») qui liste des méthodes qu'une classe **s'engage** à implémenter. Contrairement à une classe abstraite, une

interface ne contient **aucun état** (pas d'attributs) et une classe peut implémenter **plusieurs interfaces** (là où l'héritage de classe est souvent limité à un seul parent).

> ***Analogie** : une classe abstraite, c'est comme un **formulaire pré-rempli** avec certains champs déjà complétés (méthodes concrètes) et d'autres à remplir obligatoirement (méthodes abstraites). Une interface, c'est comme un **contrat de travail** : il liste les compétences requises, mais chaque employé les met en pratique à sa façon.*

Java :

```
// Classe abstraite
public abstract class Forme {
    public abstract double aire();
    public void afficher() { System.out.println("Aire : " + aire()); }
}

// Interface
public interface Dessinable {
    void dessiner();
}

public class Cercle extends Forme implements Dessinable {
    private double rayon;
    public Cercle(double rayon) { this.rayon = rayon; }

    @Override
    public double aire() { return Math.PI * rayon * rayon; }

    @Override
    public void dessiner() { System.out.println("Dessin du cercle"); }
}
```

Copier

Python :

```
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def aire(self) -> float:
        pass

    def afficher(self):
        print(f"Aire : {self.aire()}")

class Dessinable(ABC):
    @abstractmethod
    def dessiner(self):
        pass

class Cercle(Forme, Dessinable):
    def __init__(self, rayon: float):
        self.__rayon = rayon

    def aire(self) -> float:
        import math
        return math.pi * self.__rayon ** 2

    def dessiner(self):
        print("Dessin du cercle")
```

Copier

C# :

```
public abstract class Forme
{
    public abstract double Aire();
    public void Afficher() => Console.WriteLine($"Aire : {Aire()}");
}

public interface IDessinable
{
    void Dessiner();
}

public class Cercle : Forme, IDessinable
{
    private double rayon;
    public Cercle(double rayon) { this.rayon = rayon; }

    public override double Aire() => Math.PI * rayon * rayon;
    public void Dessiner() => Console.WriteLine("Dessin du cercle");
}
```

Copier

PHP :

```
abstract class Forme {
    abstract public function aire(): float;

    public function afficher(): void {
        echo "Aire : " . $this->aire() . "\n";
    }
}

interface Dessinable {
    public function dessiner(): void;
}

class Cercle extends Forme implements Dessinable {
    public function __construct(private float $rayon) {}

    public function aire(): float {
        return M_PI * $this->rayon ** 2;
    }

    public function dessiner(): void {
        echo "Dessin du cercle\n";
    }
}
```

Copier

Les principes avancés de la POO

Encapsulation et abstraction

Gestion des données via des accesseurs (getters et setters)

Un **getter** (accesseur en lecture) est une méthode qui **retourne la valeur** d'un attribut privé. Un **setter** (accesseur en écriture) est une méthode qui **modifie la valeur** d'un attribut privé.

Pourquoi ne pas rendre les attributs publics directement ? Parce que les getters et setters ne sont pas de simples passe-plats : ils permettent de **valider**, **transformer** et **contrôler** l'accès aux données. Un setter peut refuser une valeur invalide (salaire négatif, email sans @), et un getter peut formater la donnée avant de la retourner.

> ***Analogie** : c'est comme un guichet de banque. Vous ne pouvez pas accéder directement au coffre (attribut privé). Vous passez par le guichetier (getter/setter) qui vérifie votre identité et la légitimité de votre demande avant d'agir.*

Java :

```
public class Employe {
    private String nom;
    private double salaire;

    public String getNom() { return nom; }
    public void setNom(String nom) {
        if (nom == null || nom.trim().isEmpty())
            throw new IllegalArgumentException("Nom invalide");
        this.nom = nom.trim();
    }

    public double getSalaire() { return salaire; }
    public void setSalaire(double salaire) {
        if (salaire < 0)
            throw new IllegalArgumentException("Salaire négatif interdit");
        this.salaire = salaire;
    }
}
```

Copier

Python :

```
class Employe:
    def __init__(self, nom: str, salaire: float):
        self.nom = nom         # Passe par le setter
        self.salaire = salaire

    @property
    def nom(self) -> str:
        return self.__nom

    @nom.setter
    def nom(self, valeur: str):
        if not valeur or not valeur.strip():
            raise ValueError("Nom invalide")
        self.__nom = valeur.strip()

    @property
    def salaire(self) -> float:
        return self.__salaire

    @salaire.setter
    def salaire(self, valeur: float):
        if valeur < 0:
            raise ValueError("Salaire négatif interdit")
        self.__salaire = valeur
```

Copier

PHP :

```
class Employe {
    private string $nom;
    private float $salaire;

    public function getNom(): string { return $this->nom; }
    public function setNom(string $nom): void {
        if (empty(trim($nom)))
            throw new InvalidArgumentException("Nom invalide");
        $this->nom = trim($nom);
    }

    public function getSalaire(): float { return $this->salaire; }
    public function setSalaire(float $salaire): void {
        if ($salaire < 0)
            throw new InvalidArgumentException("Salaire négatif interdit");
        $this->salaire = $salaire;
    }
}
```

Copier

Abstraction et simplification de la complexité

L'abstraction permet de **cacher la complexité** derrière une interface simple. L'utilisateur de la classe n'a pas besoin de connaître les détails d'implémentation.

> ***Analogie** : quand vous envoyez un SMS, vous tapez un message et appuyez sur « Envoyer ». Vous n'avez pas besoin de savoir que le téléphone convertit le texte en signal radio, le transmet à une antenne relais, qui le route vers le réseau de l'opérateur du destinataire, etc. Toute cette complexité est **abstraite** derrière un bouton simple.*

```
# L'utilisateur n'a pas besoin de connaître le protocole SMTP
class ServiceEmail(ABC):
    @abstractmethod
    def envoyer(self, destinataire: str, sujet: str, corps: str):
        pass

class ServiceEmailSmtp(ServiceEmail):
    def __init__(self, serveur: str, port: int):
        self.__serveur = serveur
        self.__port = port

    def envoyer(self, destinataire: str, sujet: str, corps: str):
        # Toute la complexité est cachée ici
        self.__connecter()
        self.__authentifier()
        self.__transmettre(destinataire, sujet, corps)
        self.__deconnecter()

    def __connecter(self): pass      # Détails cachés
    def __authentifier(self): pass
    def __transmettre(self, dest, sujet, corps): pass
    def __deconnecter(self): pass

# Utilisation simple
email = ServiceEmailSmtp("smtp.example.com", 587)
email.envoyer("user@example.com", "Bonjour", "Contenu du mail")
```

Copier

Héritage

Héritage simple et héritage multiple (selon le langage)

Langage	Héritage simple	Héritage multiple de classes	Alternative
Java	`extends`	Non	Interfaces (`implements`)
Python	`class Fille(Parent):`	Oui	MRO (Method Resolution Order)
C#	`:`	Non	Interfaces
PHP	`extends`	Non	Interfaces + Traits
C++	`:`	Oui	—

Héritage simple (universel) :

```
// Java
public class Animal {
    protected String nom;
    public Animal(String nom) { this.nom = nom; }
    public void parler() { System.out.println("..."); }
}

public class Chien extends Animal {
    public Chien(String nom) { super(nom); }

    @Override
    public void parler() { System.out.println("Wouf !"); }
}
```

Copier

```
# Python
class Animal:
    def __init__(self, nom: str):
        self.nom = nom

    def parler(self):
        print("...")

class Chien(Animal):
    def parler(self):
        print("Wouf !")
```

Copier

```php
// PHP
class Animal {
    public function __construct(protected string $nom) {}
    public function parler(): void { echo "...\n"; }
}

class Chien extends Animal {
    public function parler(): void { echo "Wouf !\n"; }
}
```

Copier

Héritage multiple (Python) :

```python
class Sportif:
    def entrainer(self):
        print(f"{self.nom} s'entraîne")

class Musicien:
    def jouer(self):
        print(f"{self.nom} joue de la musique")

class EtudiantPolyvalent(Animal, Sportif, Musicien):
    def __init__(self, nom: str, formation: str):
        super().__init__(nom)
        self.formation = formation

etudiant = EtudiantPolyvalent("Alice", "Informatique")
etudiant.entrainer()  # Alice s'entraîne
etudiant.jouer()      # Alice joue de la musique
```

Copier

Alternative en PHP (Traits) :

```php
trait SportifTrait {
    public function entrainer(): void {
        echo "{$this->nom} s'entraîne\n";
    }
}

trait MusicienTrait {
    public function jouer(): void {
        echo "{$this->nom} joue de la musique\n";
    }
}

class EtudiantPolyvalent extends Animal {
    use SportifTrait, MusicienTrait;

    public function __construct(string $nom, private string $formation) {
        parent::__construct($nom);
    }
}
```

Copier

Surcharge et redéfinition des méthodes (overloading / overriding)

Ce sont deux mécanismes souvent confondus mais très différents :

- **Surcharge (overloading)** : avoir **plusieurs méthodes avec le même nom** mais des **paramètres différents** dans la même classe. Le compilateur choisit la bonne version selon les arguments passés. C'est comme un restaurant qui propose le menu « du jour » en version entrée+plat, ou entrée+plat+dessert — même nom, options différentes.
- **Redéfinition (overriding)** : une classe fille **remplace** le comportement d'une méthode héritée de la classe parente. Le choix de la version se fait **à l'exécution** selon le type réel de l'objet. C'est comme un enfant qui apprend la recette de ses parents mais la cuisine à sa façon.

Concept	Surcharge (Overloading)	Redéfinition (Overriding)
Où ?	Même classe	Classe fille
Signature	Même nom, paramètres différents	Même nom et mêmes paramètres
Résolution	À la compilation (statique)	À l'exécution (dynamique)
Support	Java, C#, C++ (natif) / Python, PHP (simulation)	Tous les langages objet

Surcharge (Java / C#) :

```
public class Calculatrice {
    public int calculer(int a, int b) { return a + b; }
    public double calculer(double a, double b) { return a + b; }
    public int calculer(int a, int b, int c) { return a + b + c; }
}
```

Copier

Surcharge simulée (Python) :

```
class Calculatrice:
    def calculer(self, a, b, c=None):
        if c is not None:
            return a + b + c
        return a + b
```

Copier

Redéfinition (universelle) :

```
// Java
public class CalculatriceScientifique extends Calculatrice {
    @Override
    public int calculer(int a, int b) {
        System.out.println("Calcul scientifique");
        return super.calculer(a, b);
    }
}
```

Copier

```python
# Python
class CalculatriceScientifique(Calculatrice):
    def calculer(self, a, b, c=None):
        print("Calcul scientifique")
        return super().calculer(a, b, c)
```

Copier

Polymorphisme

Polymorphisme statique et dynamique

Le mot **polymorphisme** vient du grec : *poly* (plusieurs) + *morphê* (formes). En programmation, cela signifie qu'**une même action peut prendre plusieurs formes** selon le contexte.

- **Polymorphisme statique (surcharge)** : résolu **à la compilation**. Le compilateur sait à l'avance quelle version de la méthode appeler grâce aux types des paramètres. C'est la surcharge de méthodes vue précédemment.
- **Polymorphisme dynamique (redéfinition)** : résolu **à l'exécution**. Le programme ne sait qu'au moment de l'exécution quel type d'objet réel est manipulé, et appelle la bonne version de la méthode. C'est le cœur de la puissance de la POO.

> ***Exemple concret** : vous avez une liste de `Forme` (cercles, rectangles, triangles). Quand vous appelez `aire()` sur chaque élément, **le programme sait à l'exécution** quel calcul utiliser : π×r² pour un cercle, l×h pour un rectangle, etc. Vous n'avez pas besoin d'écrire un `if` pour chaque type — le polymorphisme s'en charge.*

Exemple de polymorphisme dynamique (multi-langages) :

```java
// Java
Forme[] formes = { new Cercle(5), new Rectangle(4, 6) };
for (Forme f : formes) {
    System.out.println("Aire : " + f.aire()); // Appelle la bonne méthode
}
```

Copier

```python
# Python
formes = [Cercle(5), Rectangle(4, 6)]
for f in formes:
    print(f"Aire : {f.aire()}")  # Appelle la bonne méthode
```

Copier

```csharp
// C#
Forme[] formes = { new Cercle(5), new Rectangle(4, 6) };
foreach (var f in formes)
    Console.WriteLine($"Aire : {f.Aire()}");
```

Copier

```php
// PHP
$formes = [new Cercle(5), new Rectangle(4, 6)];
foreach ($formes as $f) {
    echo "Aire : " . $f->aire() . "\n";
}
```

Copier

Utilisation des classes abstraites et des interfaces pour le polymorphisme

Critère	Classe abstraite	Interface
Héritage	Simple uniquement (sauf Python)	Multiple possible
Attributs	Oui (avec état)	Constantes uniquement
Constructeur	Oui	Non
Méthodes concrètes	Oui	Oui (défaut en Java 8+, PHP 8+, C# 8+)
Utilisation	Relation « est un » avec état commun	Contrat / capacité

Règle de choix :

- **Classe abstraite** → les sous-classes partagent un **état commun** (ex. `Vehicule` → `Voiture`, `Moto`).
- **Interface** → on définit une **capacité** (ex. `Serializable`, `Comparable`, `Exportable`).

Exemple concret :

```java
// Java — Interface pour le polymorphisme
public interface Exportable {
    String exporter();
}

public class Rapport implements Exportable {
    private String contenu;
    public Rapport(String contenu) { this.contenu = contenu; }

    @Override
    public String exporter() { return "PDF: " + contenu; }
}

public class Facture implements Exportable {
    private double montant;
    public Facture(double montant) { this.montant = montant; }

    @Override
    public String exporter() { return "CSV: " + montant + "€"; }
}

// Polymorphisme via l'interface
List<Exportable> documents = List.of(new Rapport("Bilan"), new Facture(1500));
for (Exportable doc : documents) {
    System.out.println(doc.exporter());
}
```

Copier

```python
# Python — Même concept
class Exportable(ABC):
    @abstractmethod
    def exporter(self) -> str:
        pass

class Rapport(Exportable):
    def __init__(self, contenu: str):
        self.contenu = contenu

    def exporter(self) -> str:
        return f"PDF: {self.contenu}"

class Facture(Exportable):
    def __init__(self, montant: float):
        self.montant = montant

    def exporter(self) -> str:
        return f"CSV: {self.montant}€"

documents = [Rapport("Bilan"), Facture(1500)]
for doc in documents:
    print(doc.exporter())
```

Copier

Conception et bonnes pratiques

Les design patterns (patrons de conception)

Un **design pattern** (ou patron de conception) est une **solution standard et réutilisable** à un problème de conception que l'on rencontre fréquemment en développement logiciel. Ce ne sont **pas des morceaux de code prêts à copier-coller**, mais plutôt des **schémas de pensée** — des façons éprouvées d'organiser ses classes et leurs interactions pour résoudre un type de problème donné.

> ***Analogie** : en architecture, il existe des « patrons » comme « l'escalier en colimaçon » ou « la cour intérieure ». Ce ne sont pas des plans exacts, mais des **concepts** que chaque architecte adapte à son bâtiment. En programmation, c'est pareil : un design pattern décrit **comment** structurer le code, mais l'implémentation varie selon le langage et le projet.*

Pourquoi les utiliser ?

- Éviter de « réinventer la roue » face à un problème classique.
- Avoir un **vocabulaire commun** entre développeurs (dire « on utilise un Singleton » est plus rapide que de décrire tout le mécanisme).
- Produire du code plus **maintenable** et **compréhensible** par d'autres développeurs.

Il existe des dizaines de design patterns, regroupés en trois familles :

- **Créationnels** : comment créer des objets (Singleton, Factory…).
- **Structurels** : comment organiser les classes entre elles (Adapter, Decorator…).
- **Comportementaux** : comment les objets communiquent (Observer, Strategy…).

Voici les quatre patterns les plus courants et les plus utiles à connaître.

Singleton

But : garantir qu'une classe n'a qu'**une seule instance** dans tout le programme et fournir un point d'accès global à cette instance.

Quand l'utiliser ? Quand il ne doit exister qu'**un seul exemplaire** d'un objet : une configuration globale, un logger (journal de logs), une connexion à un cache, un gestionnaire de thème, etc.

> ***Analogie** : dans une entreprise, il n'y a qu'**un seul PDG** à la fois. Quand quelqu'un demande « le PDG », tout le monde sait de qui on parle — c'est toujours la même personne. Le Singleton fonctionne pareil : une seule instance, accessible partout.*

Java :

```
public class Configuration {
    private static Configuration instance;
    private Map<String, String> parametres = new HashMap<>();

    private Configuration() {} // Constructeur privé

    public static synchronized Configuration getInstance() {
        if (instance == null) instance = new Configuration();
        return instance;
    }

    public String get(String cle) { return parametres.get(cle); }
    public void set(String cle, String val) { parametres.put(cle, val); }
}
```

Copier

Python :

```
class Configuration:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.parametres = {}
        return cls._instance

    def get(self, cle): return self.parametres.get(cle)
    def set(self, cle, val): self.parametres[cle] = val

# Même instance à chaque appel
config1 = Configuration()
config2 = Configuration()
assert config1 is config2  # True
```

Copier

PHP :

```
class Configuration {
    private static ?Configuration $instance = null;
    private array $parametres = [];

    private function __construct() {}

    public static function getInstance(): self {
        if (self::$instance === null) self::$instance = new self();
        return self::$instance;
    }

    public function get(string $cle): ?string { return $this->parametres[$cle] ?? null; }
    public function set(string $cle, string $val): void { $this->parametres[$cle] = $val; }
}
```

Copier

Factory (Fabrique)

But : déléguer la **création d'objets** à une méthode ou une classe spécialisée, sans que le code appelant ait besoin de savoir **quelle classe concrète** est instanciée.

Quand l'utiliser ? Quand la création d'un objet dépend d'une condition (un type, un paramètre, une configuration) et qu'on veut **centraliser** cette logique au lieu de mettre des `if/else` ou des `switch` partout dans le code.

> ***Analogie** : dans une pizzeria, vous dites au comptoir « une Margherita ». Vous ne préparez pas la pizza vous-même — c'est la **cuisine** (la Factory) qui s'en charge. Vous ne savez pas quel pizzaïolo la fait ni quels fours sont utilisés. Vous recevez juste votre pizza prête. La Factory fonctionne pareil : vous demandez un objet par son type, et elle vous le retourne sans que vous ayez à connaître les détails de sa création.*

Java :

```
public class FormeFactory {
    public static Forme creer(String type) {
        return switch (type.toLowerCase()) {
            case "cercle"    -> new Cercle(1);
            case "rectangle" -> new Rectangle(1, 1);
            case "triangle"  -> new Triangle(1, 1);
            default -> throw new IllegalArgumentException("Type inconnu : " + type);
        };
    }
}

Forme f = FormeFactory.creer("cercle");
```

Copier

Python :

```
class FormeFactory:
    @staticmethod
    def creer(type_forme: str) -> Forme:
        formes = {
            "cercle": lambda: Cercle(1),
            "rectangle": lambda: Rectangle(1, 1),
            "triangle": lambda: Triangle(1, 1),
        }
        if type_forme.lower() not in formes:
            raise ValueError(f"Type inconnu : {type_forme}")
        return formes[type_forme.lower()]()
```

Copier

PHP :

```
class FormeFactory {
    public static function creer(string $type): Forme {
        return match(strtolower($type)) {
            'cercle'    => new Cercle(1),
            'rectangle' => new Rectangle(1, 1),
            'triangle'  => new Triangle(1, 1),
            default      => throw new InvalidArgumentException("Type inconnu : $type"),
        };
    }
}
```

Copier

Observer (Observateur)

But : définir une dépendance **un-à-plusieurs** entre objets : quand un objet (le « sujet ») change d'état, tous les objets qui l'« observent » sont **notifiés automatiquement** et peuvent réagir.

Quand l'utiliser ? Quand plusieurs parties du programme doivent **réagir** à un événement sans que l'émetteur de l'événement ait besoin de les connaître. C'est le mécanisme derrière les systèmes d'événements, les notifications push, les mises à jour d'interface en temps réel.

> ***Analogie** : c'est comme s'**abonner** à une chaîne YouTube. Quand le créateur publie une vidéo (changement d'état), tous les abonnés (observateurs) reçoivent une notification, sans que le créateur ait besoin de leur envoyer un message individuel. Si vous vous désabonnez, vous ne recevez plus rien — sans que cela affecte les autres abonnés.*

Python :

```
class SujetObservable:
    def __init__(self):
        self._observateurs = []

    def ajouter(self, obs):
        self._observateurs.append(obs)

    def notifier(self, evenement, donnees=None):
        for obs in self._observateurs:
            obs.mise_a_jour(evenement, donnees)

class Boutique(SujetObservable):
    def __init__(self):
        super().__init__()
        self.produits = []

    def ajouter_produit(self, produit: str):
        self.produits.append(produit)
        self.notifier("nouveau_produit", produit)

class ClientNotification:
    def __init__(self, nom: str):
        self.nom = nom

    def mise_a_jour(self, evenement, donnees):
        print(f"{self.nom} notifié : {evenement} -> {donnees}")

# Utilisation
boutique = Boutique()
boutique.ajouter(ClientNotification("Alice"))
boutique.ajouter(ClientNotification("Bob"))
boutique.ajouter_produit("Laptop")
# Alice notifié : nouveau_produit -> Laptop
# Bob notifié : nouveau_produit -> Laptop
```

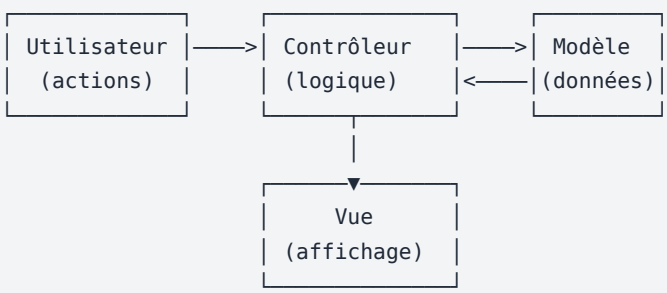
Copier

MVC (Modèle-Vue-Contrôleur)

But : séparer l'application en **trois couches distinctes** pour que chacune puisse évoluer indépendamment. C'est le pattern le plus utilisé dans le développement web (Laravel, Symfony, Django, Spring, ASP.NET...).

- **Modèle (Model)** : les **données** et la **logique métier**. C'est le cœur de l'application — il ne sait rien de l'affichage.
- **Vue (View)** : l'**affichage** et l'interface utilisateur. Elle ne fait que montrer les données, sans logique métier.
- **Contrôleur (Controller)** : le **chef d'orchestre** qui reçoit les actions de l'utilisateur, appelle le modèle, et choisit quelle vue afficher.

> ***Analogie** : dans un restaurant, le **Modèle** c'est la cuisine (les données et les recettes), la **Vue** c'est l'assiette présentée au client (l'affichage), et le **Contrôleur** c'est le serveur (il prend la commande du client, la transmet à la cuisine, et apporte le plat).*



Copier

Exemple PHP (typique des frameworks Laravel, Symfony) :

```
// Modèle
class Utilisateur {
    public function __construct(
        private string $nom,
        private string $email
    ) {}

    public function getNom(): string { return $this->nom; }
    public function setNom(string $nom): void { $this->nom = $nom; }
    public function getEmail(): string { return $this->email; }
}

// Vue
class UtilisateurVue {
    public function afficher(Utilisateur $u): void {
        echo "Nom : {$u->getNom()} | Email : {$u->getEmail()}\n";
    }
}

// Contrôleur
class UtilisateurControleur {
    public function __construct(
        private Utilisateur $modele,
        private UtilisateurVue $vue
    ) {}

    public function mettreAJourNom(string $nom): void {
        $this->modele->setNom($nom);
    }

    public function afficher(): void {
        $this->vue->afficher($this->modele);
    }
}
```

Copier

Quand et comment les utiliser

Pattern	Quand l'utiliser	Exemple concret
Singleton	Une seule instance nécessaire (config, logger, cache)	Connexion BDD, configuration globale
Factory	Création d'objets complexe ou variable selon le contexte	Parseur de fichiers (CSV, JSON, XML)
Observer	Notification automatique lors de changements	Événements UI, notifications temps réel
MVC	Séparer affichage, logique et données	Applications web (Laravel, Django, Spring)

Principes SOLID

SOLID est un acronyme qui regroupe **cinq principes de conception** formulés par Robert C. Martin (« Uncle Bob »). Ces principes ne sont pas des règles absolues, mais des **guides** pour écrire du code qui sera facile à **maintenir**, **tester** et **faire évoluer** dans le temps.

> ***Pourquoi c'est important ?*** *Sur un petit projet, on peut se permettre de tout mettre dans quelques classes. Mais dès que le projet grandit (et il grandit toujours), un code mal structuré devient un cauchemar à maintenir. Les principes SOLID aident à* ***anticiper*** *ce problème.*

S — Single Responsibility (Responsabilité unique)

Chaque classe ne doit avoir qu'une seule raison de changer. Autrement dit, une classe doit faire **une seule chose** et la faire bien.

> ***Analogie*** *: dans un hôpital, le chirurgien opère, l'anesthésiste gère l'anesthésie, et l'infirmier assure les soins post-opératoires. Chacun a* ***un seul rôle****. Si le chirurgien devait aussi gérer l'anesthésie et les soins, le risque d'erreur serait bien plus élevé.*

```
# MAUVAIS : la classe fait tout
class GestionUtilisateur:
    def creer_utilisateur(self, nom): pass
    def envoyer_email(self, email): pass
    def generer_rapport(self): pass

# BON : une responsabilité par classe
class UtilisateurService:
    def creer_utilisateur(self, nom): pass

class EmailService:
    def envoyer_email(self, email): pass

class RapportService:
    def generer_rapport(self): pass
```

Copier

O — Open/Closed (Ouverture/Fermeture)

Ouvert à l'extension, fermé à la modification. On doit pouvoir **ajouter** de nouveaux comportements sans **modifier** le code existant qui fonctionne déjà.

> ***Analogie*** *: une* ***multiprise*** *est ouverte à l'extension (on peut y brancher de nouveaux appareils) mais fermée à la modification (on ne démonte pas la multiprise pour ajouter une prise). En code, on ajoute de nouvelles classes qui implémentent une interface, sans toucher aux classes existantes.*

```php
// BON : ajouter un type de remise = créer une nouvelle classe
interface StrategieRemise {
    public function appliquer(float $prix): float;
}

class RemisePourcentage implements StrategieRemise {
    public function __construct(private float $pourcentage) {}
    public function appliquer(float $prix): float {
        return $prix * (1 - $this->pourcentage / 100);
    }
}

class RemiseMontantFixe implements StrategieRemise {
    public function __construct(private float $montant) {}
    public function appliquer(float $prix): float {
        return max(0, $prix - $this->montant);
    }
}

// Nouveau type ? On ajoute une classe, on ne modifie rien
class RemiseBlackFriday implements StrategieRemise {
    public function appliquer(float $prix): float {
        return $prix * 0.5;
    }
}
```

Copier

L — Liskov Substitution (Substitution de Liskov)

Une classe fille doit pouvoir remplacer sa classe parente sans casser le programme. Si le code fonctionne avec un objet de type `Parent`, il doit aussi fonctionner avec un objet de type `Enfant` sans surprise.

> ***Analogie** : si vous avez un permis de conduire (classe `Vehicule`), vous devez pouvoir conduire n'importe quelle voiture — une berline, un SUV, une citadine. Si un SUV avait soudainement le volant inversé (comportement inattendu), le permis ne serait plus fiable. Liskov dit que les sous-types doivent **respecter le contrat** du type parent.*

```
# MAUVAIS : Carre casse le contrat de Rectangle
class RectangleSimple:
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur

    def aire(self):
        return self.largeur * self.hauteur

class Carre(RectangleSimple):
    def __init__(self, cote):
        super().__init__(cote, cote)
    # Si on ajoute un setter qui synchronise largeur/hauteur,
    # le code client qui utilise un Rectangle sera surpris

# BON : abstraction commune sans héritage trompeur
class Forme2D(ABC):
    @abstractmethod
    def aire(self) -> float: pass

class Rectangle(Forme2D):
    def __init__(self, l, h): self.l, self.h = l, h
    def aire(self): return self.l * self.h

class Carre(Forme2D):
    def __init__(self, c): self.c = c
    def aire(self): return self.c ** 2
```

Copier

I — Interface Segregation (Ségrégation d'interface)

Plusieurs petites interfaces spécialisées plutôt qu'une seule interface « couteau suisse ». Une classe ne doit pas être forcée d'implémenter des méthodes dont elle n'a pas besoin.

> ***Analogie** : imaginez un contrat de travail qui exige que **tous** les employés sachent programmer, faire la comptabilité, cuisiner et conduire un camion. C'est absurde — chaque poste a ses propres compétences. De même, chaque interface doit être **ciblée** sur un ensemble cohérent de méthodes.*

```
// MAUVAIS
public interface Machine {
    void imprimer();
    void scanner();
    void faxer();
}

// BON
public interface Imprimante { void imprimer(); }
public interface Scanner { void scanner(); }
public interface Fax { void faxer(); }

// Chaque classe implémente uniquement ce dont elle a besoin
public class ImprimanteSimple implements Imprimante {
    public void imprimer() { System.out.println("Impression..."); }
}

public class Multifonction implements Imprimante, Scanner, Fax {
    public void imprimer() { System.out.println("Impression..."); }
    public void scanner() { System.out.println("Scan..."); }
    public void faxer() { System.out.println("Fax..."); }
}
```

Copier

D — Dependency Inversion (Inversion des dépendances)

Dépendre d'abstractions (interfaces), pas d'implémentations concrètes. Les modules de haut niveau (la logique métier) ne doivent pas dépendre directement des modules de bas niveau (la base de données, l'envoi d'emails...). Les deux doivent dépendre d'une **interface commune**.

> ***Analogie** : votre téléphone utilise un **chargeur USB-C** (l'abstraction). Peu importe le fabricant du câble ou de l'adaptateur secteur (l'implémentation concrète), tant qu'il respecte le standard USB-C, ça fonctionne. Si votre téléphone était soudé à un seul chargeur précis, ce serait très peu pratique — c'est exactement ce qu'on évite avec ce principe.*

```
# MAUVAIS : couplage fort
class CommandeService:
    def __init__(self):
        self.db = BaseDeDonneesMySQL()  # Dépendance directe

    def sauvegarder(self, commande):
        self.db.inserer(commande)

# BON : injection de dépendance
class Repository(ABC):
    @abstractmethod
    def sauvegarder(self, entite): pass

    @abstractmethod
    def trouver_par_id(self, id): pass

class CommandeRepositoryMySQL(Repository):
    def sauvegarder(self, entite): pass  # INSERT MySQL
    def trouver_par_id(self, id): pass   # SELECT MySQL

class CommandeRepositoryMemoire(Repository):
    def __init__(self): self.data = {}
    def sauvegarder(self, entite): self.data[entite.id] = entite
    def trouver_par_id(self, id): return self.data.get(id)

class CommandeService:
    def __init__(self, repository: Repository):
        self.repository = repository  # Dépend de l'abstraction

    def creer(self, commande):
        self.repository.sauvegarder(commande)

# Facile de changer l'implémentation
service = CommandeService(CommandeRepositoryMySQL())
service_test = CommandeService(CommandeRepositoryMemoire())
```

Copier

Éviter les anti-patterns

Un **anti-pattern**, c'est l'inverse d'un design pattern : c'est une **mauvaise pratique** qui semble être une bonne idée sur le moment, mais qui crée des problèmes à long terme (code difficile à maintenir, bugs difficiles à trouver, impossibilité de faire évoluer le projet).

> ***Analogie** : c'est comme ranger toutes ses affaires dans un seul tiroir pour « gagner du temps ». Au début ça va, mais au bout de quelques semaines, retrouver quoi que ce soit devient un calvaire. Les anti-patterns fonctionnent pareil : un raccourci aujourd'hui devient une dette technique demain.*

Voici les anti-patterns les plus fréquents à reconnaître et à éviter :

God Object (Objet divin)

Problème : une classe qui fait **tout** et connaît **tout**. Elle gère les utilisateurs, les emails, les paiements, les rapports... C'est la « classe à tout faire » qui finit par contenir des centaines ou des milliers de lignes.

Pourquoi c'est grave ? Modifier quoi que ce soit dans cette classe risque de casser autre chose. Impossible de la tester unitairement. Impossible de travailler dessus à plusieurs.

Solution : découper en classes spécialisées (principe SRP — une responsabilité par classe).

Spaghetti Code

Problème : code **sans structure**, où les dépendances sont emmêlées dans tous les sens, la logique est impossible à suivre, et une modification à un endroit provoque des bugs ailleurs. Le nom vient de l'image de spaghettis entremêlés dans une assiette — impossible de tirer un fil sans en emporter dix autres.

Pourquoi c'est grave ? Le code devient **illisible**, même pour celui qui l'a écrit. Corriger un bug en crée deux nouveaux. Ajouter une fonctionnalité prend 10 fois plus de temps que prévu.

Solution : appliquer les principes SOLID, découper en classes et méthodes avec des responsabilités claires, et séparer les couches (présentation, logique métier, accès aux données).

Code dupliqué (copier-coller)

Problème : le même bloc de code apparaît **à plusieurs endroits** dans le projet (souvent issu d'un copier-coller). Quand il faut corriger un bug dans ce code, il faut le corriger **partout** — et on en oublie toujours un.

Pourquoi c'est grave ? Si la logique dupliquée doit changer, il faut retrouver et modifier **chaque copie**. Oublier une seule copie introduit un comportement incohérent et des bugs difficiles à tracer.

Solution : extraire le code commun dans une **méthode**, une **classe utilitaire** ou une **classe parente**. C'est le principe DRY : *Don't Repeat Yourself* (ne te répète pas).

```
# MAUVAIS : duplication
class ClientService:
    def valider_email(self, email):
        return email and "@" in email and "." in email

class FournisseurService:
    def valider_email(self, email):
        return email and "@" in email and "." in email

# BON : extraction
class Validateur:
    @staticmethod
    def valider_email(email: str) -> bool:
        return bool(email and "@" in email and "." in email)
```

Copier

Magic Numbers (nombres magiques)

Problème : des valeurs numériques apparaissent directement dans le code **sans aucune explication** de ce qu'elles représentent. Quand on lit `if (age > 18)` ou `prix * 1.20`, on peut deviner… mais avec `if (status == 3)` ou `timeout = 86400`, c'est impossible sans contexte.

Pourquoi c'est grave ? Le code devient **incompréhensible** pour les autres développeurs

(et pour soi-même dans 3 mois). Si la valeur doit changer, il faut la chercher partout dans le code.

Solution : remplacer les valeurs numériques par des **constantes nommées** qui expliquent leur signification.

```
// MAUVAIS
if ($age > 18) { /* ... */ }
$prix = $montant * 1.20;

// BON
const AGE_MAJORITE = 18;
const TAUX_TVA = 1.20;

if ($age > AGE_MAJORITE) { /* ... */ }
$prix = $montant * TAUX_TVA;
```

Copier

Environnement de développement

Choix du langage objet

Le choix du langage dépend du **contexte du projet** et des **besoins spécifiques** :

Langage	Points forts	Cas d'usage typique
Java	Robuste, multiplateforme, écosystème mature	Applications d'entreprise, Android, API REST
C#	Intégration Microsoft, LINQ, .NET	Applications Windows, jeux (Unity), web (ASP.NET)
Python	Simplicité, polyvalence, bibliothèques riches	Data science, scripts, prototypage rapide, web (Django/Flask)
PHP	Web natif, large communauté, frameworks puissants	Sites web, CMS, API (Laravel, Symfony)
TypeScript	Typage fort sur JavaScript, fullstack	Applications web modernes (Angular, React, Node.js)
C++	Performance, contrôle mémoire	Systèmes embarqués, jeux AAA, moteurs graphiques
Kotlin	Moderne, interopérable Java, concis	Android, backend (Spring Boot)
Swift	Sécurisé, performant, syntaxe moderne	Applications iOS/macOS

Outils recommandés

Catégorie	Outils
IDE	IntelliJ IDEA (Java/Kotlin), Visual Studio (C#), VS Code (polyvalent), PhpStorm (PHP), PyCharm (Python)
Dépendances	Maven/Gradle (Java), Composer (PHP), pip/Poetry (Python), npm/yarn (TypeScript), NuGet (C#)
Versioning	Git avec GitHub, GitLab ou Bitbucket
Tests	JUnit (Java), PHPUnit (PHP), pytest (Python), Jest (TypeScript), NUnit (C#)
CI/CD	GitHub Actions, GitLab CI, Jenkins
Documentation	Javadoc, Sphinx, phpDocumentor, Swagger/OpenAPI

Récapitulatif

Concept	Description
Encapsulation	Cacher les détails internes, exposer via des méthodes contrôlées
Héritage	Réutiliser et spécialiser le comportement d'une classe parente
Polymorphisme	Un même appel, des comportements différents selon le type réel
Abstraction	Ne montrer que l'essentiel, masquer la complexité
SOLID	5 principes pour un code maintenable et extensible
Design Patterns	Solutions éprouvées aux problèmes récurrents de conception
Anti-patterns	Pratiques à éviter pour garder un code sain

La maîtrise de la POO est **universelle** : ces concepts s'appliquent quel que soit le langage choisi et permettent de construire des applications **robustes, maintenables et évolutives**.