Introduction au langage Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 La modélisation objet 1
2 Les objets 2
3 Les messages 2
4 Les classes - Définition 2
5 Concept de Classe dans les langages de programmation 3
6 UML 4
  6.1 Java 4
  6.2 Python 4
  6.3 Kotlin 5
  6.4 Typescript 6
  6.5 Javascript 6
7 Les classes - Utilisation 8
8 Origines et Motivations de la Création du Langage Java 8
9 Les caractéristiques de Java 9
10 Du code source à l'exécution 9
11 Code source et bytecode 9
12 Implantation et Versions 10
13 Environnement de Développement Java 10
14 Java en résumé 11

1 La modélisation objet

La Programmation Orientée Objet (POO) trouve ses origines dans les problèmes de simulation des années 1960. Elle est largement utilisée dans divers domaines tels que la modélisation (UML), la programmation (Smalltalk, C++, Java, etc.) et les bases de données (SGBDOO, bien que presque disparus, et SGBDRO). Le concept majeur de la POO est l'encapsulation, où les objets communiquent entre eux via un ensemble de messages prédéfinis sans connaître leur implantation interne. Les objets peuvent être regroupés en classes, et d'autres concepts clés tels que la composition, la

délégation et l'héritage enrichissent cette approche. Un objet peut être perçu sous deux angles : structurel, comme une instance d'un type masquant une structure derrière des opérations, et conceptuel, correspondant à une entité du monde réel pouvant être généralisée ou spécialisée.

2 Les objets

Définition 1 (Objet). Un objet en programmation orientée objet représente une entité avec une identité unique, un état défini par des attributs, et un comportement décrit par des méthodes.

Les méthodes permettent à l'objet de réagir aux messages qu'il reçoit, et ces réactions peuvent modifier son état interne. Les attributs, nommés et valorisés, décrivent l'état interne de l'objet. L'encapsulation est un principe clé qui cache cet état interne et fournit des méthodes pour y accéder, ce qui améliore à la fois la sécurité et l'évolutivité du code. En encapsulant les données, on protège les informations internes des accès non autorisés et on permet de modifier l'implémentation interne sans impacter le code externe qui utilise l'objet. Cependant, cette encapsulation peut introduire une légère surcharge de performance en raison des méthodes d'accès et de modification des attributs.

3 Les messages

En programmation orientée objet, chaque objet possède un ensemble de méthodes qui peuvent être soit publiques, soit privées. Les méthodes publiques constituent l'interface de l'objet, définissant l'ensemble des messages qu'il peut recevoir et traiter. Par exemple, considérez un objet `unePersonne` avec les méthodes suivantes : `saluer()`, `donnerPrénom(String prénom)`, `afficherPrénom()`, et `retournerPrénom()`. Voici comment ces méthodes peuvent être utilisées :

- `unePersonne.saluer();` permet à l'objet de saluer.
- `unePersonne.donnerPrénom("Pierre");` attribue le prénom « Pierre » à l'objet.
- `unePersonne.afficherPrénom();` affiche le prénom de l'objet.
- `unePersonne.retournerPrénom();` retourne le prénom de l'objet.

4 Les classes - Définition

Certains objets possèdent des propriétés très proches, voire identiques. Par exemple, les voitures dans un programme peuvent partager des caractéristiques communes telles que la marque, le modèle, l'année et la couleur.

Une classe est un ensemble d'objets présentant des similitudes en termes d'identification, de comportement, ou de description de l'état dans une application. Une **instance d'une classe** est un objet qui appartient à cette classe.

La classe Voiture peut être définie avec les attributs suivants : la marque, le modèle, l'année et la couleur. Elle peut également inclure des méthodes telles que démarrer, arrêter et accélérer. Instances de la Classe Voiture Prenons deux exemples d'instances de la classe Voiture : La première instance, voiture1, a pour marque « Toyota », pour modèle « Corolla », pour année 2020 et pour couleur « Rouge ». La deuxième instance, voiture2, a pour marque « Honda », pour modèle « Civic », pour année 2018 et pour couleur « Bleue ». Ces instances partagent les mêmes attributs et méthodes définis par la classe Voiture, mais avec des valeurs spécifiques à chaque instance.

5 Concept de Classe dans les langages de programmation

Chaque langage de programmation et modèle de conception comme UML présente des choix distincts en ce qui concerne le concept de classes, influencés par leurs principes de conception et objectifs spécifiques.

En UML, les classes sont utilisées pour modéliser des entités du monde réel ou des concepts abstraits. Elles définissent les propriétés (attributs) et comportements (méthodes) des objets qui en sont des instances. UML permet de représenter la structure et les relations entre les classes à l'aide de diagrammes.

Java, un langage orienté objet par excellence, met l'accent sur l'encapsulation, l'héritage et le polymorphisme à travers ses classes. Les classes Java sont des structures fondamentales pour organiser le code en modules réutilisables, encapsulant étroitement les données et les méthodes pour contrôler l'accès à celles-ci.

Python utilise également des classes pour créer des objets, mais de manière plus flexible grâce à sa nature dynamique. Les classes Python permettent l'héritage multiple, le polymorphisme et la surcharge d'opérateurs, tout en soutenant les méthodes spéciales pour une personnalisation avancée du comportement des objets.

JavaScript et TypeScript, bien que différents dans leur typage et structure, partagent le concept de classes introduit avec ECMAScript 6 (ES6). Les classes JavaScript simplifient la syntaxe de création d'objets en utilisant le mot-clé class, tout en restant fondamentalement basées sur des prototypes. TypeScript étend ce concept en ajoutant un typage statique aux classes JavaScript, offrant ainsi des fonctionnalités supplémentaires de vérification de types et d'autocomplétion dans les environnements de développement.

6 UML

6.1 Java

```java
public class Voiture {
    private String marque;
    private String modele;
    private int annee;
    private String couleur;
    private static int nombreDeVoitures = 0;

    public Voiture(String marque, String modele, int annee, String couleur) {
        this.marque = marque;
        this.modele = modele;
        this.annee = annee;
        this.couleur = couleur;
        nombreDeVoitures++;
    }

    public void demarrer() {
        System.out.println(marque + " " + modele + " démarre.");
    }

    public void arreter() {
        System.out.println(marque + " " + modele + " s'arrête.");
    }

    public void accelerer(int vitesse) {
        System.out.println(marque + " " + modele + " accélère à " + vitesse + "
km/h.");
    }

    public static int getNombreDeVoitures() {
        return nombreDeVoitures;
    }
}
```

6.2 Python

```python
class Voiture:
    nombre_de_voitures = 0

    def __init__(self, marque, modele, annee, couleur):
        self.marque = marque
```

```python
        self.modele = modele
        self.annee = annee
        self.couleur = couleur
        Voiture.nombre_de_voitures += 1

    def demarrer(self):
        print(f"{self.marque} {self.modele} démarre.")

    def arreter(self):
        print(f"{self.marque} {self.modele} s'arrête.")

    def accelerer(self, vitesse):
        print(f"{self.marque} {self.modele} accélère à {vitesse} km/h.")

    @classmethod
    def get_nombre_de_voitures(cls):
        return cls.nombre_de_voitures
```

6.3 Kotlin

```kotlin
class Voiture(
    private val marque: String,
    private val modele: String,
    private val annee: Int,
    private val couleur: String
) {
    companion object {
        var nombreDeVoitures: Int = 0
            private set
    }

    init {
        nombreDeVoitures++
    }

    fun demarrer() {
        println("$marque $modele démarre.")
    }

    fun arreter() {
        println("$marque $modele s'arrête.")
    }

    fun accelerer(vitesse: Int) {
```

```
        println("$marque $modele accélère à $vitesse km/h.")
    }
}
```

6.4 Typescript

```typescript
class Voiture {
    private static nombreDeVoitures: number = 0;

    constructor(
        private marque: string,
        private modele: string,
        private annee: number,
        private couleur: string
    ) {
        Voiture.nombreDeVoitures++;
    }

    public demarrer(): void {
        console.log(`${this.marque} ${this.modele} démarre.`);
    }

    public arreter(): void {
        console.log(`${this.marque} ${this.modele} s'arrête.`);
    }

    public accelerer(vitesse: number): void {
       console.log(`${this.marque} ${this.modele} accélère à ${vitesse} km/h.`);
    }

    public static getNombreDeVoitures(): number {
        return Voiture.nombreDeVoitures;
    }
}
```

6.5 Javascript

```javascript
// Constructeur Voiture
function Voiture(marque, modele, annee, couleur) {
    this.marque = marque;
    this.modele = modele;
    this.annee = annee;
    this.couleur = couleur;
    Voiture.nombreDeVoitures++;
```

```
}

// Propriété statique pour compter le nombre de voitures
Voiture.nombreDeVoitures = 0;

// Méthodes de la classe Voiture
Voiture.prototype.demarrer = function() {
    console.log(this.marque + " " + this.modele + " démarre.");
};

Voiture.prototype.arreter = function() {
    console.log(this.marque + " " + this.modele + " s'arrête.");
};

Voiture.prototype.accelerer = function(vitesse) {
    console.log(this.marque + " " + this.modele + " accélère à " + vitesse + "
km/h.");
};

// Méthode statique pour obtenir le nombre de voitures
Voiture.getNombreDeVoitures = function() {
    return Voiture.nombreDeVoitures;
};
```

Dans la programmation orientée objet, on remarque souvent que certains objets possèdent des propriétés très proches, voire identiques. Par exemple, dans un programme, les propriétés des voitures, telles que la marque, le modèle, l'année et la couleur, sont communes. De manière plus générale, ces propriétés communes peuvent exister entre différents types de véhicules.

Pour organiser ces similitudes, on introduit la notion de classe. Une classe est un ensemble d'objets présentant des similitudes en termes d'identification, de comportement ou de description de l'état dans une application. Par exemple, une classe `Voiture` pourrait avoir des attributs comme la marque, le modèle, l'année, et la couleur, ainsi que des méthodes pour démarrer, arrêter, et accélérer. Un objet appartenant à une classe est appelé **instance** de cette classe.

- **Classe `Voiture`** :
  - **Attributs** : `marque`, `modèle`, `année`, `couleur`.
  - **Méthodes** : `démarrer()`, `arrêter()`, `accélérer()`.
- **Instances de la Classe `Voiture`** :
  - `voiture1` : marque=« Toyota », modèle=« Corolla », année=2020, couleur=« Rouge ».

- `voiture2` : marque=« Honda », modèle=« Civic », année=2018, couleur=« Bleue ».

Ces instances de la classe `Voiture` partagent les mêmes attributs et comportements, illustrant les similitudes entre objets et la puissance de la notion de classe en programmation orientée objet. La classe permet de créer des objets structurés de manière cohérente, facilitant ainsi leur gestion, réutilisation et maintenabilité.

7 Les classes - Utilisation

Une classe décrit les propriétés de ses instances en définissant leurs variables d'état et leurs méthodes. Les variables d'état, ou attributs, représentent les données propres à chaque instance, tandis que les méthodes définissent les comportements que les instances peuvent effectuer.

Une classe permet de créer de nouveaux objets via des constructeurs. Les constructeurs sont des méthodes spéciales qui initialisent les nouvelles instances de la classe en leur assignant des valeurs initiales pour leurs attributs.

Une classe facilite également le partage d'informations entre objets et la gestion des instances. Pour ce faire, des propriétés peuvent être associées aux classes elles-mêmes, plutôt qu'aux objets individuels. Ces propriétés de classe, également appelées attributs ou méthodes statiques, sont partagées par toutes les instances de la classe et peuvent être utilisées pour maintenir des informations communes ou pour coordonner les actions entre les instances.

8 Origines et Motivations de la Création du Langage Java

Java a été créé par James Gosling et une équipe de développeurs chez Sun Microsystems au début des années 1990. Les origines du langage remontent à 1991, lorsque Sun Microsystems a initié le projet « Green » pour développer une technologie pour les appareils électroménagers intelligents. L'équipe a rapidement constaté que les langages existants, comme C++ et C, étaient inadaptés pour les applications distribuées et portables. Le langage Java a donc été conçu pour répondre à plusieurs besoins spécifiques : offrir une portabilité maximale (écrire une fois, exécuter partout), améliorer la sécurité et la robustesse des applications, simplifier le développement en éliminant les aspects complexes de C++, et fournir des fonctionnalités intégrées pour la gestion de la mémoire. Java a été officiellement lancé en 1995, et sa capacité à fonctionner sur n'importe quel dispositif grâce à la machine virtuelle Java (JVM) a été une révolution, faisant de lui un langage incontournable pour le développement web et d'applications mobiles. En 2010, Oracle a racheté Sun Microsystems, acquérant ainsi Java et poursuivant son développement et son évolution. Java est distribué sous une licence open source pour son code source, tandis que les binaires sont disponibles sous une licence propriétaire. Cela permet

aux développeurs de contribuer au code source tout en respectant les restrictions associées aux binaires distribués par Oracle. Des distributions de binaires libres, telles que Temurin (anciennement AdoptOpenJDK), fournissent une alternative open source populaire pour les utilisateurs soucieux de la liberté d'utilisation et de distribution des logiciels Java.

9 Les caractéristiques de Java

Java est un langage de programmation orienté objet célèbre pour sa portabilité grâce à la machine virtuelle Java (JVM), permettant de « Compiler une fois, exécuter partout ». Il se distingue par sa sécurité renforcée grâce à un typage fort et des mécanismes de vérification tout au long de l'exécution. Java facilite la programmation multi-thread, essentielle pour gérer des tâches concurrentes, et offre la flexibilité de charger des classes dynamiquement pendant l'exécution. Sa bibliothèque standard complète inclut des API pour les interfaces graphiques, le réseau, les bases de données, et plus encore. Java utilise un modèle d'exécution combinant compilation en bytecode portable et interprétation par la JVM, avec des optimisations JIT pour améliorer les performances. De récentes avancées comme l'AOT (Ahead-Of-Time Compilation) et une optimisation JIT continue renforcent encore ses capacités d'exécution, consolidant Java comme un choix privilégié pour les applications critiques et distribuées à grande échelle.

10 Du code source à l'exécution

Le processus d'exécution d'un programme Java commence par la compilation du code source, écrit en fichiers `.java` à l'aide du compilateur Java (`javac`). Ce processus transforme le code source en bytecode Java, représentant un pseudo-code compréhensible par la Java Virtual Machine (JVM). La JVM, une machine virtuelle spécifique à Java, est ensuite implémentée sur différents systèmes d'exploitation tels que Windows, Unix et MacOS, assurant ainsi la portabilité des applications Java. Lors de l'exécution, la JVM interprète le bytecode Java et peut optimiser le code en utilisant des techniques JIT (Just-In-Time), fournissant des améliorations de performance au fur et à mesure de l'exécution. Oracle propose une implantation de référence de la JVM et fournit des outils pour faciliter le développement et l'optimisation des applications Java. Pour des détails techniques approfondis sur la spécification du bytecode Java, consultez la documentation Java SE.

11 Code source et bytecode

```
/**
* Cette application Java affiche simplement
* "Bonjour à tous !"
```

```
* sur la sortie standard.
*/
class BonjourATous {
  public static void main ( String [ ] args ) {
    System.out.println("Bonjour a tous !");
  }
}
```

La classe peut être compilée: `javac BonjourATous.java`

puis exécutée:`java BonjourATous`

il est aussi possible de consulter le bytecode: `javap -c BonjourATous.class`

12 Implantation et Versions

Java, développé par Oracle, est soutenu par une implantation de référence que vous pouvez explorer via la documentation Java SE. Le Java Development Kit (JDK) est essentiel pour le développement et l'exécution des applications Java, tandis que le Java Runtime Environment (JRE) permet uniquement l'exécution. Le langage Java évolue régulièrement, introduisant de nouvelles fonctionnalités tout en retirant celles devenues obsolètes, marquées comme deprecated. Pour planifier vos déploiements et migrations, référez-vous au roadmap de support Oracle qui détaille les versions à long terme (LTS) comme Java 8, Java 11, Java 17, et Java 21, chacune avec des périodes de support spécifiques allant jusqu'en 2028.

13 Environnement de Développement Java

Pour démarrer efficacement le développement Java, il est essentiel de disposer au minimum du JDK (Java Development Kit) ainsi que d'un éditeur de texte adapté. Ces outils vous permettent de compiler, exécuter et gérer vos projets Java de manière optimale. Pour une gestion avancée des dépendances et la construction de projets structurés, Maven ou Gradle sont recommandés, offrant des fonctionnalités robustes et une gestion efficace des bibliothèques externes.

En ce qui concerne l'environnement de développement intégré (IDE), plusieurs options s'offrent à vous pour bénéficier d'une assistance complète tout au long du processus de développement. Eclipse, projet open source soutenu par IBM, propose une suite complète d'outils pour le développement Java, accessible via http://www.eclipse.org. IntelliJ IDEA, développé par JetBrains, est largement plébiscité pour son éditeur performant et ses fonctionnalités avancées, disponibles en versions communautaire et commerciale sur https://www.jetbrains.com/idea. Pour une approche plus légère mais puissante, Visual Studio Code offre des extensions dédiées au

développement Java, permettant une personnalisation selon vos besoins via https://code.visualstudio.com/docs/languages/java.

14 Java en résumé

Java présente plusieurs avantages significatifs pour le développement logiciel. Tout d'abord, sa portabilité est une caractéristique majeure grâce au bytecode Java, qui peut s'exécuter sur n'importe quelle machine virtuelle Java (JVM), garantissant ainsi une compatibilité sur divers systèmes d'exploitation. De plus, Java est reconnu pour sa sécurité robuste, intégrant des mécanismes de vérification lors de l'exécution par la JVM pour prévenir les failles potentielles. En outre, Java dispose d'une vaste bibliothèque standard qui facilite le développement d'applications complexes en offrant des fonctionnalités prêtes à l'emploi pour divers domaines.

Cependant, Java est confronté à des défis en matière de performance. L'interprétation du bytecode Java ainsi que les vérifications effectuées par la JVM peuvent entraîner des ralentissements par rapport à des langages compilés directement en code machine. Bien que des techniques d'optimisation à la volée comme HotSpot soient intégrées dans les JVM pour améliorer les performances, celles-ci peuvent ne pas toujours être suffisantes pour les applications nécessitant une réactivité maximale. Pour de telles exigences, les développeurs peuvent opter pour des langages comme C ou C++, qui offrent des performances optimisées au prix d'une portabilité potentiellement réduite lors de l'intégration de code natif avec Java.