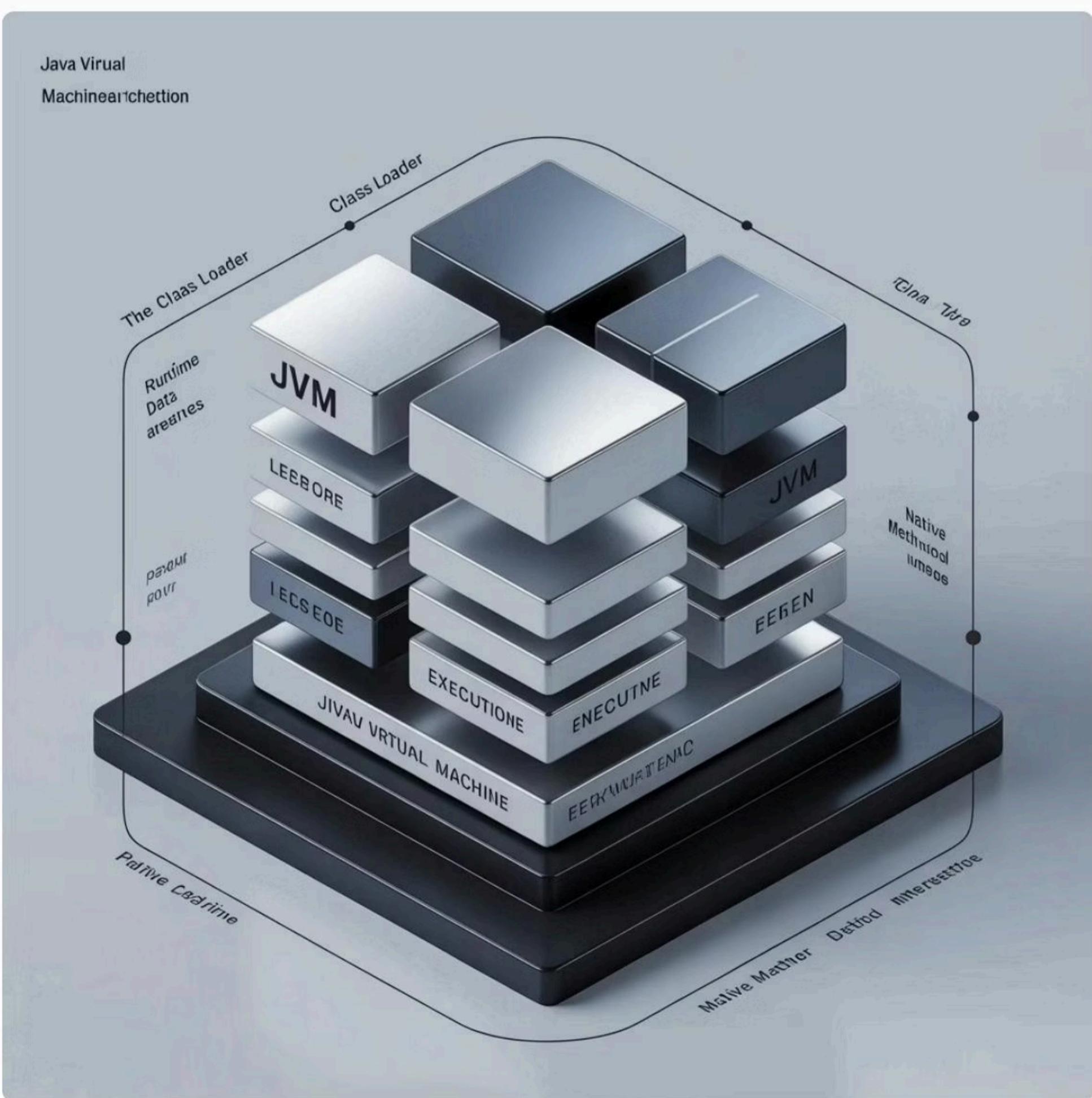
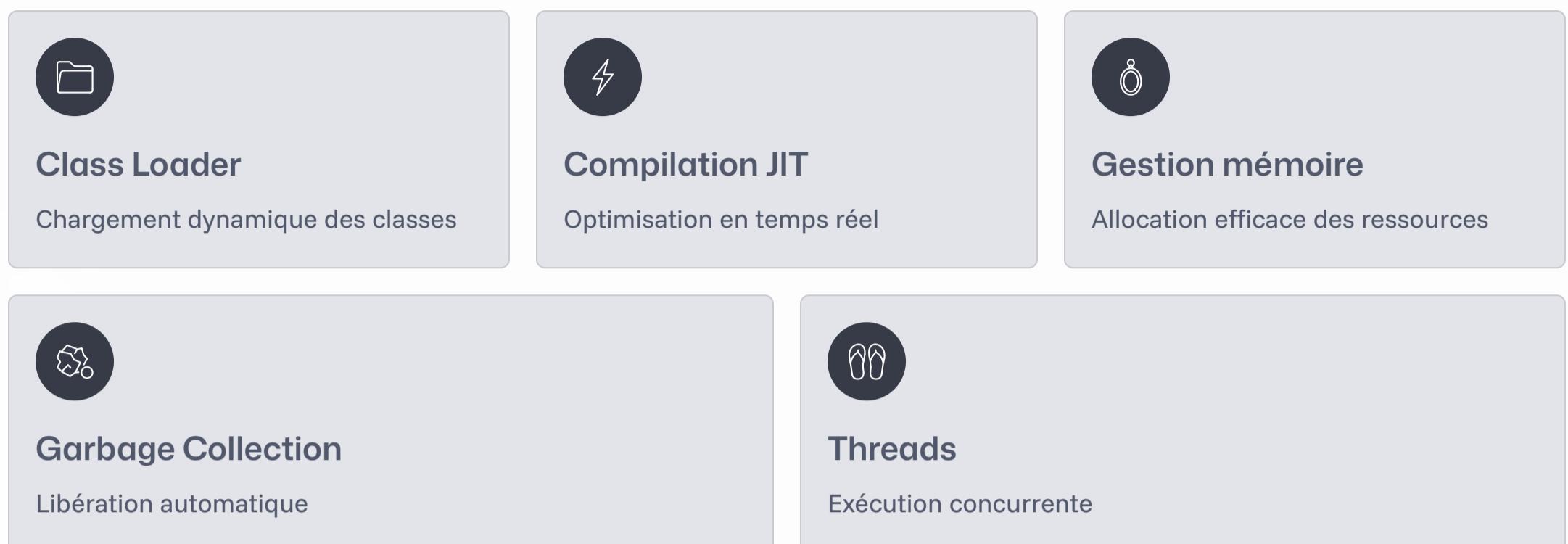


# Bonnes pratiques en Java : Compréhension du fonctionnement de la JVM

La JVM représente le cœur de l'écosystème Java. Comprendre son fonctionnement permet d'optimiser les applications et résoudre les problèmes de performance.

# Architecture globale de la JVM



# Le compilateur Just-In-Time

## Principe fondamental

Le JIT compile le bytecode en code machine natif **pendant l'exécution**, permettant des optimisations adaptatives basées sur le comportement réel.

- Observation en temps réel
- Identification des hot paths
- Optimisations agressives ciblées



Cette approche surpassé souvent les compilateurs statiques traditionnels grâce à son adaptation au contexte d'exécution.

# Fonctionnement détaillé du JIT



## Interprétation initiale

Bytecode interprété ligne par ligne pour démarrage rapide



## Profilage dynamique

Collecte de statistiques sur méthodes fréquentes



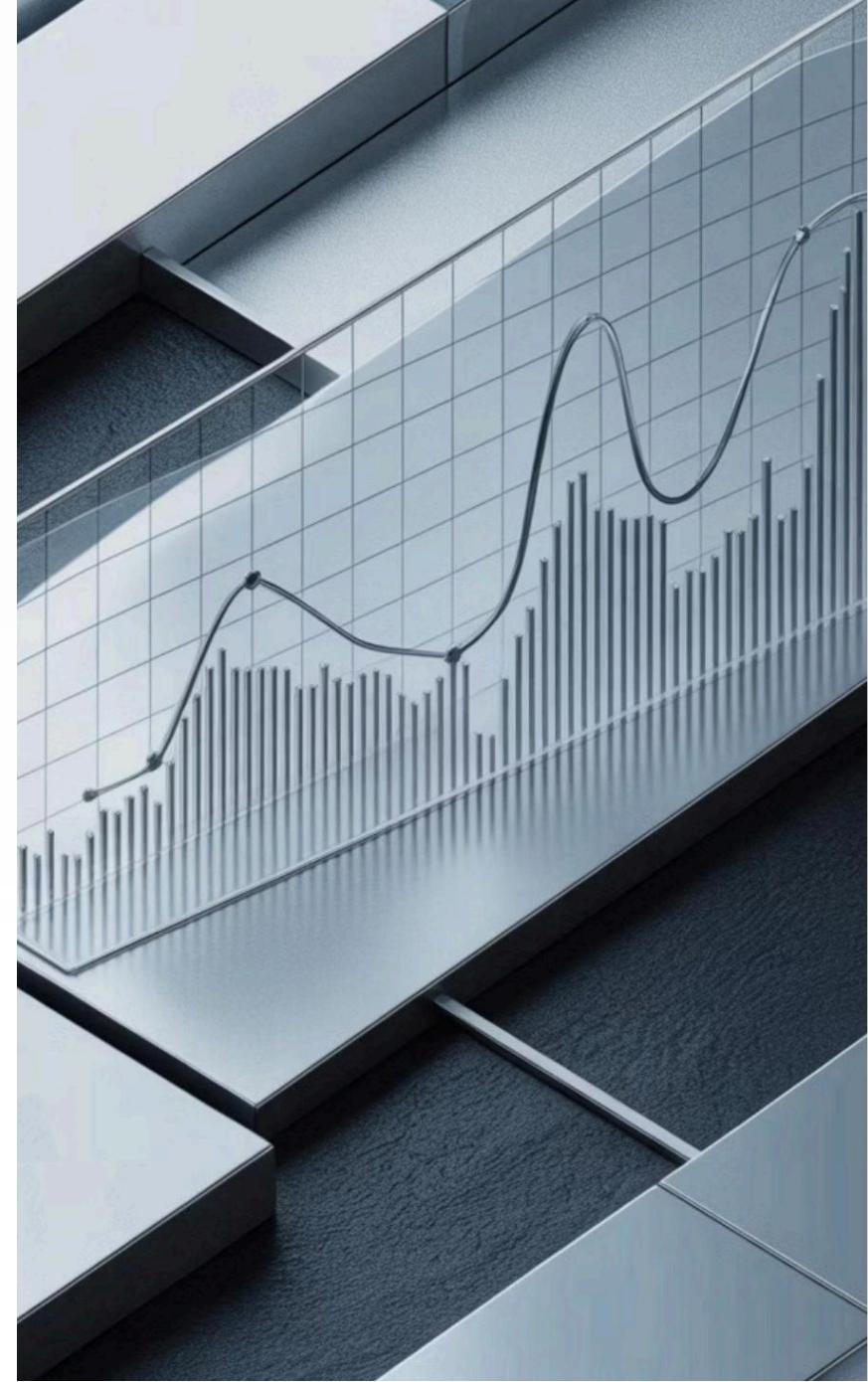
## Compilation sélective

Méthodes chaudes compilées en code natif optimisé



## Optimisation continue

Recompilation avec optimisations plus agressives



# Optimisations du compilateur JIT

1

## Inlining

Intégration du code des méthodes appelées pour éliminer le coût des appels

2

## Élimination de code mort

Suppression des instructions n'affectant jamais le résultat final

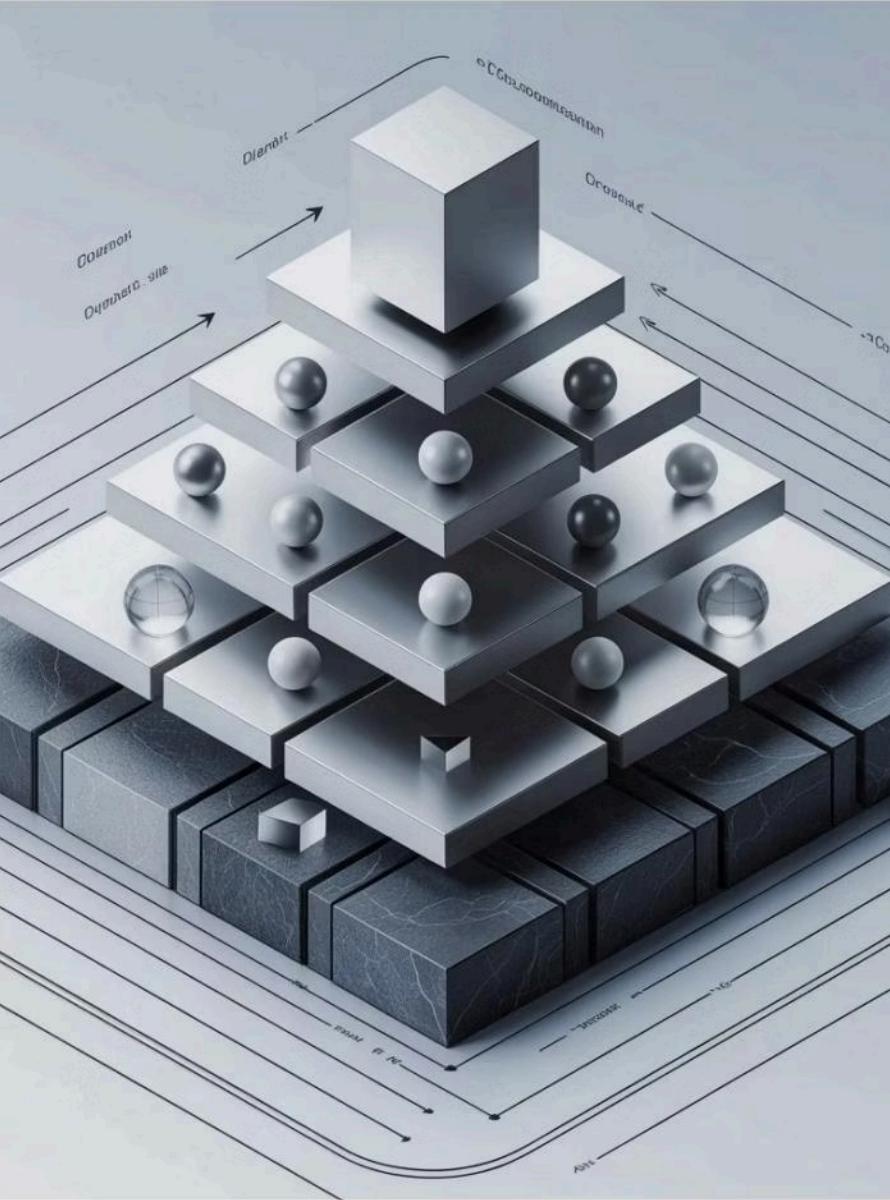
3

## Optimisations des boucles

Déroulement, vectorisation pour accélérer les itérations fréquentes

Le JIT applique des dizaines d'optimisations sophistiquées : propagation de constantes, allocation d'échappement, élimination de vérifications redondantes.

- Performance :** Ces techniques permettent d'atteindre souvent plus de 80% des performances du C++ optimisé équivalent.



# La Tiered Compilation

Évolution stratégique combinant le meilleur des deux mondes : **démarrage rapide et optimisations poussées**.

## Approche hybride

Plusieurs niveaux de compilation progressifs plutôt qu'un choix binaire entre interprétation et compilation agressive.

## Adaptation dynamique

Progression entre niveaux basée sur l'utilisation réelle du code.

# Les cinq niveaux de compilation



La progression est dynamique : une méthode peut passer rapidement du niveau 0 au niveau 3 pour le profilage, puis atteindre le niveau 4 si son utilisation intensive le justifie.

# Compilateurs C1 et C2



## Compilateur C1 (Client)

- Compilation rapide, faible latence
- Optimisations légères et conservatrices
- Idéal pour applications interactives
- Faible impact sur démarrage



## Compilateur C2 (Server)

- Compilation lente mais très optimisée
- Analyse approfondie du flux de données
- Optimisations agressives et spéculatives
- Performance maximale pour code chaud

# Avantages de la Tiered Compilation



## Démarrage accéléré

Réactivité immédiate sans attendre optimisations C2



## Profilage précis

Optimisations basées sur comportement effectif



## Performance optimale

Code fréquent bénéficie d'optimisations avancées

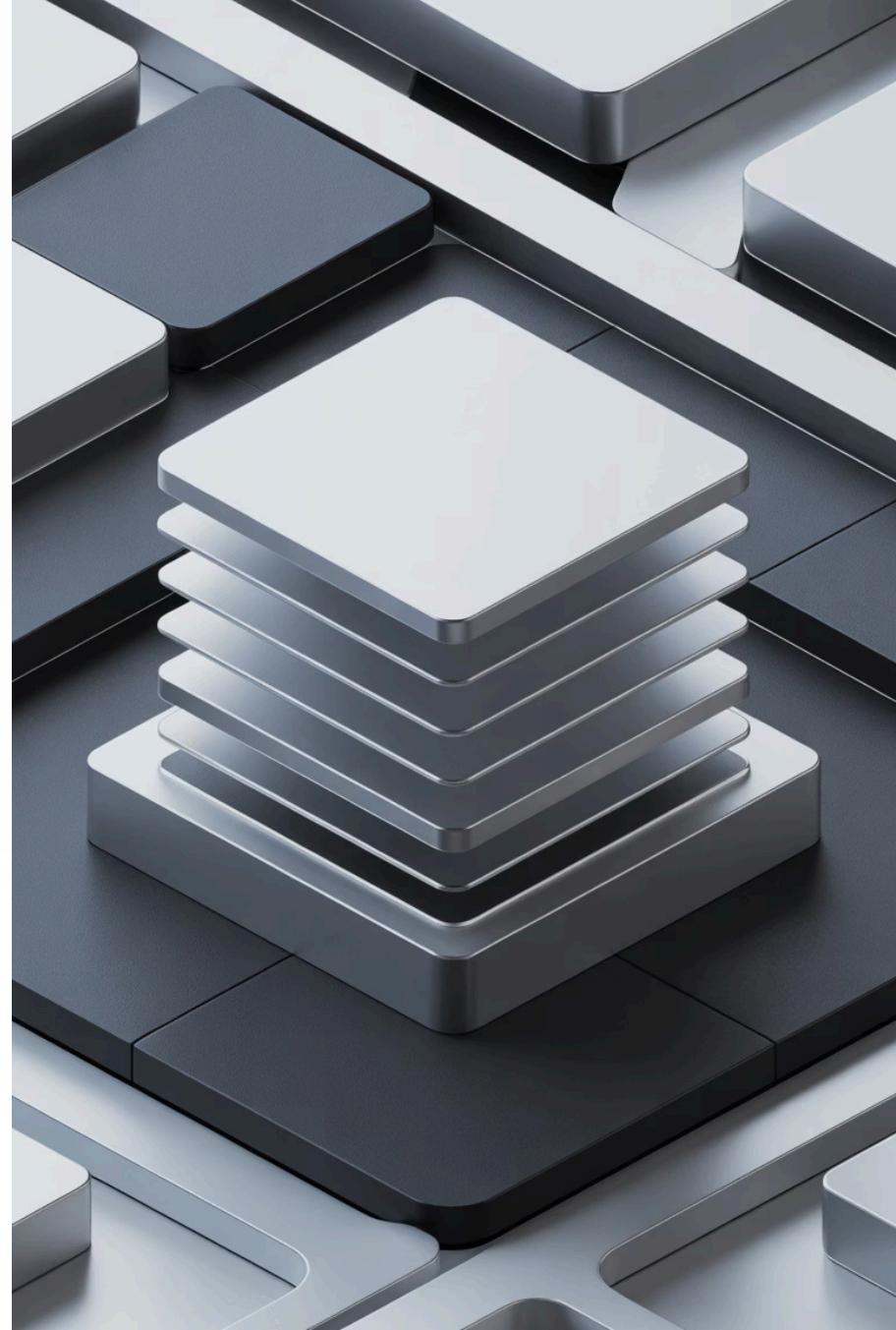


## Ressources efficaces

Investissement proportionnel à l'importance du code

# Architecture mémoire de la JVM

La gestion de la mémoire constitue un pilier fondamental. Comprendre son architecture est essentiel pour écrire des applications performantes.



# Organisation de la mémoire

## Heap (Tas)

Espace partagé pour tous les objets Java. Zone la plus grande nécessitant garbage collection.

## Stack (Pile)

Mémoire privée par thread pour exécution des méthodes. Gestion LIFO automatique.

## Metaspace

Métadonnées des classes, code compilé, structures internes de la JVM.

Cette séparation permet d'optimiser l'allocation et la libération selon les patterns d'utilisation des objets.

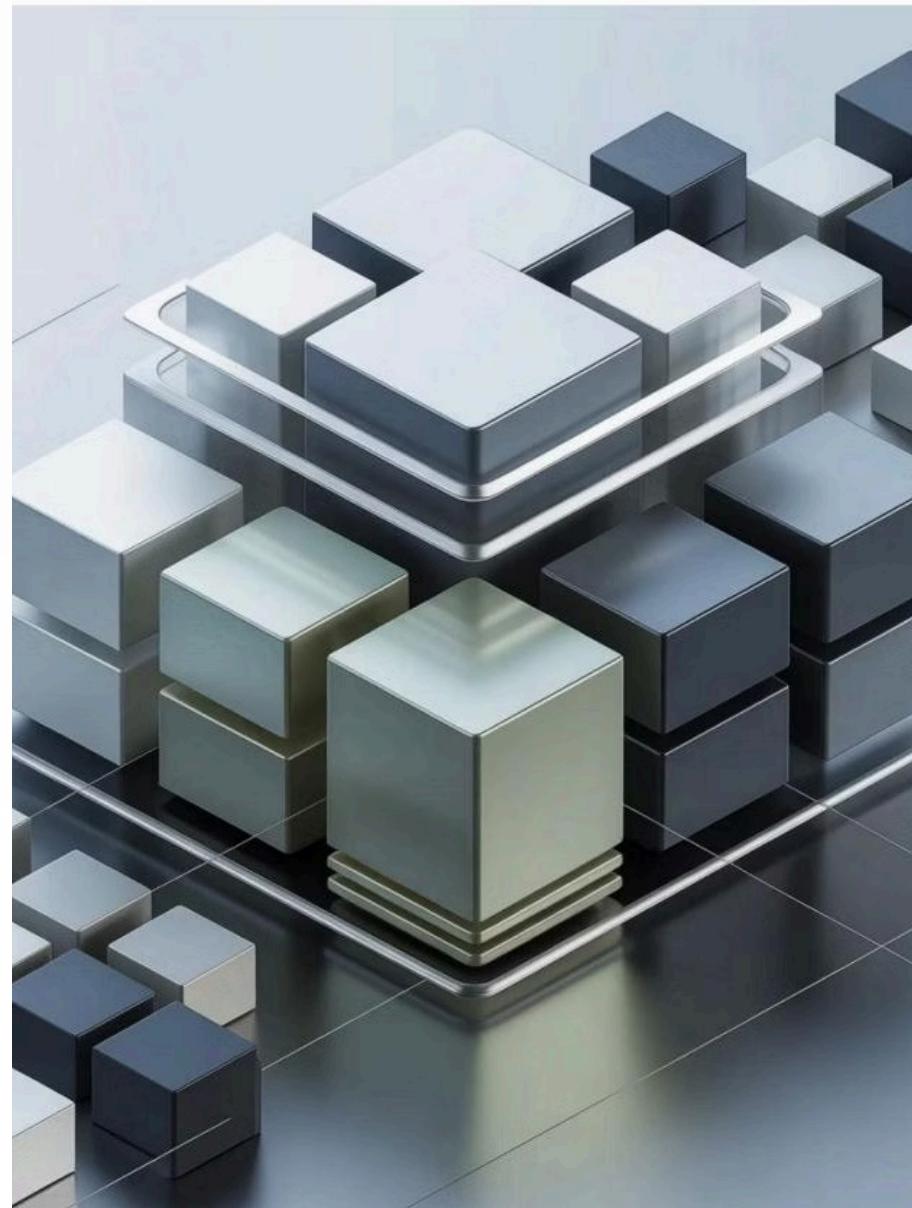
# La Heap : Espace des objets

Tous les objets créés avec `new` résident dans la heap. Zone partagée entre tous les threads nécessitant synchronisation.

Divisée en générations pour optimiser le garbage collection selon l'hypothèse générationnelle : *les jeunes objets meurent rapidement.*

## Caractéristiques

- Taille configurable
- Croissance dynamique
- Garbage collection automatique



# Structure de la Heap

1

## Young Generation

### Eden + Survivor

Allocation initiale pour nouveaux objets. Collections fréquentes et rapides.

2

## Old Generation

### Tenured

Objets ayant survécu à plusieurs collectes. Collections moins fréquentes mais coûteuses.

:n

Survivor Space



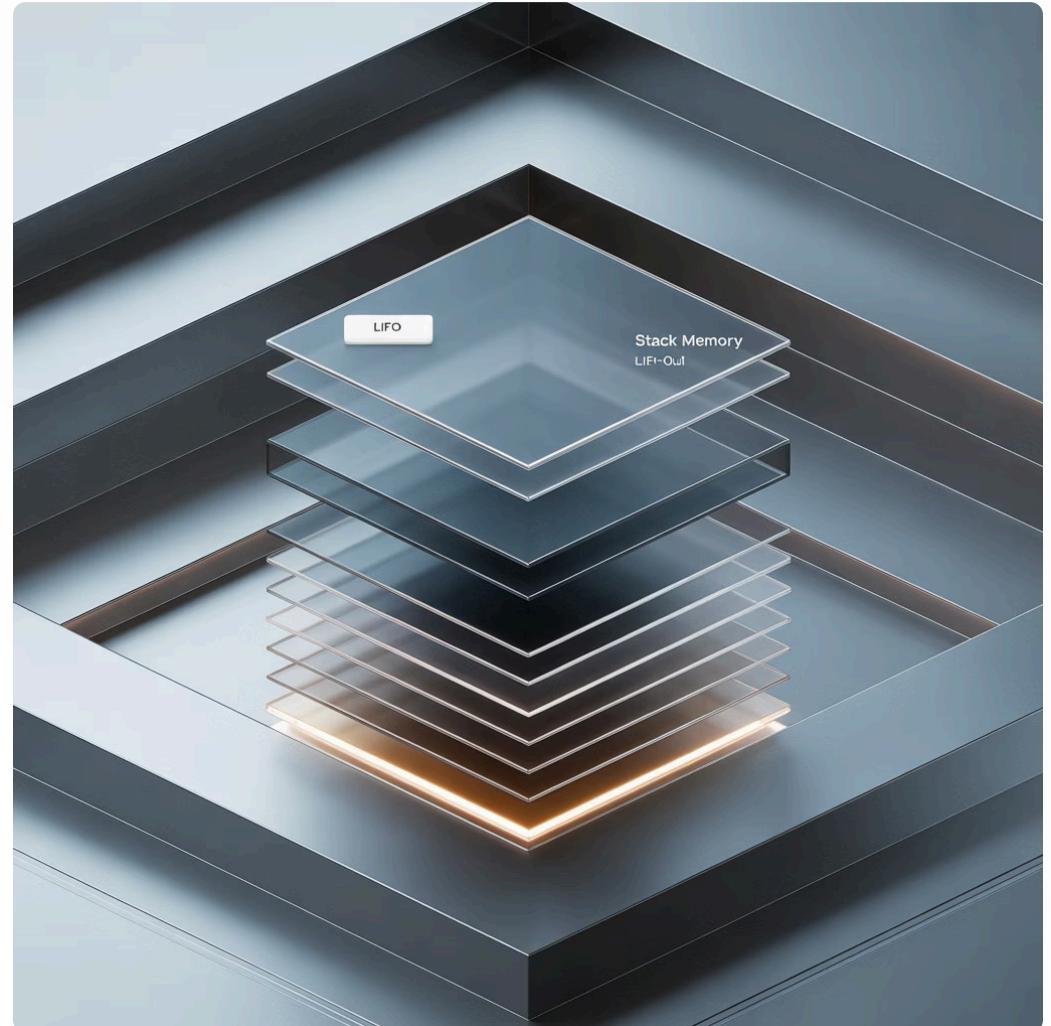
Eden :

# La Stack : Mémoire d'exécution

## Espace privé par thread

Chaque thread possède sa propre stack, éliminant les besoins de synchronisation.

- Frame créé à chaque appel de méthode
- Variables locales et paramètres
- Libération automatique LIFO
- Pas de garbage collection



- Gestion déterministe ultra-rapide, mais taille limitée (risque de StackOverflowError).

# Heap vs Stack : Comparaison

Aspect	Heap	Stack
Portée	Partagée entre threads	Privée à chaque thread
Type de données	Objets et tableaux	Variables locales primitives
Gestion mémoire	Garbage collection	Libération automatique LIFO
Taille	Grande, configurable	Limitée, fixe
Performance	Allocation plus lente	Allocation très rapide

# Exemple pratique : Allocation mémoire

```
public class Person {  
    private String name; // Référence en stack  
    private int age; // Valeur en stack  
  
    public void greet() {  
        String message = "Hello"; // Référence locale  
        System.out.println(message + name);  
    }  
}
```

Person p = new Person(); // p en stack  
// Objet Person en heap

Référence p stockée sur la stack

Objet Person réside dans la heap

Variables locales de greet() sur la stack

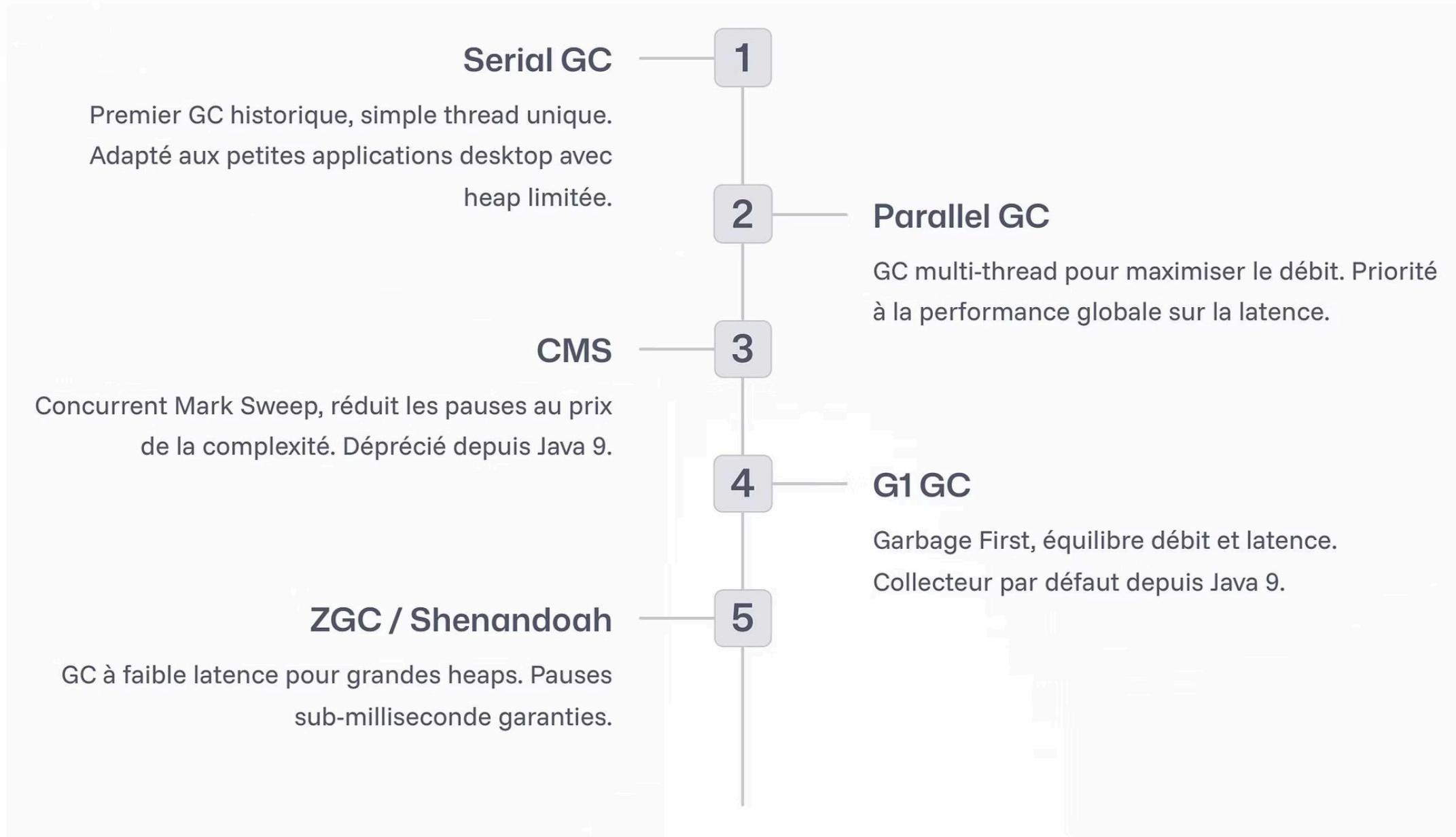


# Le Garbage Collection

Mécanisme automatique libérant les développeurs de la gestion manuelle de la mémoire.

Le GC identifie les objets inaccessibles et récupère leur mémoire via l'analyse de graphe d'accessibilité.

# Panorama des Garbage Collectors



## Serial GC

Simple thread, petites applications

## Parallel GC

Multi-threads, débit maximal

## G1 GC

Pauses prévisibles, par défaut

## Shenandoah

Faible latence, évacuation concurrente

## ZGC

Scalabilité extrême, pauses < 10ms



# G1 Garbage Collector

Garbage First partitionne la heap en régions de taille égale (1-32 MB).  
Collecte prioritairement les régions avec le plus d'objets morts.

## Flexibilité

Affectations de générations changent dynamiquement

## Prédicatif

Respecte objectifs de temps de pause configurables

## Incrémental

Maximise mémoire récupérée par pause

# G1 GC : Phases de collection



## Young Collection

Collection rapide Eden et Survivor.  
Evacuation vers Survivor ou Old.



## Concurrent Marking

Identification concurrente des objets  
vivants en parallèle.



## Mixed Collection

Combine Young et Old regions. Priorise  
régions avec plus de garbage.

G1 ajuste dynamiquement le nombre de régions collectées pour respecter `-XX:MaxGCPauseMillis`.

# Shenandoah GC : Faible latence

## Approche révolutionnaire

Conçu par Red Hat pour minimiser les temps de pause. Travail de collection effectué de manière concurrente avec les threads applicatifs.

- Forwarding pointers sophistiqués
- Barrières mémoire pour concurrence
- Evacuation concurrente des objets



- ❑ **Pauses sub-millisecondes** même pour heaps de plusieurs centaines de GB. Compromis : 5-10% de perte de débit vs G1.

# Caractéristiques de Shenandoah



## Evacuation concurrente

Objets déplacés pendant exécution de l'application



## Pauses ultra-courtes

1-10 ms indépendamment de la taille heap



## Support grandes heaps

Performances constantes même avec plusieurs centaines de GB



## Adaptation dynamique

Ajustement selon pression mémoire et patterns d'allocation



# ZGC : Scalabilité extrême

Développé par Oracle, ZGC vise des pauses **< 10 millisecondes** même pour heaps de plusieurs téraoctets.

## Colored Pointers

Métadonnées encodées dans les bits inutilisés des pointeurs 64-bit

## Load Barriers

Tracking de l'état des objets directement dans les références

## Régions dynamiques

Taille variable (multiples de 2 MB) pour flexibilité exceptionnelle

# Quand utiliser quel GC ?



## G1 GC - Par défaut

- Applications généralistes
- Heaps 4-64 GB
- Bon équilibre débit/latence
- Pauses acceptables (50-200 ms)



## Shenandoah - Faible latence

- Applications temps réel
- Trading, gaming, télécoms
- Exigence pauses < 10 ms
- Disponible dans OpenJDK



## ZGC - Scalabilité extrême

- Heaps massives (> 100 GB)
- Microservices cloud
- Pauses ultra-courtes garanties
- Nécessite Java 15+

40

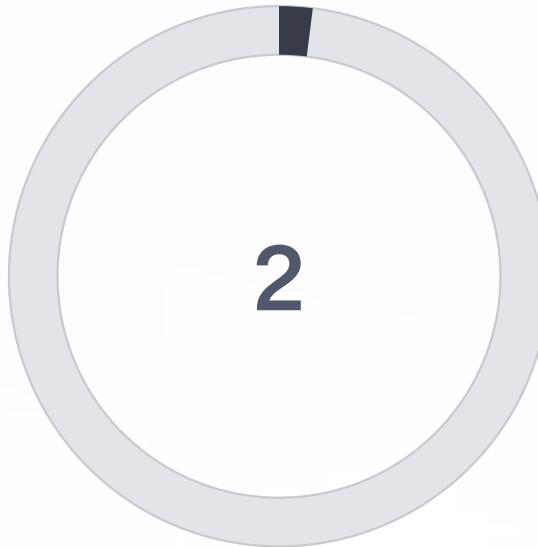
■ Shenandoah

# Synthèse : Maîtriser la JVM



## Niveaux de compilation

Tiered compilation optimise progressivement le code chaud



## Zones mémoire principales

Heap pour objets, Stack pour exécution des méthodes



## GC nouvelle génération

G1, Shenandoah et ZGC offrent pauses sub-millisecondes

---

La compréhension approfondie de la JVM transforme les développeurs en experts capables d'optimiser pour des performances exceptionnelles. **Mesurez, profilez, expérimitez** : la performance se mesure et s'optimise de manière méthodique.