

# Java Virtual Machine et Garbage Collectors

*Introduction aux GC et à la JVM*

# Introduction

Les machines virtuelles sont un élément indispensable de tous les nouveaux langages.

Offre la portabilité, les optimisations, la gestion de la mémoire.

# Introduction

Java est une plateforme et un langage.

La JVM est une “spécification”

Les implémentations varient selon les vendeurs  
(IBM, Oracle, Azul, ... )

# Introduction

JVM transforme le bytecode en langage machine

Ce langage intermédiaire permet la portabilité totale.

Java est-il un langage interprété ou compilé ?

# JVM

Différents modes de lancement

Effectue toutes les optimisations d'un processus (inlining, ... )

Énormément de paramètres de personnalisation

# javac

javac compile les .java en .class

Qui optimise le code ?

# JIT

Compilateur Just-In-Time.

La compilation du code se fait au besoin afin d'optimiser l'application

Peut se paramétrer de plein de manières !

# javac ne fait rien !

```
public final class NoOptimisation {  
    private final static void empty() {}  
    private final static void dead() {  
        String bar = "Bar";  
    }  
    public static void main(String... args) throws Exception {  
        for(int i=0; i<10_000_000; i++)  
            empty();  
        System.out.println("Hello world");  
    }  
}
```



# javac ne fait rien !

```
private static void empty();
```

Code:

```
0: return
```

```
private static void dead();
```

Code:

```
0: ldc      #2 // String Bar
```

```
2: astore_0
```

```
3: return
```

```
public static void main(java.lang.String...);
```

Code:

```
3: ldc      #6 // int 10000000
```

```
5: if_icmpge 17
```

```
8: invokestatic #4 // Method empty:()V
```

# Exemple : Code de base

```
public class Component {  
    private Point location;  
    public Point getLocation() { return new Point(location); }
```

```
    public double getDistanceFrom(Component other) {  
        Point otherLocation = other.getLocation();  
        int deltaX = otherLocation.getX() - location.getX();  
        int deltaY = otherLocation.getY() - location.getY();  
        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);  
    }  
}
```

```
public class Point {  
    private int x, y;  
    // constructors / getters / setters  
}
```

# Example : Inlining

```
public class Component {  
    private Point location;  
    public Point getLocation() { return new Point(location); }  
  
    public double getDistanceFrom(Component other) {  
        Point otherLocation = new Point(other.x, other.y);  
        int deltaX = otherLocation.x - location.x;  
        int deltaY = otherLocation.y - location.y;  
        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);  
    }  
}
```

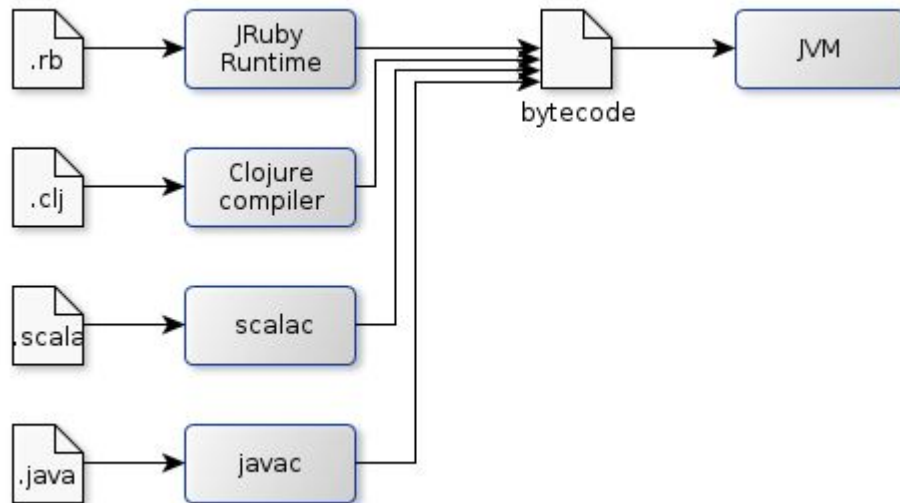
# Example : Escape analysis

```
public class Component {  
    private Point location;  
    public Point getLocation() { return new Point(location); }  
  
    public double getDistanceFrom(Component other) {  
        int tempX = other.x, tempY = other.y;  
        int deltaX = tempX - location.x;  
        int deltaY = tempY - location.y;  
        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);  
    }  
}
```

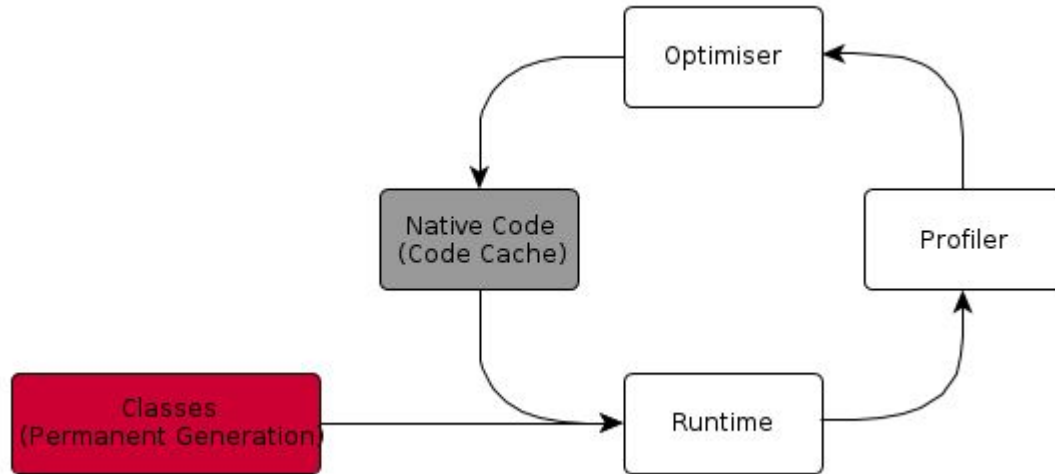
# GCC vs JIT

Un compilateur comme GCC effectue les mêmes optimisations. Alors qu'apporte le compilateur JIT ?

# JVM et compilation



# JVM et compilation



# JVM et mémoire





# Ramasse-miettes

# Introduction

Libère la mémoire automatiquement

Facilite la vie du développeur

Présent dans presque tous les langages de programmation depuis le C++

# Pile et tas

Tout processus a un espace d'adressage propre.

Tout processus est constitué d'un thread ayant une pile propre

Tous les threads d'un processus partagent le tas

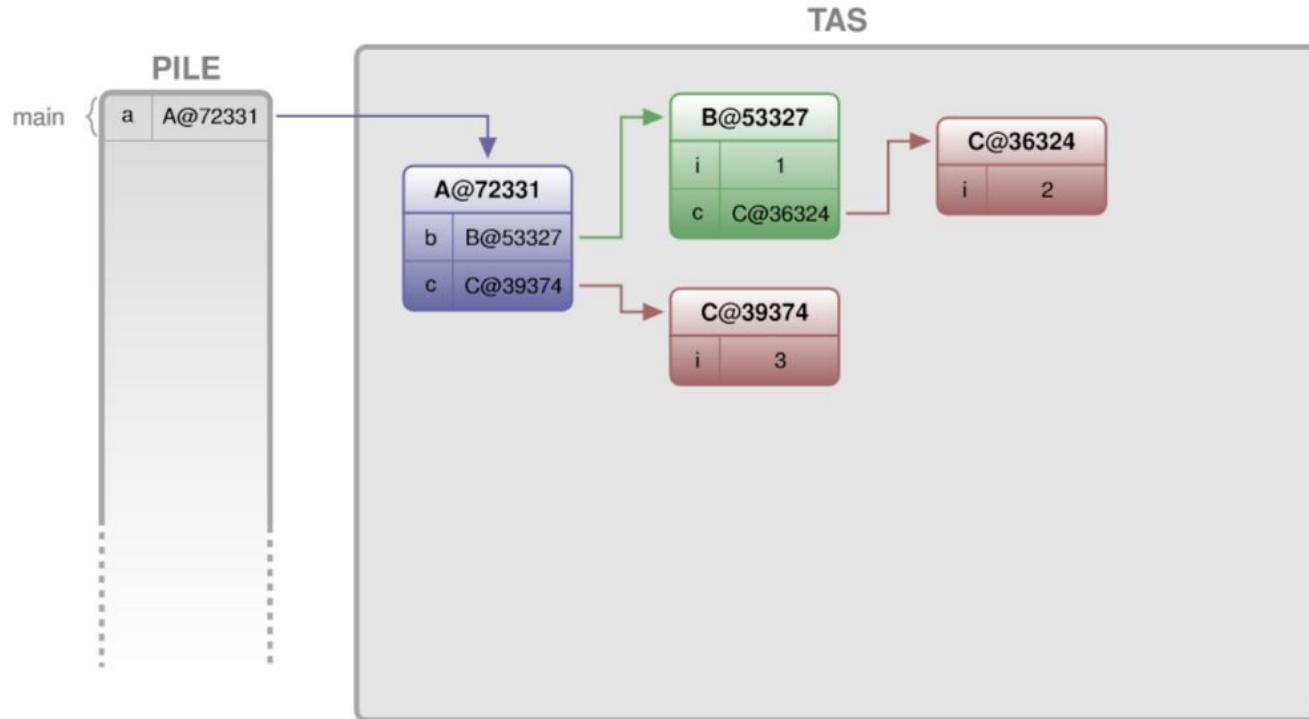
# Exemple : Pile et tas

```
class A {  
    B b;  
    C c;  
    public A(){  
        b = new B(1,2);  
        c = new C(3);  
    }  
}
```

```
class B {  
    int i;  
    C c;  
    B(int x, int y){  
        i = x;  
        c = new C(y);  
    }  
}
```

```
class C {  
    int i;  
    public C(int y){  
        i = y;  
    }  
}
```

# Exemple : Pile et tas



# Types de GC

Générationnel : Séparation de la mémoire

Concurrent : S'exécute en parallèle de l'application

Parallèle : Peut utiliser plusieurs threads

Incrémental : Nettoie une partie des zones mémoire et non la totalité

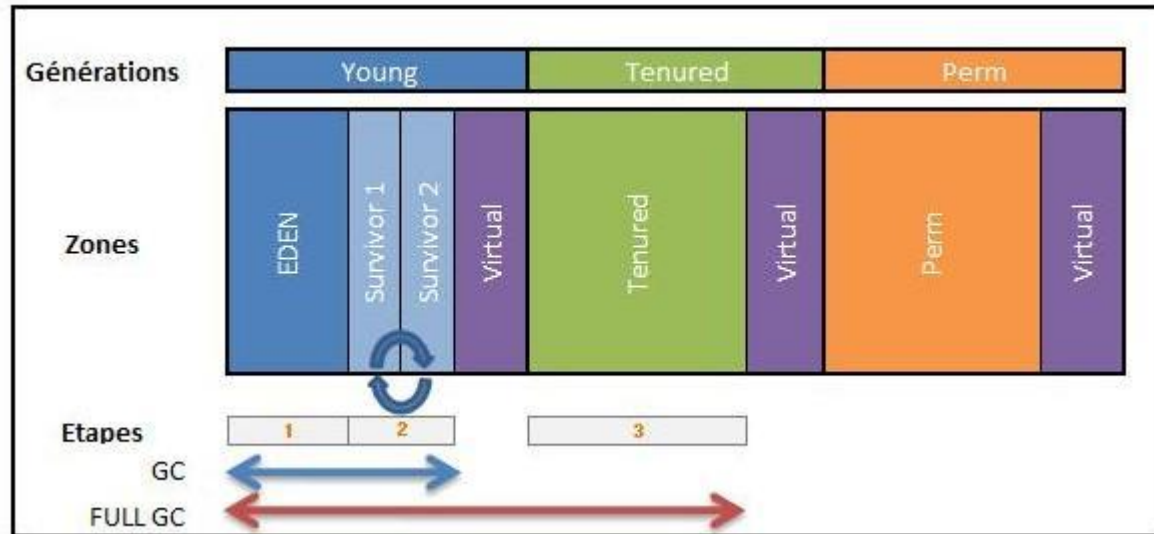
# L'hypothèse générationnelle

“La plupart des objets meurent jeunes”

Solution : Scinder la mémoire en plusieurs espaces

Intérêt : Ne plus parcourir le tas en entier

# L'hypothèse générationnelle





# Stop-the-world

Propre à (presque) tous les GC !

1. Arrêt des threads applicatifs (mutators)

1. Lancement des GC threads et passage du GC

1. Réveil des threads applicatifs

# Les collections

Une collection correspond au passage du GC.

Young Collection → Eden et Survivor (Young)

Full GC → Young + Old Collection

# Stop-the-world ... ou pas

Les GC non-concurrents sont stop-the world

Les GC concurrents ne le sont pas  
complètement

# Algorithmes de collections

Mark & Sweep & Compact → On marque les objets en vie, suppression des objets morts, compactage de la mémoire.

Mark & Evacuate → On marque les objets en vie et on les copie dans un autre espace.

# Les GC existants

Parallel Scavenge

CMS

G1

# Parallel Scavenge

Générationnel, parallèle, copiant et compactant.

Mark & Evacuate pour la zone young

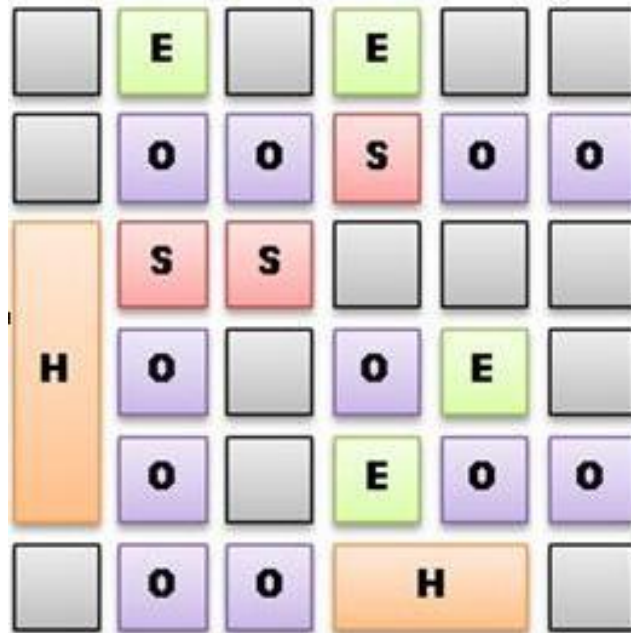
Mark & Sweep pour la zone old

# G1

Générationnel, parallèle, concurrent,  
incrémental, soft real-time !

Prend en compte l'avènement du  
multi-processeurs

# G1





# G1

Comment déterminer le temps que va prendre le nettoyage de zones ?

# G1

Comment déterminer le temps que va prendre le nettoyage de zones ?

Grâce aux statistiques. Cela permet de rendre de plus en plus précis le temps que peut prendre une charge de travail.

# G1 : La complexité

Lors d'un déplacement d'objet, sa référence devient obsolète.

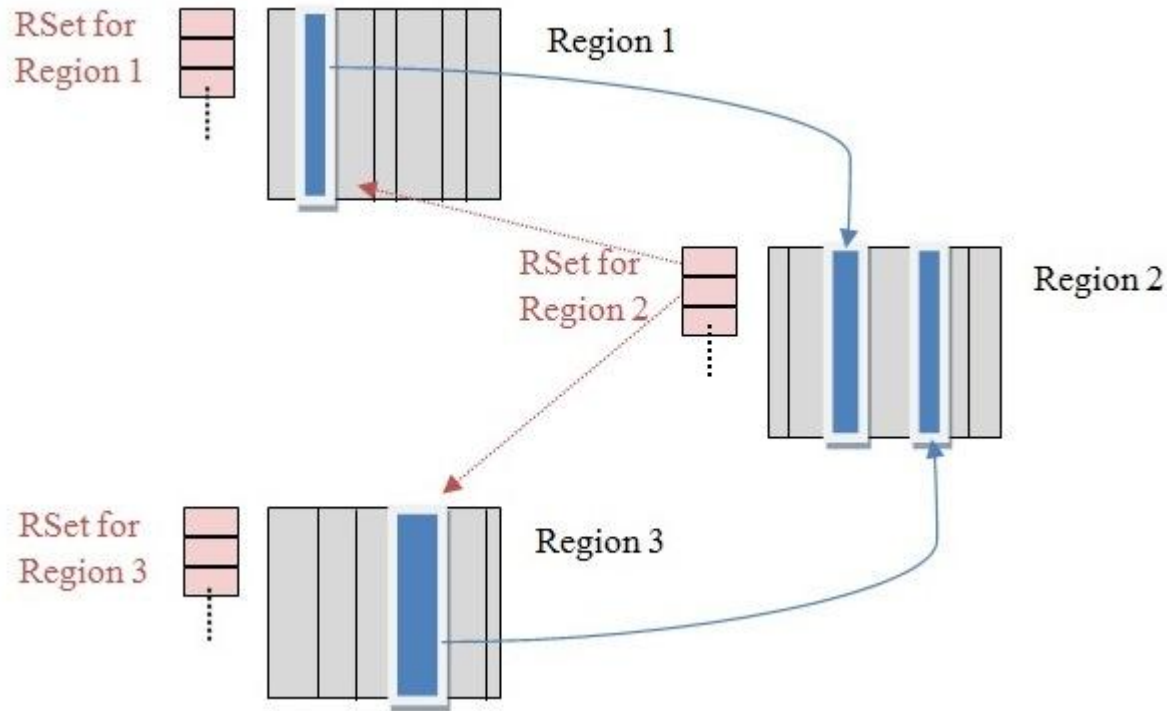
Que faire contre cela ?

# G1 : Remember Sets

Mettre à jour les références lors d'un déplacement d'objet.

Cela implique de connaître tous les objets ayant une référence vers l'objet à déplacer !

# G1 : Remember Sets



# G1 : Remember Sets

Un RS contient les références vers les objets ayant une référence vers des objets de la région courante ... !

Si la région possède des objets très référencés, que faire ?

# G1 : Remember Sets

Utiliser différents niveaux de granularité.

Ce GC étant concurrent, que faire lors de la mise à jour des références puisque l'application tourne toujours ?

# G1 : Concurrency

Écrire dans un buffer la mise à jour avec un verrou

Des threads mettent à jour les références  
(Concurrent Refinement Thread)



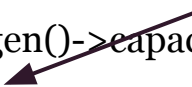
# Quiz

# Trouver un bug

```
// Calcul le nombre de threads GC selon la taille de la heap lors d'une collection  
// Le code a été simplifié pour la lisibilité
```

```
active_workers_by_heap_size = MAX(2, Universe::heap()->capacity() / 32);
```

```
size_t ParallelScavengeHeap::capacity() const {  
    return young_gen()->capacity_in_bytes() + old_gen()->capacity_in_bytes();  
}
```



# System.gc()

Pourquoi faut-il éviter d'utiliser System.gc() ?

# System.gc()

Pourquoi faut-il éviter d'utiliser System.gc() ?

*Aucune certitude du lancement du GC. De plus, c'est forcément une Full collection.*

# Algorithme

Combien peut-il y avoir d'algorithmes de GC tournant dans un processus (mais pas en parallèle) ?

# Algorithme

Combien peut-il y avoir d'algorithmes de GC tournant dans un processus (mais pas en parallèle) ?

*Plusieurs (au moins 2)*

# Processeurs

Combien lançons-nous de threads pour un serveur web sur un 8 coeurs ?

# Processeurs

Combien lançons-nous de threads pour un serveur web sur un 8 coeurs ?

*8 ... Bien que ça dépendent du type d'opérations (IO ...)*



# Processeurs

Combien lançons-nous de threads pour un serveur web sur un 16 coeurs ?

# Processeurs

Combien lançons-nous de threads pour un serveur web sur un 16 coeurs ?

*Moins de 16. Cela varie selon les CPU. La JVM calcule comme suit :*

$$8 + (NB\_CPU - 8) * 5 / 8$$

# malloc ou new ?

Qu'est-ce qui est le plus rapide entre un malloc et un new (de taille équivalente) ?

# malloc ou new ?

Qu'est-ce qui est le plus rapide entre un malloc et un new (de taille équivalente) ?

*new. Il est entre 6 et 10 fois plus rapide*

# free ou GC copiant

Qu'est ce qui est le plus rapide entre un free et un algorithme copiant pour la suppression d'objets ?

# free ou GC copiant

Qu'est ce qui est le plus rapide entre un free et un algorithme copiant pour la suppression d'objets ?

*Le coût de suppression est de  $O$  pour le GC copiant*

# La pile

Peut-on allouer dans la pile ?

# La pile

Peut-on allouer dans la pile ?

*La JVM le fait pour nous de manière intelligente*



# La pile

Quel est l'intérêt d'allouer dans la pile ?

# La pile

Quel est l'intérêt d'allouer dans la pile ?

*Car la désallocation ne coûte rien !*

# Zing

Le meilleur GC actuel est un GC Java : Azul  
Zing

Basé sur Hotspot, ultra-optimisé, n'est jamais  
stop-the-world

# Quelques statistiques

G1 → Exécution de SPECJBB 2005 (benchmark payant) pendant 6min

Le temps de passage du GC est de 1,6% sans stop-the-world !

L'application avait une heap de ...

# Conclusion

La JVM fait plus qu'un développeur (escape analysis, ... )

Les GC peuvent afficher des performances excellentes.

Néanmoins, il faut savoir les tuner ...