



CodeGen

Présenté par Yann Caron
skyguide

ENSG Géomatique

Plan du cours

Analyse sémantique

Optimisations

Code Generation

Garbage collector



Analyse sémantique



Analyse sémantique

- ✓ Définition :
 - ✓ Analyse du sens des éléments du programme
- ✓ Plusieurs opérations :
 - ✓ Contrôle des types (type checking)
 - ✓ Optimisation locales
- ✓ Important : Avant l'exécution du programme

Algorithmes

- ✓ La plupart des algorithmes d'analyse sémantique peuvent être exprimés sous la forme d'un parcours en profondeur de l'AST
- ✓ Pré : traitement du nœud n de l'AST
- ✓ Récursion : traitement des enfants du nœud
- ✓ Post : fin du traitement du nœud
- ✓ Important : L'optimisation peut s'effectuer par application successive du même algorithme (algorithme récurrent)

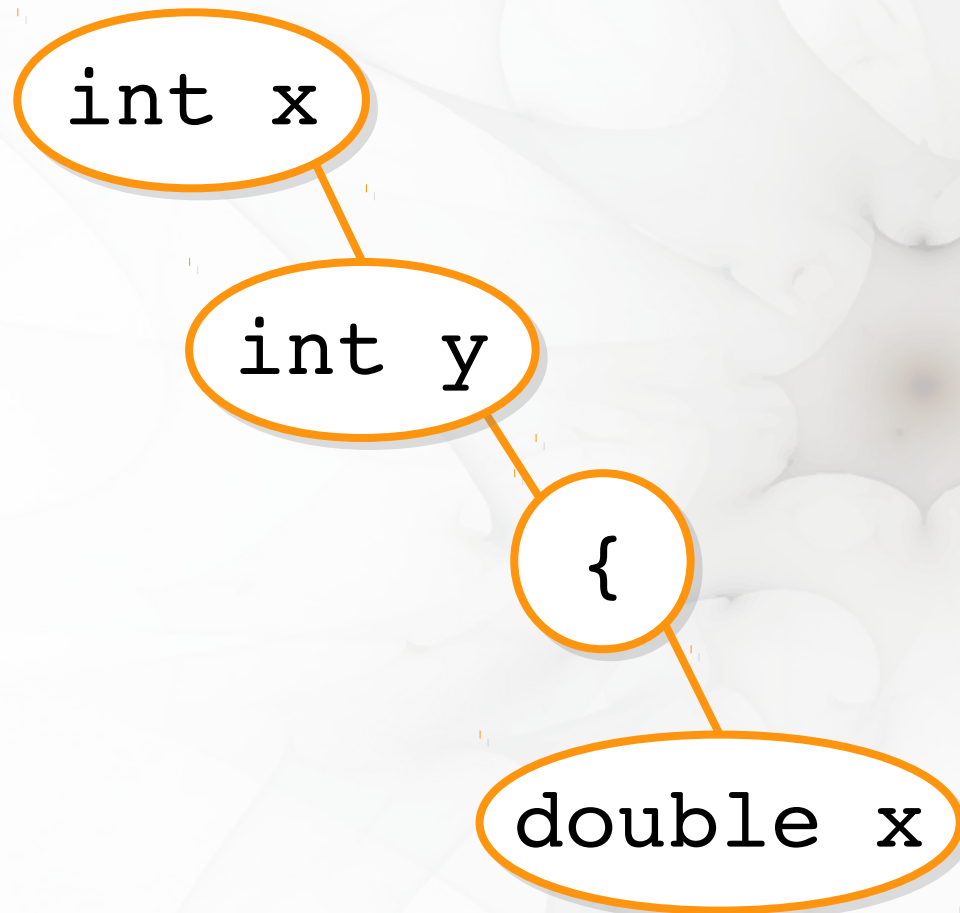
Type Checking

- ✓ Définition d'un type de variable :
 - ✓ Ensemble des valeurs possible d'une variable
ex : $[0-9]^+$
 - ✓ Ensemble des opérations possibles sur ces valeurs
ex : $+, -, /, *, \%$
- ✓ Les Classes sont une notion moderne des types
ex : Surcharge des opérateurs C++

Table des symboles

- ✓ Afin d'effectuer l'analyse des type une structure de donnée spéciale est nécessaire
- ✓ La table des symboles
- ✓ Trois opération :
 - ✓ addSymbol : Ajoute le symboles et les informations tel que le type
 - ✓ findSymbol : Recherche le symbole depuis le haut de la pile vers le bas (NULL si non trouvé)
 - ✓ removeSymbol : Supprime l'élément du haut de la pile

Stack machine



Symbol Table

Recherche ↓	
x (double)	← Masquage
y (int)	
x (int)	

Table des symboles

- ✓ Cet structure de donnée ne permet pas de gérer les contextes différents (appel de fonctions, de méthodes, objets etc..)
- ✓ Attention, on ne peut pas compter sur les stack frames (l'analyse intervient avant l'exécution)

Table des symboles

- ✓ Solution :
- ✓ Rajouter 3 méthodes :
 - ✓ enterScope : Démarre un nouveau contexte imbriqué
 - ✓ checkScope(x) : vérifie que x soit défini dans le contexte courant
 - ✓ exitScope : Sort du contexte courant

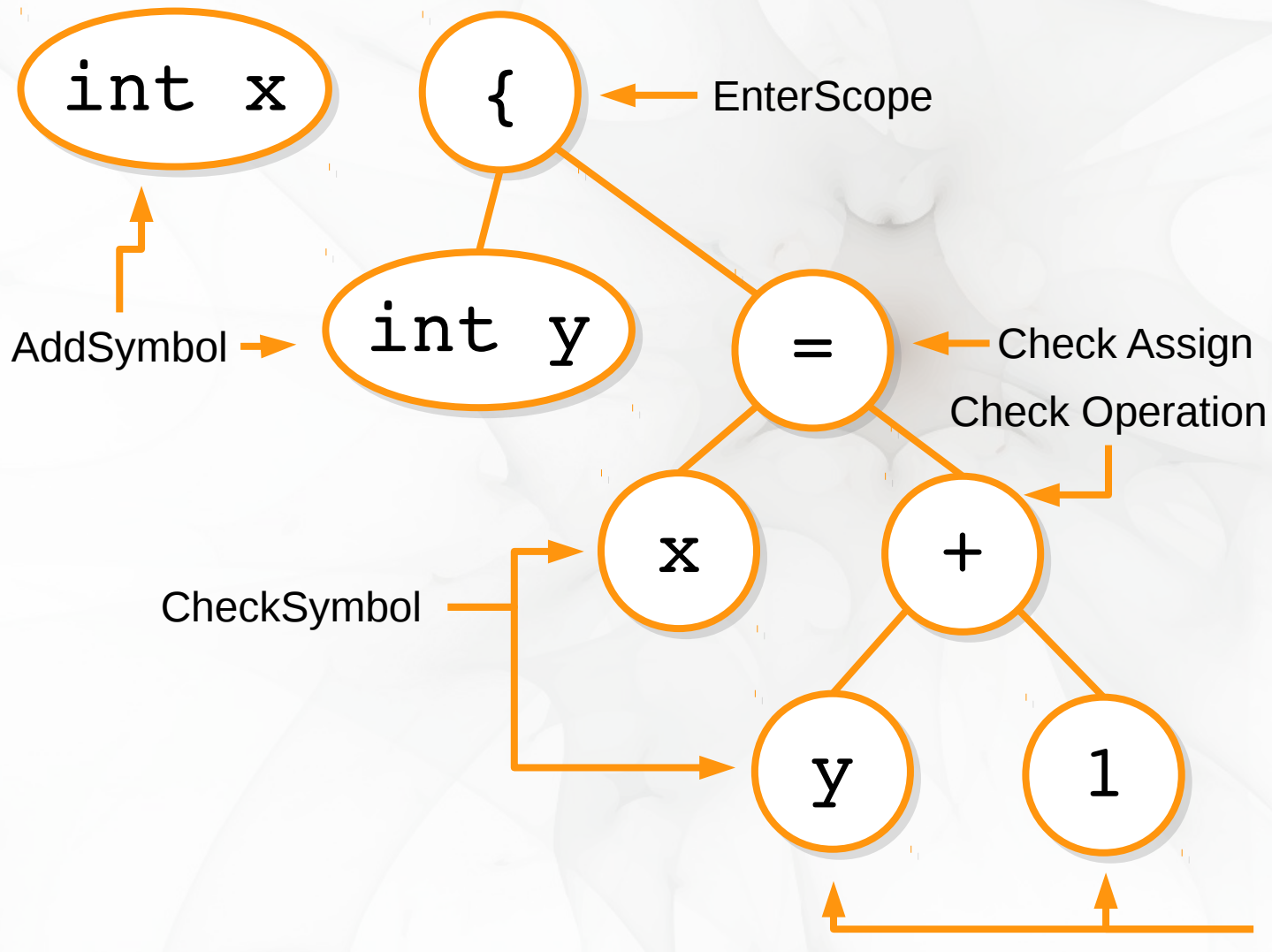
Type checking

- ✓ Analyse la sémantique des types
- ✓ Vérifie que le programme réponde aux règles fixés par le typage du langage :
 - ✓ Que la valeur respecte le type / class
 - ✓ Que les opérations soient permise
 - ✓ Que la conversion soit permise
 - ✓ Évalue le masquage

Type checking

- ✓ Analyse statique (basé sur la syntaxe uniquement et non sur le contexte d'exécution)
- ✓ Moins flexible, mais plus sûre
- ✓ Algorithme basé sur un parcours en profondeur de l'AST
- ✓ Entretiens la table des symboles

Type checking



Global scope

Nested scope 1

y (int)

x (int)

Optimisations



Optimisations

- ✓ But :
- ✓ Rendre plus performant le programme
 - ✓ En terme d'utilisation CPU
 - ✓ Exemple : simplifier les calculs
 - ✓ En terme de recherche des symboles
 - ✓ Exemple : remplacer les variables
 - ✓ En terme d'utilisation de la mémoire

Optimisations

- ✓ Peu être effectué à plusieurs niveaux :
- ✓ Sur le code assembleur
 - ✓ Expose les opportunités d'optimisation
 - ✓ Mais est dépendant de la machine
 - ✓ Et doit être réécrit en cas de changement de machine cible
- ✓ AST
 - ✓ Indépendant de la machine
 - ✓ Mais de trop haut niveau

Optimisations

- ✓ Langage intermédiaire
 - ✓ Expose les opportunités d'optimisations
 - ✓ Et est indépendant de la machine
 - ✓ Meilleur compromis
- ✓ Notion de bloque
 - ✓ Une suite de instruction sans label (sauf 1ere ligne) ni renvoie (sauf dernière ligne)
 - ✓ Rien qui pourrait en modifier le flot d'exécution)

Optimisation Assembleur

- ✓ Peephole
- ✓ Principe : identifier des portions de codes remplaçables par une code plus efficace
- ✓ La portion doit être un bloque (ni label, ni renvoie)
- ✓ Exemple :
 - ✓ `addui $a $b 0` → `move $a $b`
 - ✓ `move $a $a` peut être supprimé
 - ✓ donc `addui $a $a 0` peut être supprimé

Optimisations Locales

- ✓ S'effectue sur l'AST ou l'IL
- ✓ Simplification algébriques
- ✓ Suppression des instructions inutiles

- ✓ $x := x + 0$

- ✓ $x := x * 1$

- ✓ Simplification →

- ✓ $x := x * 0$ → $x := 0$

- ✓ $x := x ** 2$ → $x := x * x$

Optimisations Locales

- ✓ Les opérations sur des constantes peuvent être résolues lors de la compilation
 - ✓ Une opération de type $x := y \text{ op } z$
 - ✓ Si y et z sont des constantes
 - ✓ Alors l'opération peut être résolue lors de la compilation
-
- ✓ Exemple :
 - ✓ $x := 2 + 2 \quad \longrightarrow \quad x := 4$

Optimisations Locales

- ✓ Les conditions qui sont toujours fausses peuvent être supprimés
- ✓ Exemple : if $2 < 0$ jump 0
- ✓ Les conditions toujours vrais peuvent être remplacé
- ✓ Exemple : if $2 > 0$ then jump 0 → jump0
- ✓ Règle générale : Supprimer les blocs qui ne peuvent être atteints

Optimisations Locales

- ✓ Substituer les opérations

- ✓ Exemple :

$x := y + z$

... (sans modification de x , y ou z)

$w := y + z$  $w := x$

- ✓ Plus généralement : élimination des expressions communes

Optimisations Locales

- ✓ Propagation des copies

- ✓ Exemple :

$b := y + z$

$a := b$

$x := 2 * a$



$x := 2 * b$

- ✓ De plus, si a n'est pas utilisé autre part l'expression $a := b$ pourra être supprimée

Example

a := 5
x := 2 * a
y := x + 6
t := x * y



a := 5
x := 10
y := 16
t := x * y



t := 160

Optimisations Locales

- ✓ Les conditions qui sont toujours fausses peuvent être supprimés
- ✓ Exemple : if $2 < 0$ jump 0
- ✓ Les conditions toujours vrais peuvent être remplacé
- ✓ Exemple : if $2 > 0$ then jump 0 → jump0
- ✓ Règle générale : Supprimer les blocs qui ne peuvent être atteints

Optimisations Globales

- ✓ Analyse du flot d'exécution
 - ✓ Graph d'exécution
 - ✓ Détection du code jamais exécuté
- ✓ Propagation des constantes
- ✓ Analyse des boucles
 - ✓ Dans les langages fonctionnel, remplacer les récursions par des boucles
- ✓ Analyse si les variables sont utilisés et s'il est nécessaire de les calculer

Algorithme récurrent

- ✓ Observation : chaque optimisation peu en révéler une autre
- ✓ Exemple : la propagation des constantes peut révéler une condition toujours vraie

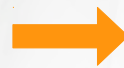
a := 5

if a > 0

 print a

else

 print "error"



print 5

Algorithme récurrent

- ✓ Solution :
- ✓ appliquer toutes les optimisations
- ✓ recommencer jusqu'à ce que plus aucune optimisation ne soit possible

Conclusion

- ✓ Naïvement on pourrait penser que C++ est plus performant que Java ou C# (compilé vs interprété)
- ✓ Java et C# sont compilés à la volé (JIT)
- ✓ Le langage intermédiaire permet des optimisation plus poussées
- ✓ L'assembleur C++ doit être compatible avec plusieurs processeurs
- ✓ Conclusion Java et C# on de meilleures performances

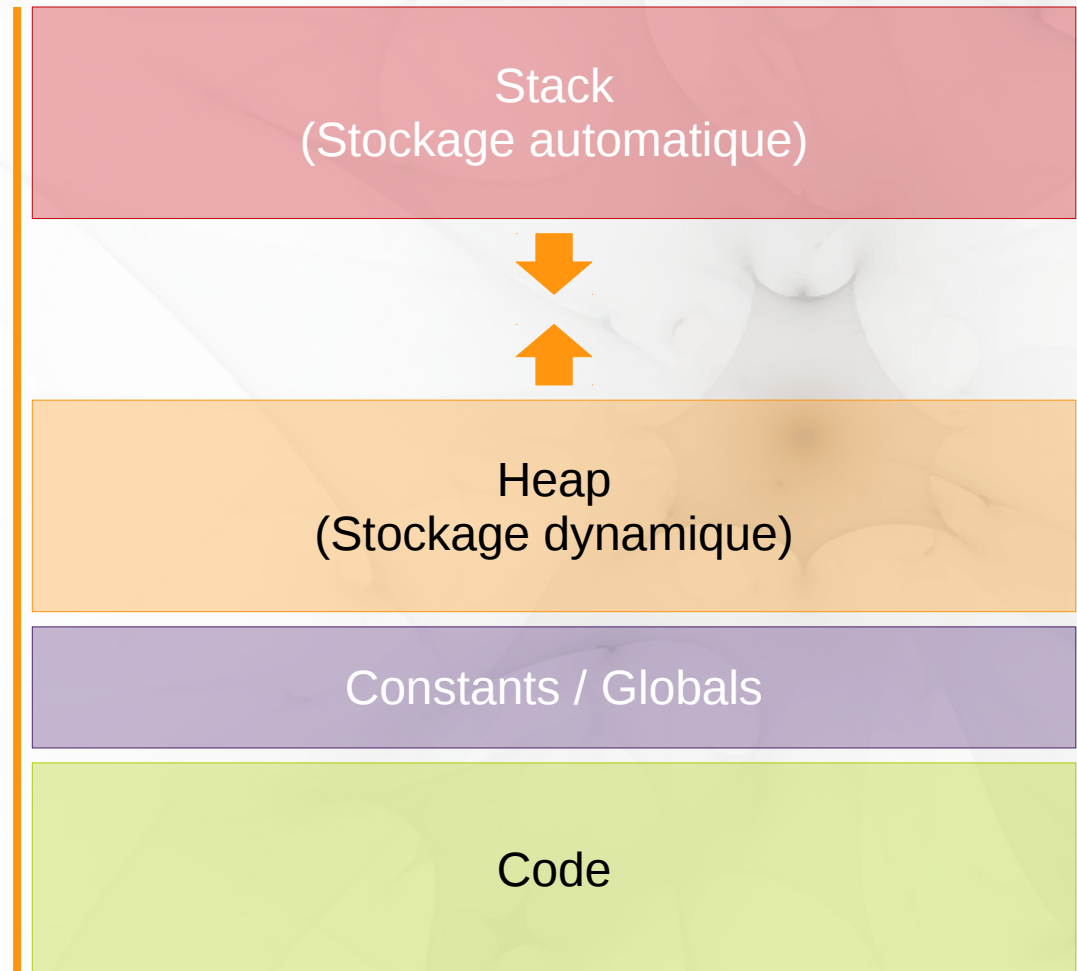
Code Generation



Mémoire d'un programme

- ✓ Un programme a un espace mémoire fini
- ✓ Il a besoin de différentes zones :
 - ✓ Sa propre zone, stocké lui même (fixe)
 - ✓ Une zone pour les constantes (fixe)
 - ✓ Une zone pour l'allocation des objets (variable)
 - ✓ Une zone pour effectuer des calculs ou stocker les portés (scopes) (variable)
- ✓ Comment stocker deux zones variables dans une zone mémoire fixe ?

Mémoire d'un programme



- ✓ Code (code source)
- ✓ Constantes
- ✓ Heap (les objets)
- ✓ Stack (scopes)

Mémoire d'un programme

- ✓ La stack représente une pile (fil), l'ordre de libération de la mémoire est relative à celle de l'allocation
 - ✓ Contient les variables qui sont libérées lorsque le programme sort du scope
- ✓ A contrario, le heap est un espace alloué et libéré de façon indépendantes
 - ✓ Contient les objets créés et libérés
 - ✓ Malloc et free en c, new / gc en java

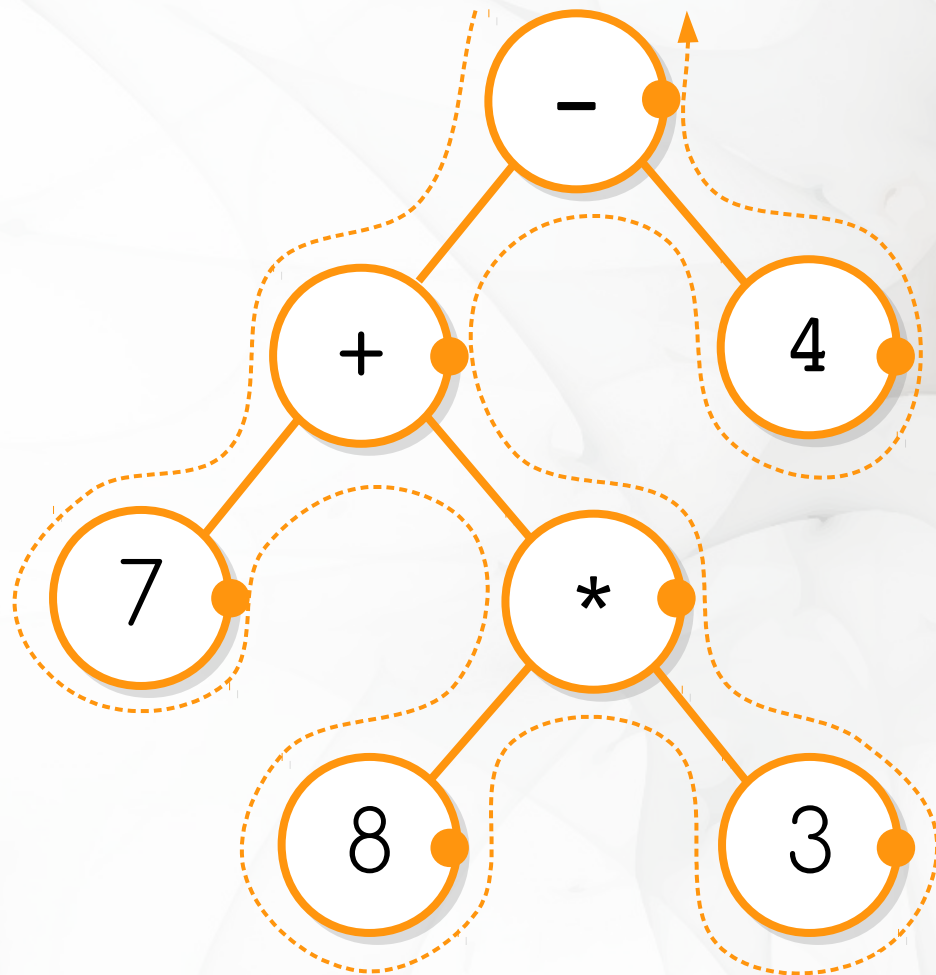
Mémoire d'un programme

- ✓ Lorsqu'il n'y a plus de mémoire possible : Stack Overflow ou Out of memory (heap)
- ✓ Stack overflow arrive le plus fréquemment lors d'appel récursifs qui dépassent la limite
- ✓ Out of memory, lorsque les objets occupent toute la place mémoire
 - ✓ fuite mémoire en C++
 - ✓ liens persistants en Java (GC)

Stack machine

- ✓ Basé sur une pile d'exécution
- ✓ Avantages :
 - ✓ Simplicité d'implémentation
 - ✓ Nombre d'instruction réduit
 - ✓ Code machine généré réduit par rapport à une machine à registre (code 2x plus important)
 - ✓ Accès rapide aux opérandes

Stack machine



```
// reverse polish  
((7 (8 3 *) +) 5 -)
```

```
// stack machine opcode  
push 7          // 7  
push 8          // 7 8  
push 3          // 7 8 3  
multiply        // 7 24  
add            // 31  
push 4          // 31 4  
subtract       // 27
```


Stack machine

```
// reverse polish  
((7 (8 3 *) +) 5 -)
```

```
// stack machine opcode  
push 7      // 7  
push 8      // 7 8  
push 3      // 7 8 3  
multiply    // 7 24  
add         // 31  
push 4      // 31 4  
subtract    // 27
```

			*						
		3	3		+			-	
	8	8	8	24	24		4	4	
7	7	7	7	7	7	31	31	31	27

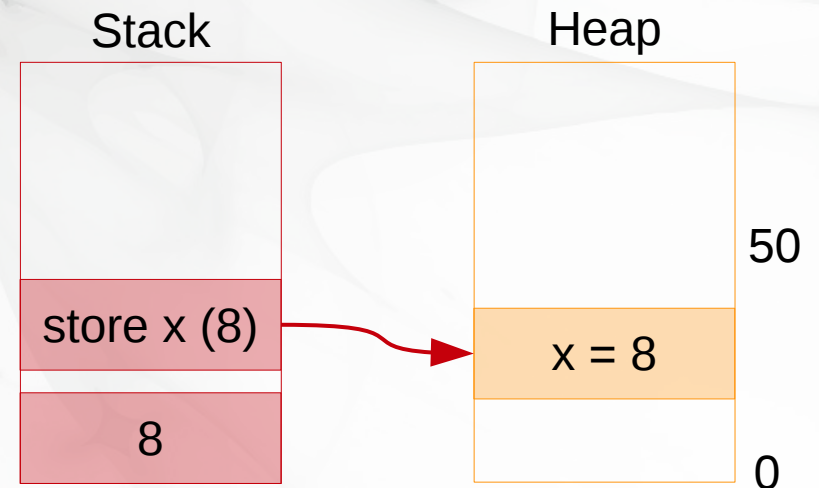
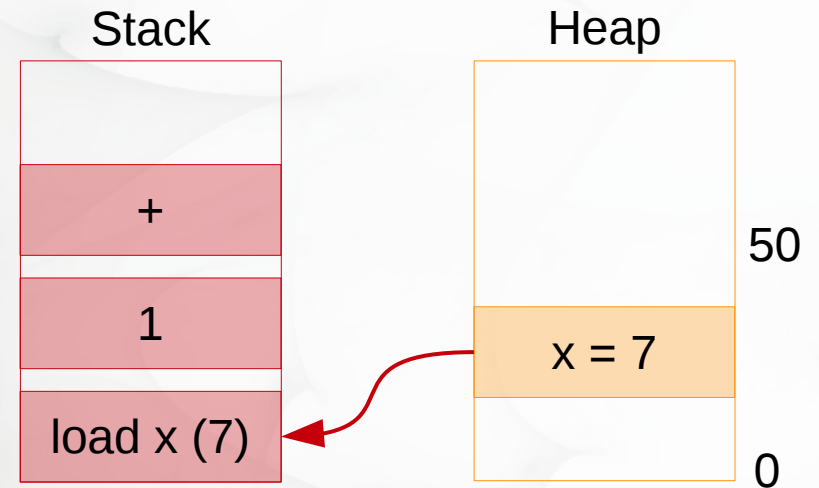
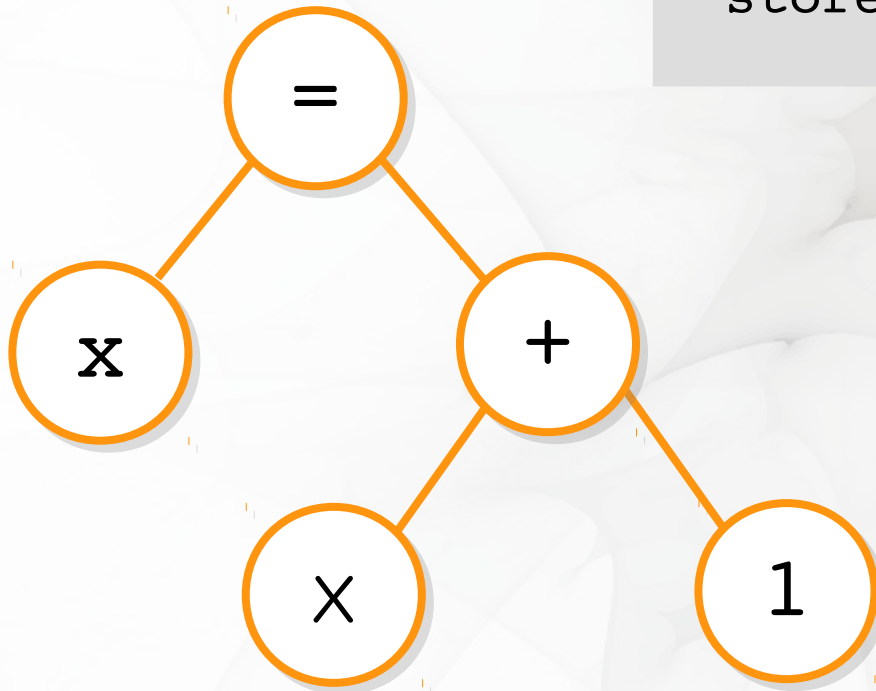
Temps

Stack machine

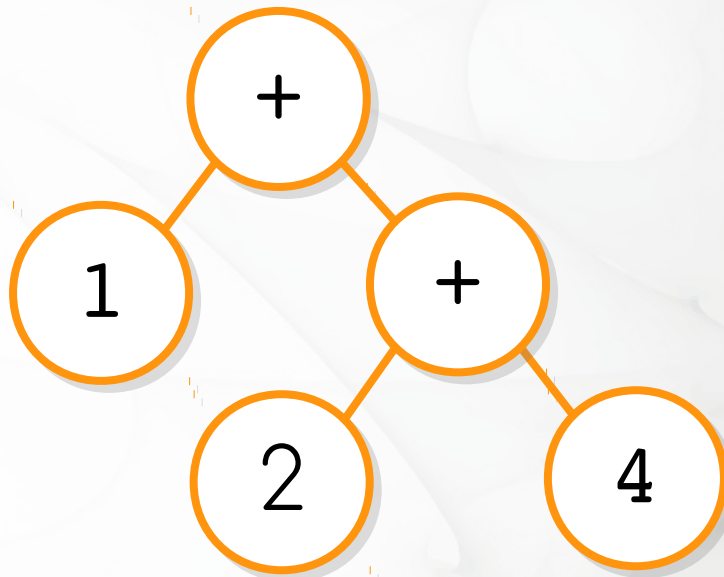
- ✓ Inconvénient :
- ✓ Nécessité de stocker les résultats intermédiaires en mémoire (coût d'accès)
- ✓ Nécessite un surcroît d'appel pour stocker et récupérer les résultats
- ✓ JVM utilise une modèle hybride avec une stack et un registre

Stack machine

```
load x
push 1
add
store x
```



Register machine



```
store r0, 2
store r1, 4
add
store r1, 1
add
```

Registre

r0	2
r1	

Registre

r0	2
r1	4

Registre

r0	6
r1	

+

Registre

r0	6
r1	1

Registre

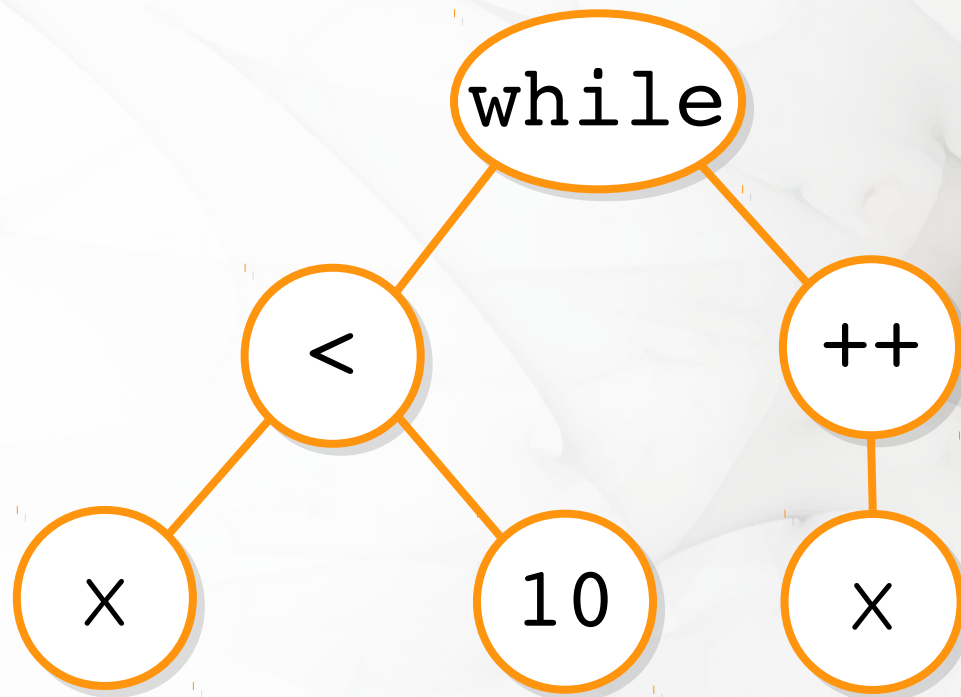
r0	7
r1	

+

Structures de Contrôle

- ✓ Le langage machine ne connaît que 2 instructions de contrôle :
- ✓ « if » et « goto »
- ✓ En programmation impérative structurée, ces instructions ont été remplacées par des instructions de plus haut niveau :
- ✓ for, do / while, repeat / until, if / then / else

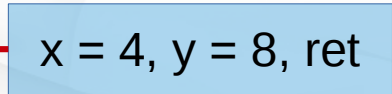
Structures de contrôle



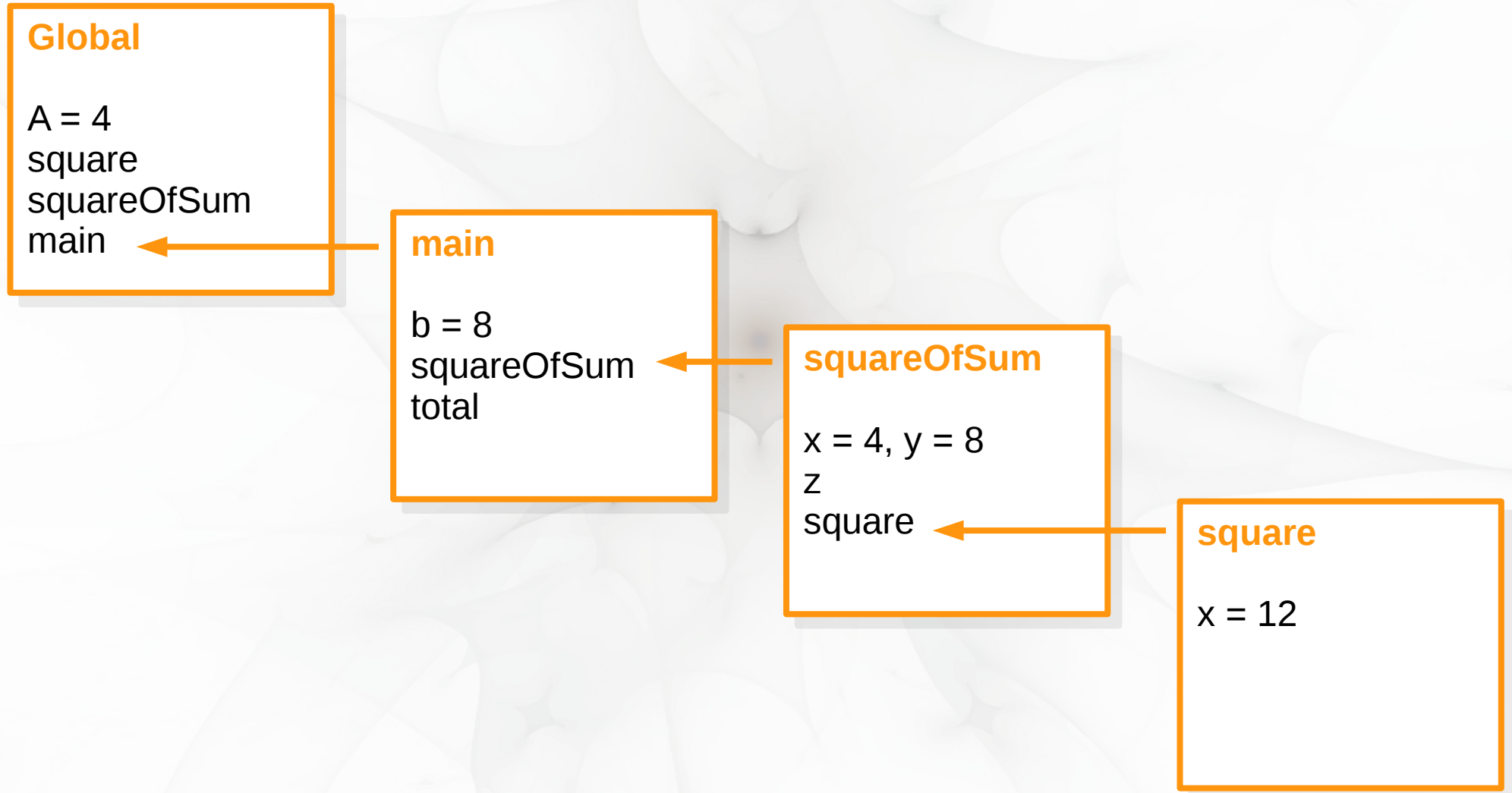
```
L0 // test
    load x
    push 10
    jump_if_lt L1
    jump L2
L1 // body
    load x
    push 1
    add
    store x
    jump L1
L2 // exit
```

The assembly code block illustrates the control flow of the while loop. It is organized into three labels: L0 (test), L1 (body), and L2 (exit). An orange arrow points from the 'while' node of the AST to the L0 section. Another orange arrow points from the '<' node to the 'jump_if_lt L1' instruction. A third orange arrow points from the '++' node to the 'jump L1' instruction. A fourth orange arrow points from the 'jump L1' instruction back to the L1 section, representing the loop's continuation. A final orange arrow points from the 'jump L1' instruction to the L2 section, representing the loop's exit.

Démonstration

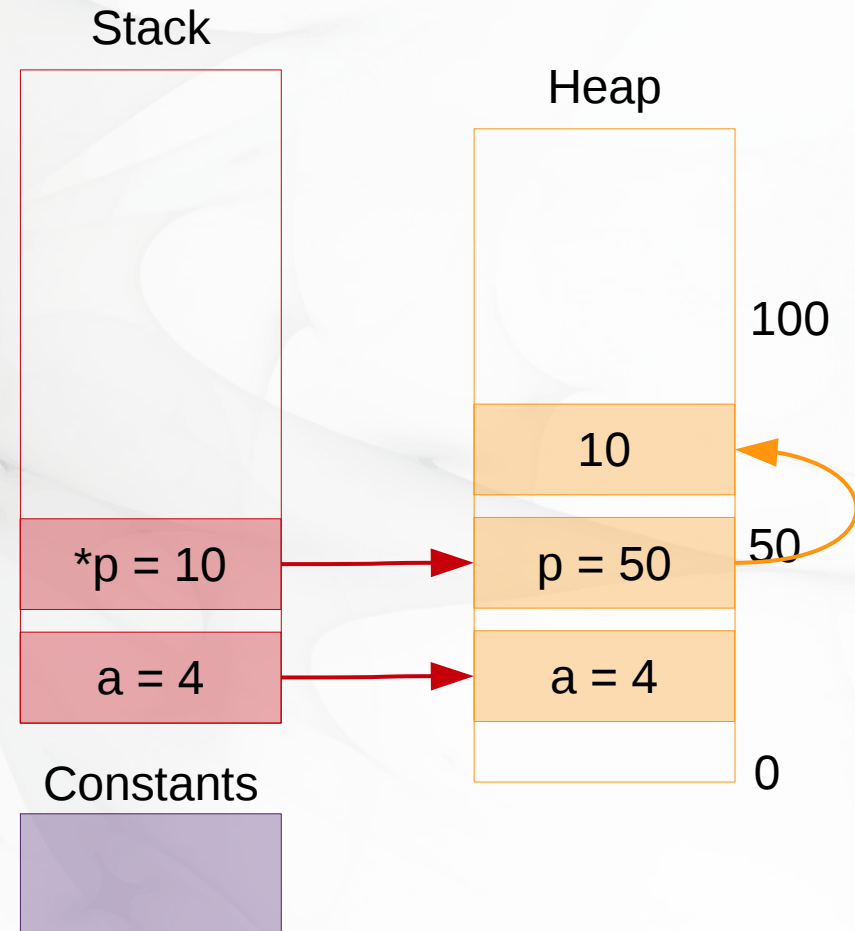


Portée du programme



Pointeurs

```
int main () {  
    int a = 4;  
    int *p;  
  
    p = malloc(sizeof(int));  
    → *p = 10;  
  
    // attention ! free  
    free(p);  
  
    p = malloc(sizeof(int));  
    *p = 20;  
}
```



Garbage collector



Garbage collector

- ✓ La gestion de la désallocation de la mémoire est automatique en Java
- ✓ Grâce au ramasse miette (garbage collector)
- ✓ La JVM en implémente plusieurs algorithmes

Gestion par Destructeur

- ✓ Contrairement au langage C++ dont la gestion s'effectue par le destructeur
- ✓ A la charge du développeur
- ✓ Nombreux inconvénients :
 - ✓ Fastidieux
 - ✓ Risque de fuites mémoire
 - ✓ Risque d'appel d'éléments détruits
 - ✓ Risque de désallocations multiples

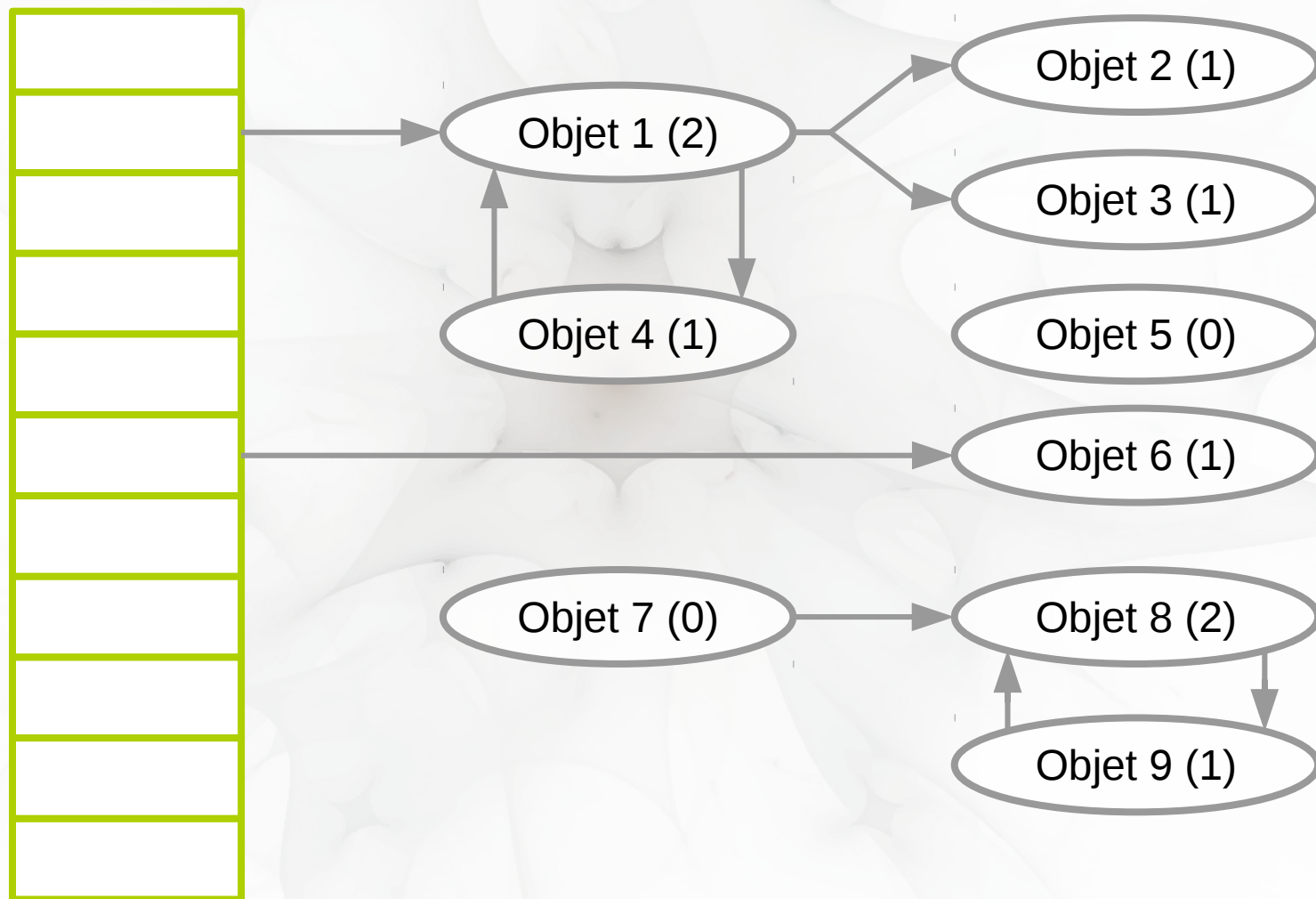
Algorithmes

- ✓ Reference counting
 - ✓ Pose problèmes
- ✓ Mark and sweep
 - ✓ Utilisé par la JVM
- ✓ Tri-color marking

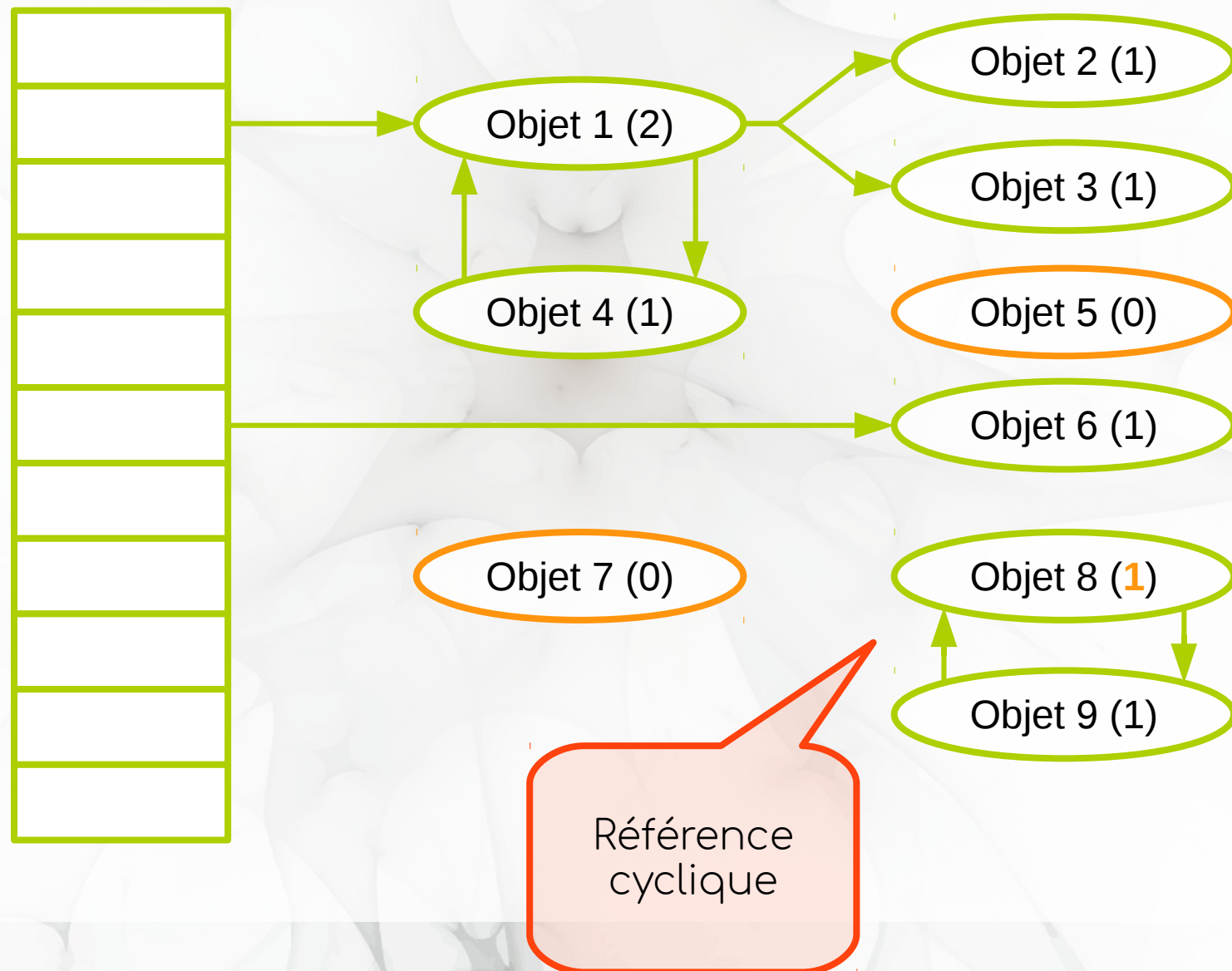
Reference Counting

- ✓ Principe :
 - ✓ Pour chaque objet, un compte est entretenue
 - ✓ Incrémenté en cas d'une nouvelle référence
 - ✓ Décrémenté en cas de destruction d'un parent (parcours en profondeur du graphe)
 - ✓ Quand le compteur est à 0, l'objet peut être détruit

Reference Counting



Reference Counting



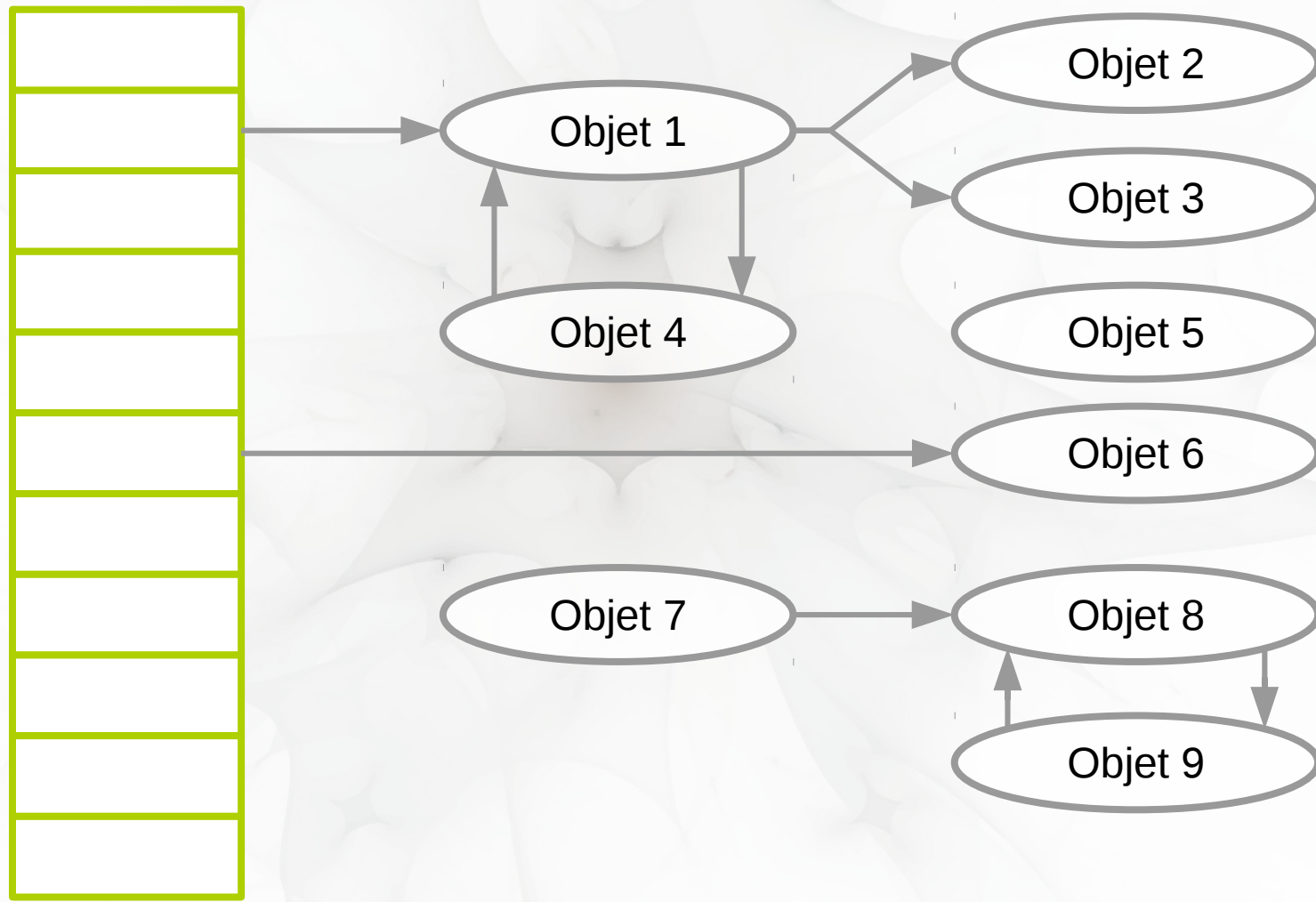
Reference Counting

- ✓ Avantages :
 - ✓ Mis à jour en temps réel
 - ✓ Complexité et coûts mémoire faibles
- ✓ Inconvénients :
 - ✓ Cas des références cycliques (des objets se référençant mutuellement)

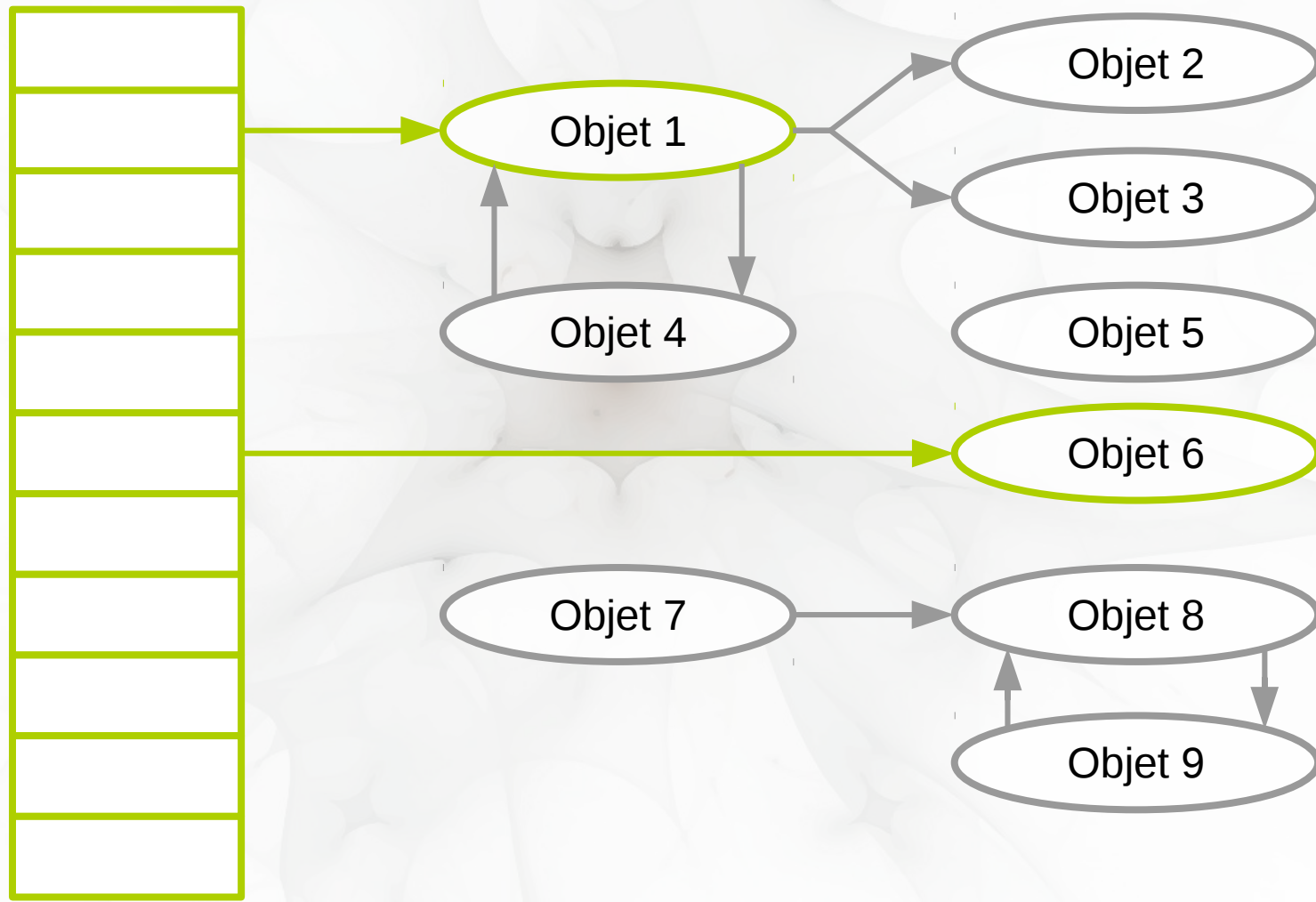
Mark and Sweep

- ✓ Principe :
 - ✓ Nécessite un flag booléen par objet
 - ✓ Étape 1 : Parcours en profondeur de l'arbre et marquage des objets utilisés
 - ✓ Étape 2 : Suppression des objets non marqués
 - ✓ Étape 3 : Nettoyage des flags

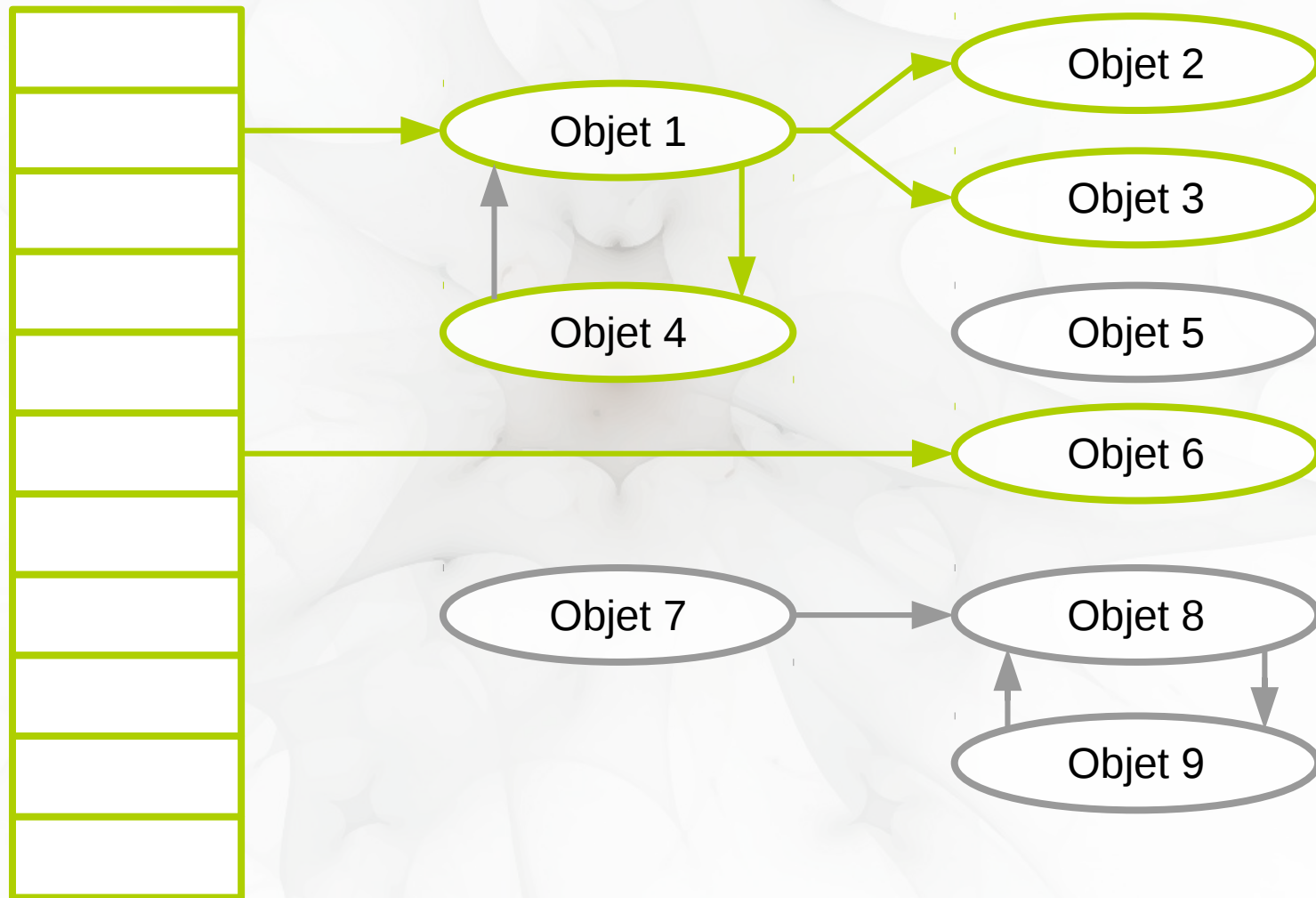
Mark and Sweep



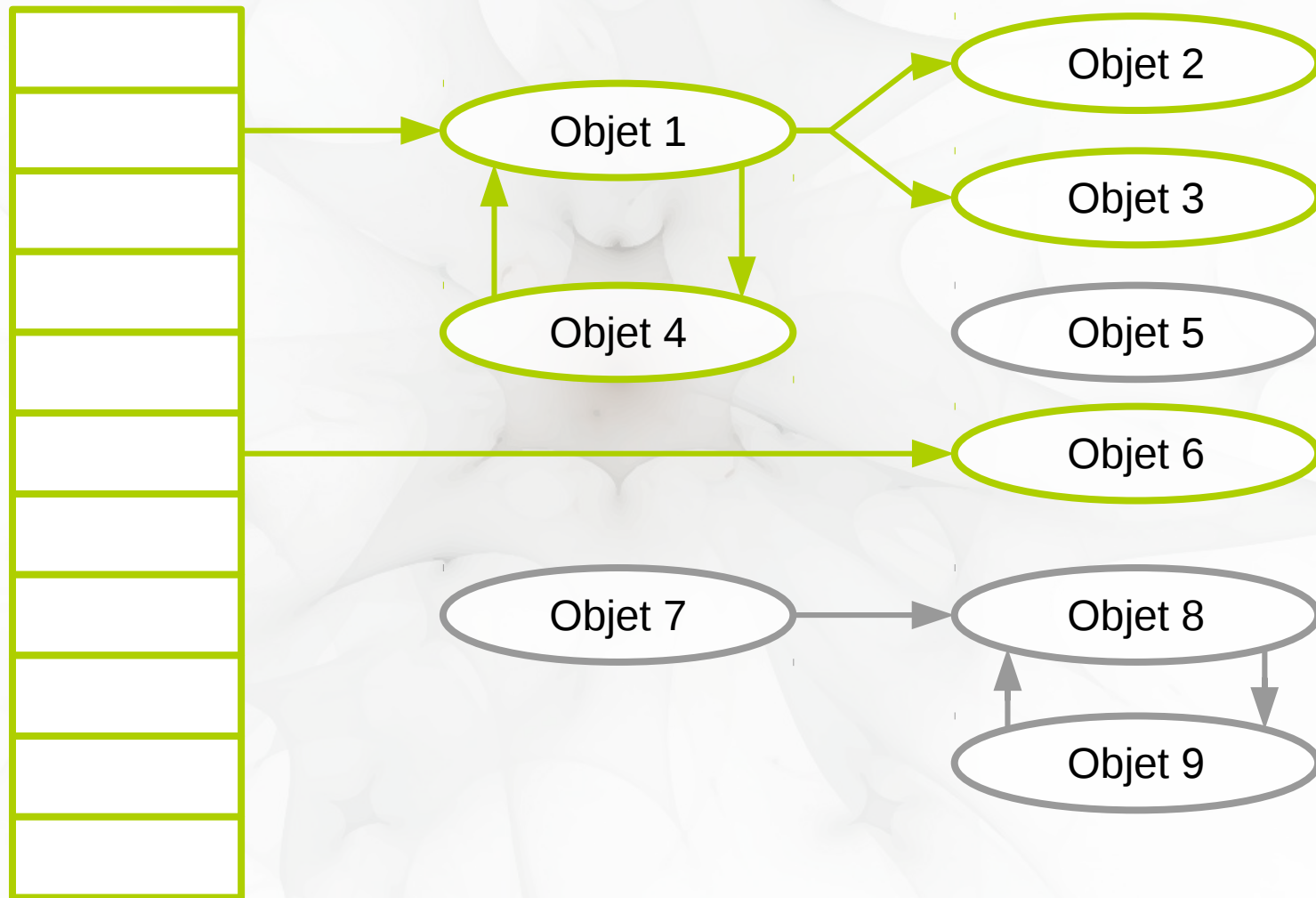
Mark and Sweep



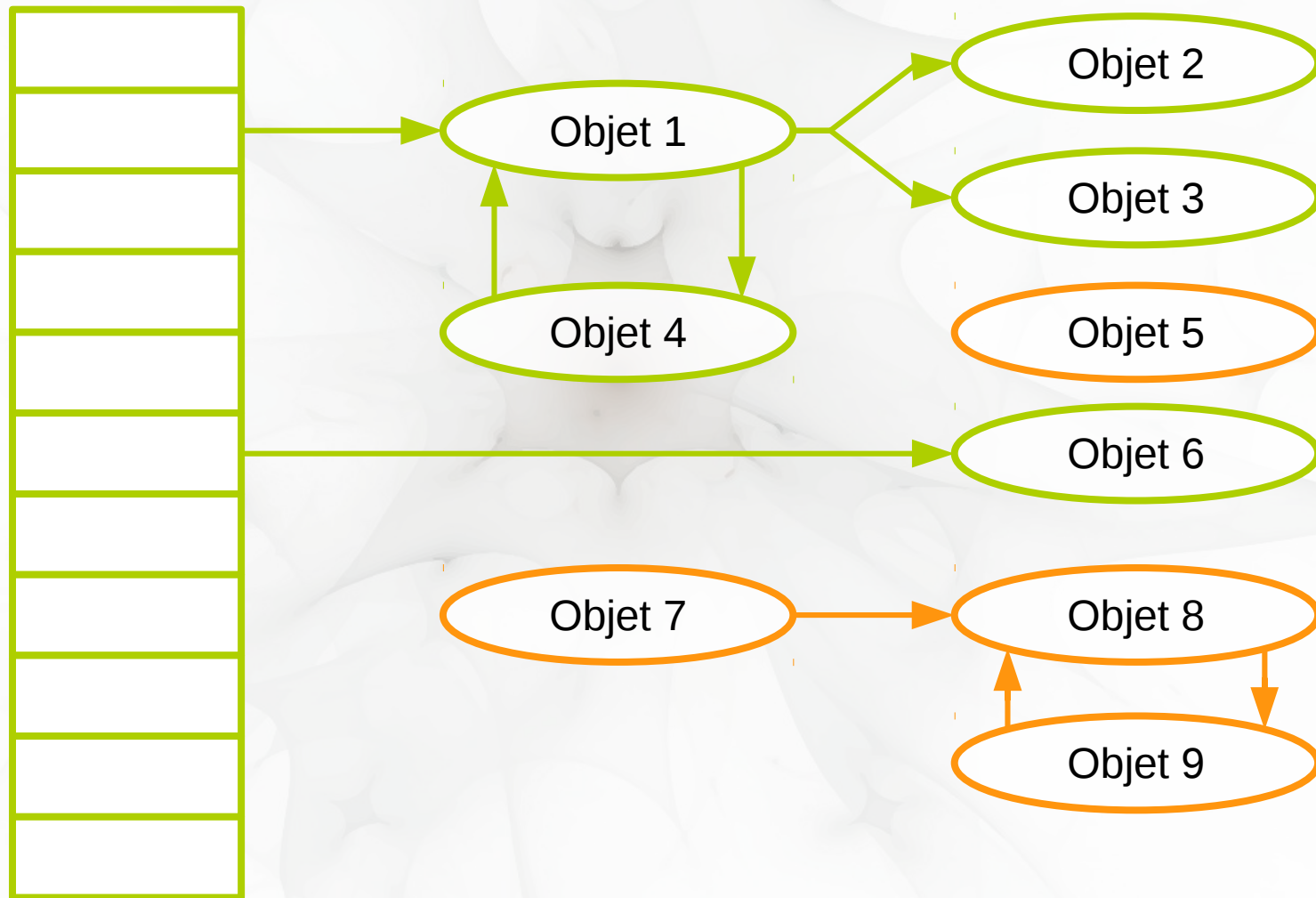
Mark and Sweep



Mark and Sweep



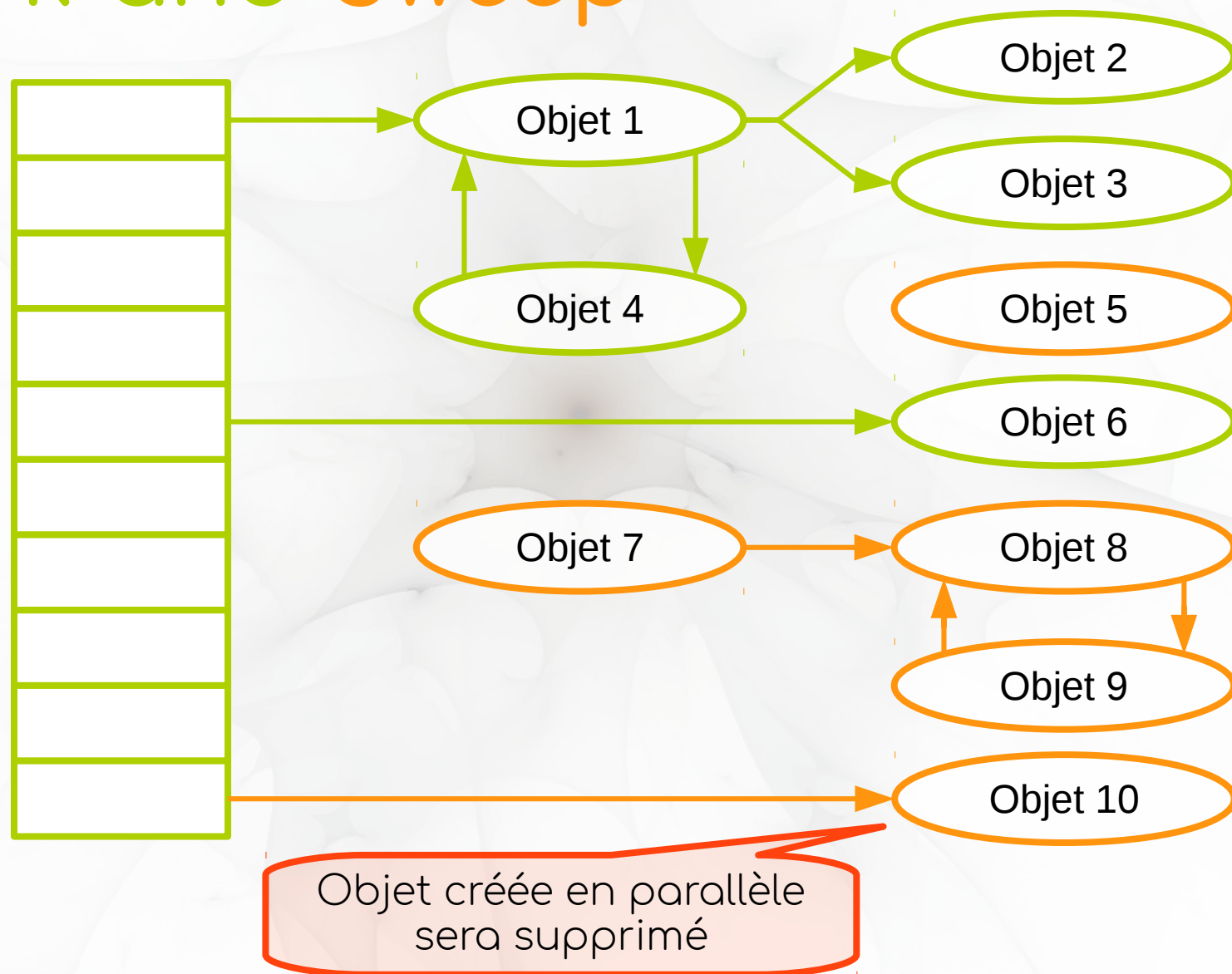
Mark and Sweep



Mark and Sweep

- ✓ Avantages :
 - ✓ Élimination des références circulaires
- ✓ Inconvénients :
 - ✓ Ne détecte pas si l'objet est dans du code inaccessible (peu important)
 - ✓ Nécessite un arrêt des accès mémoire (stop the world) → rends les application « temps réel » ou « time critical » impossibles

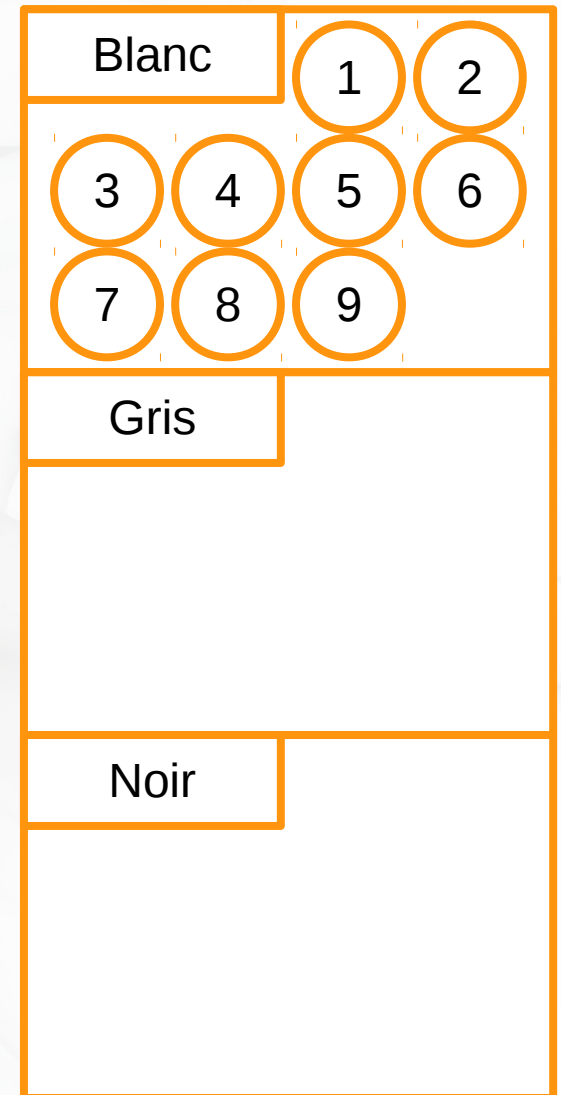
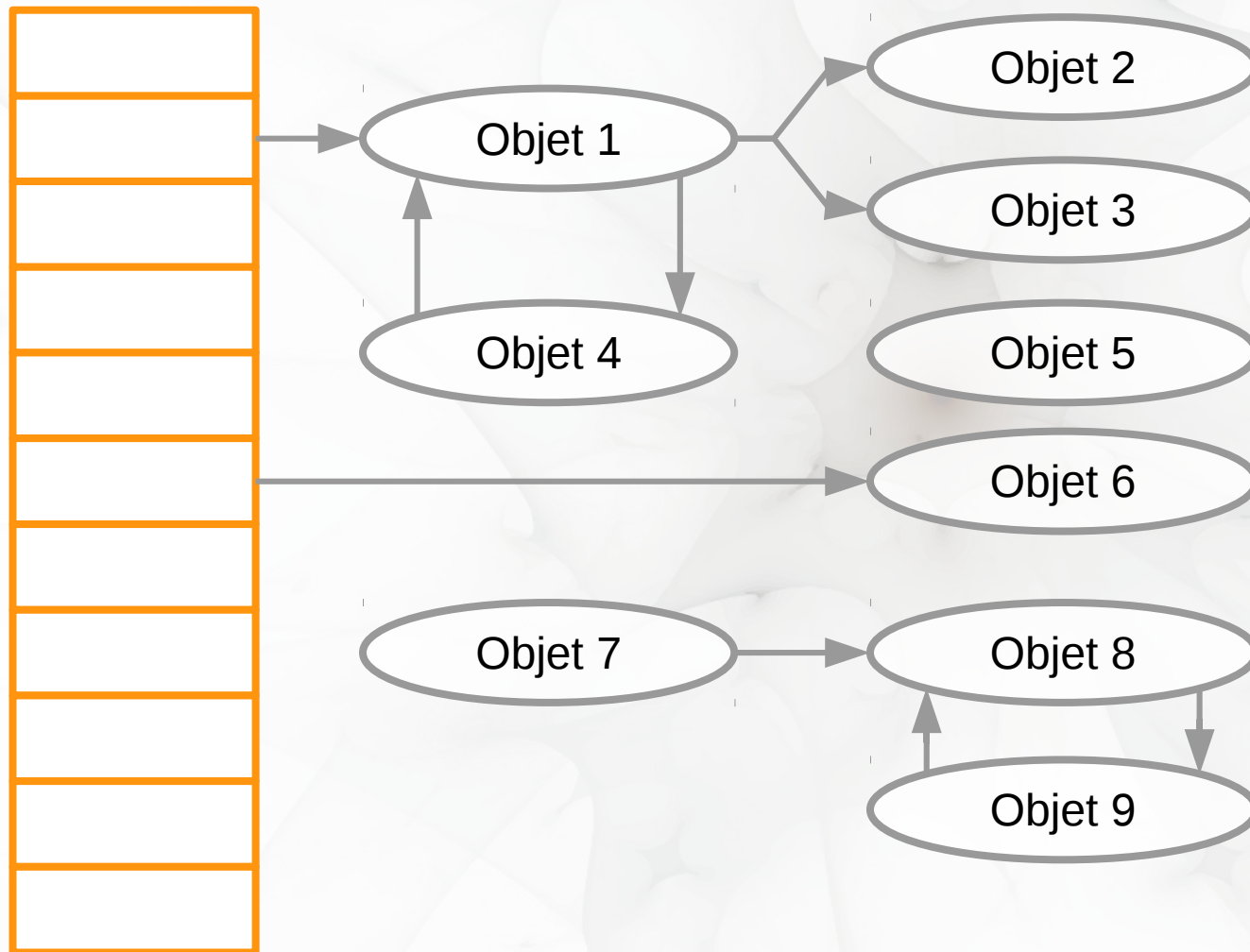
Mark and Sweep



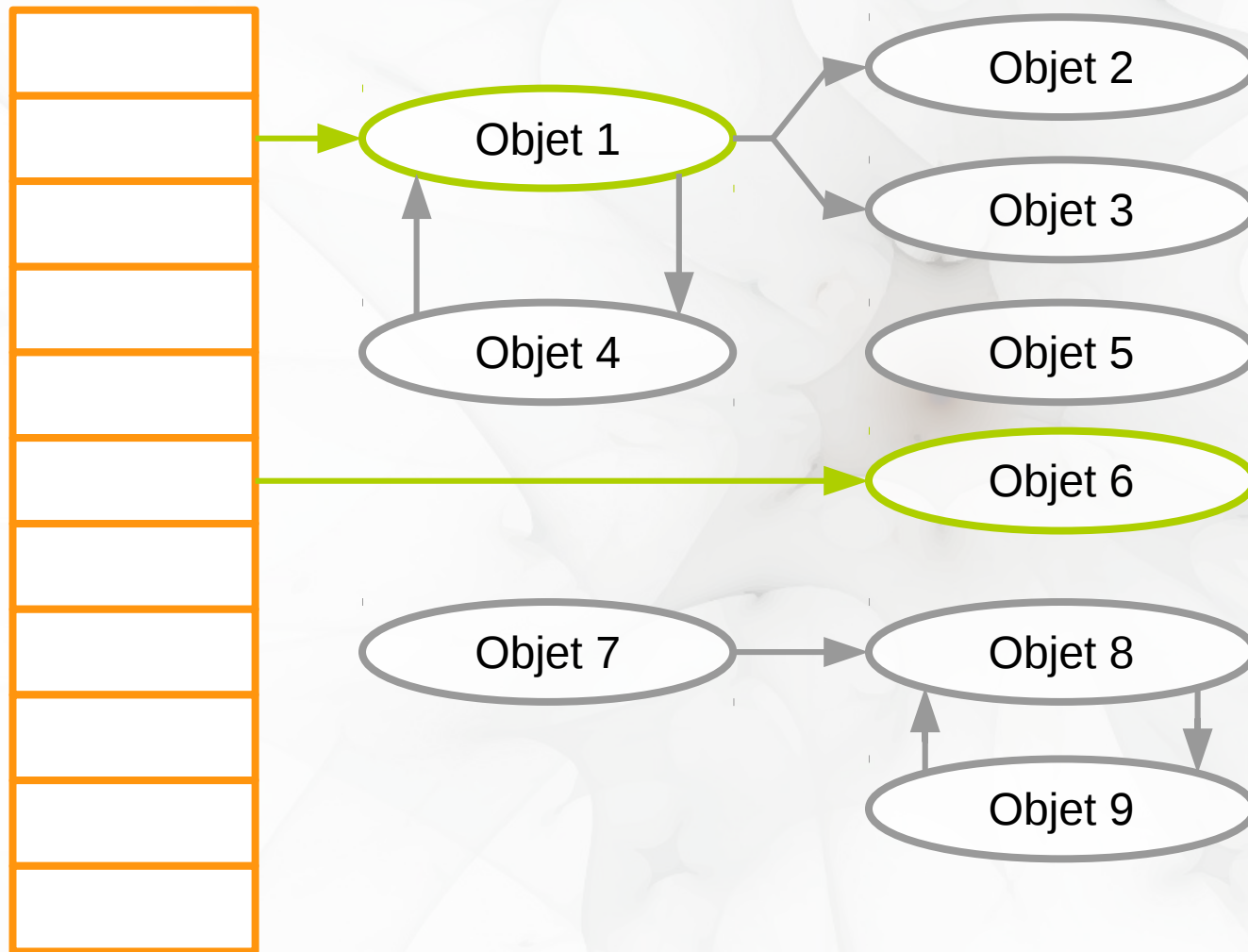
Tri-color marking

- ✓ Principe :
 - ✓ Trois ensembles sont nécessaire
 - ✓ Blanc contient tous les objets à évaluer
 - ✓ Noir, objets qui n'ont pas de lien vers un objet de l'ensemble blanc et qui sont atteignable depuis la racine (non candidats à la collection)
 - ✓ Gris, objets atteignable depuis la racine mais dont les liens n'ont pas encore été évalués

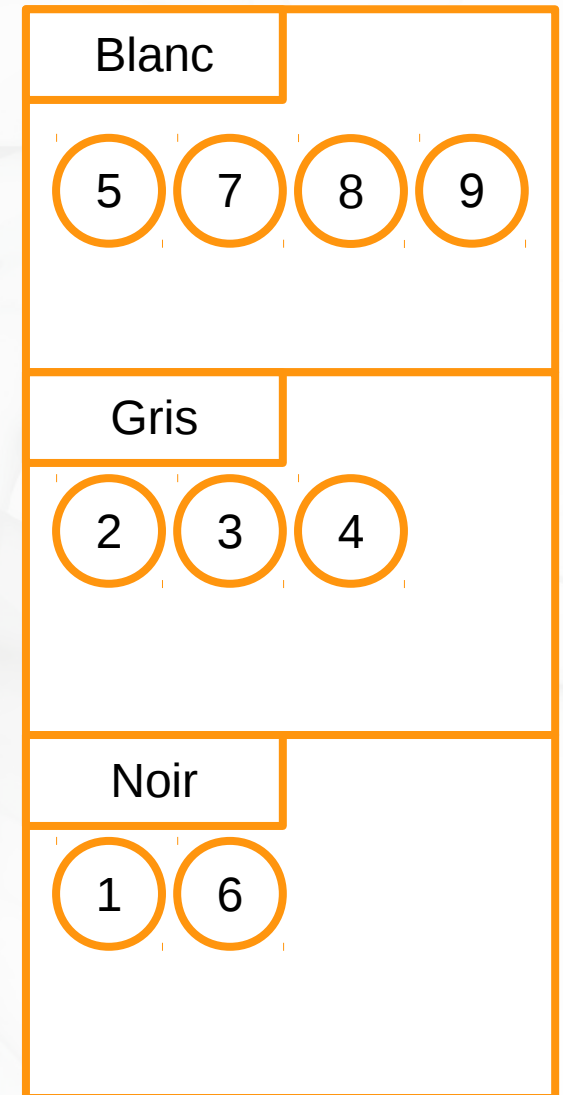
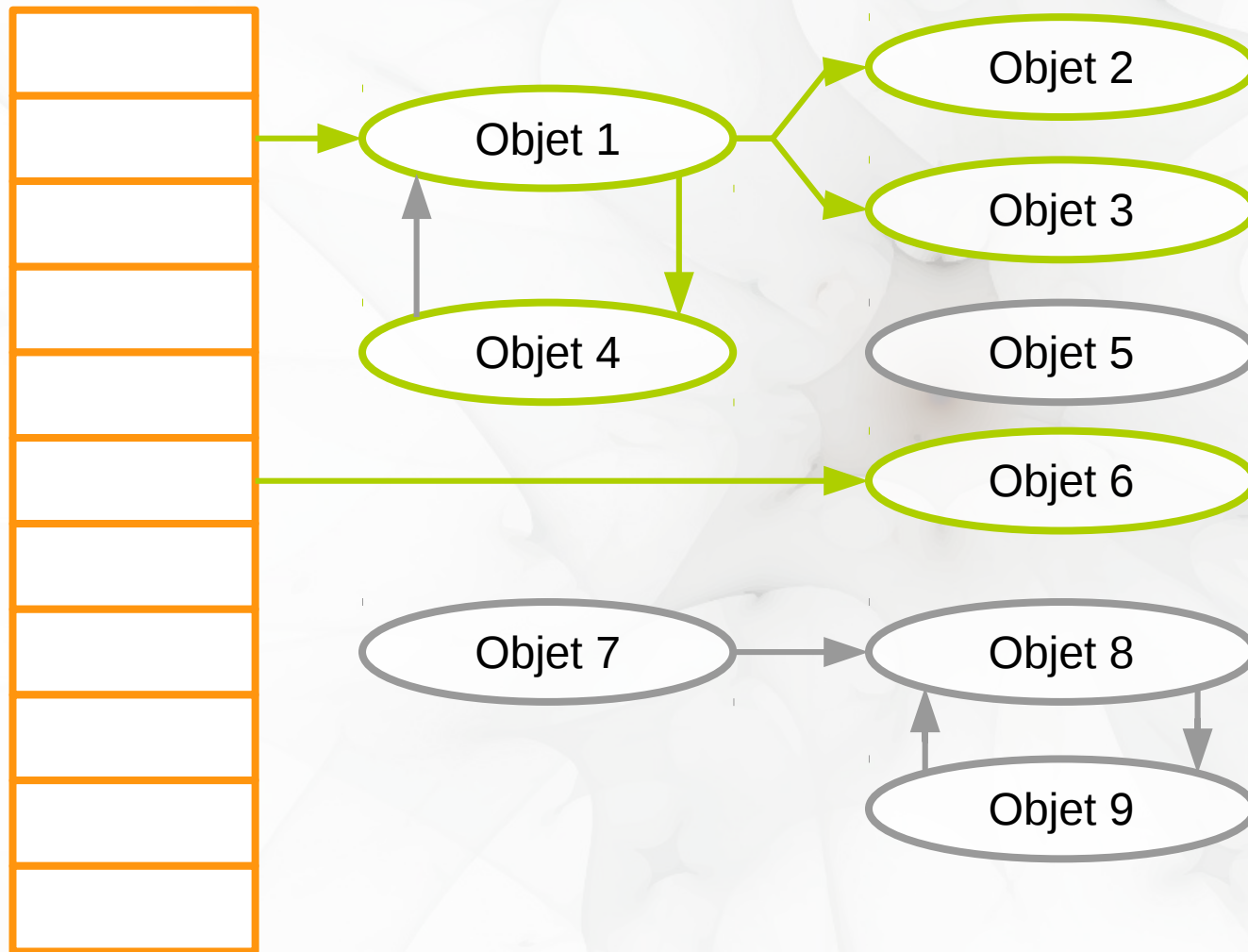
Tri-color marking



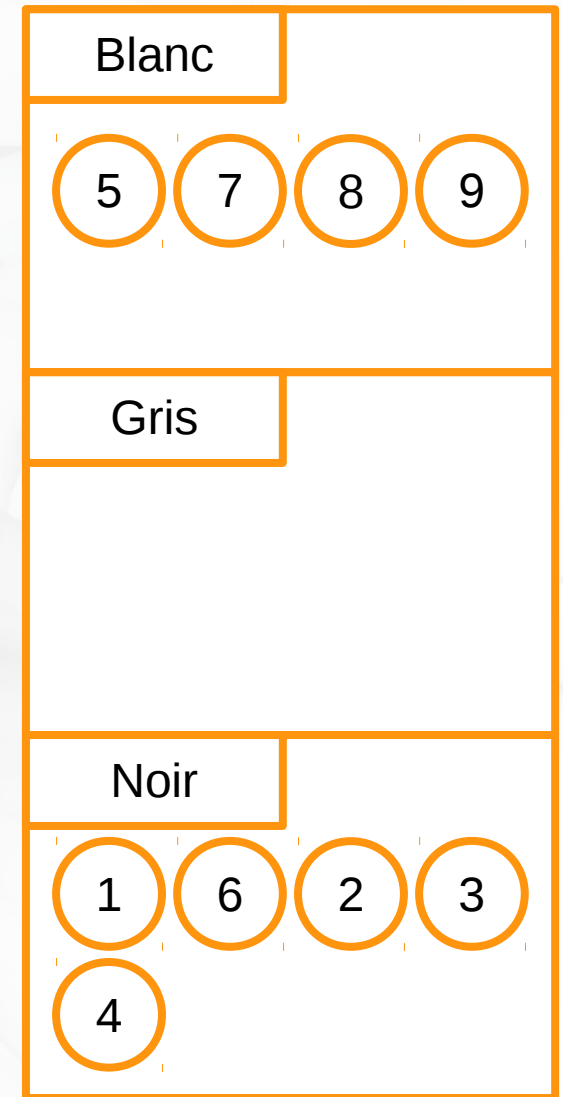
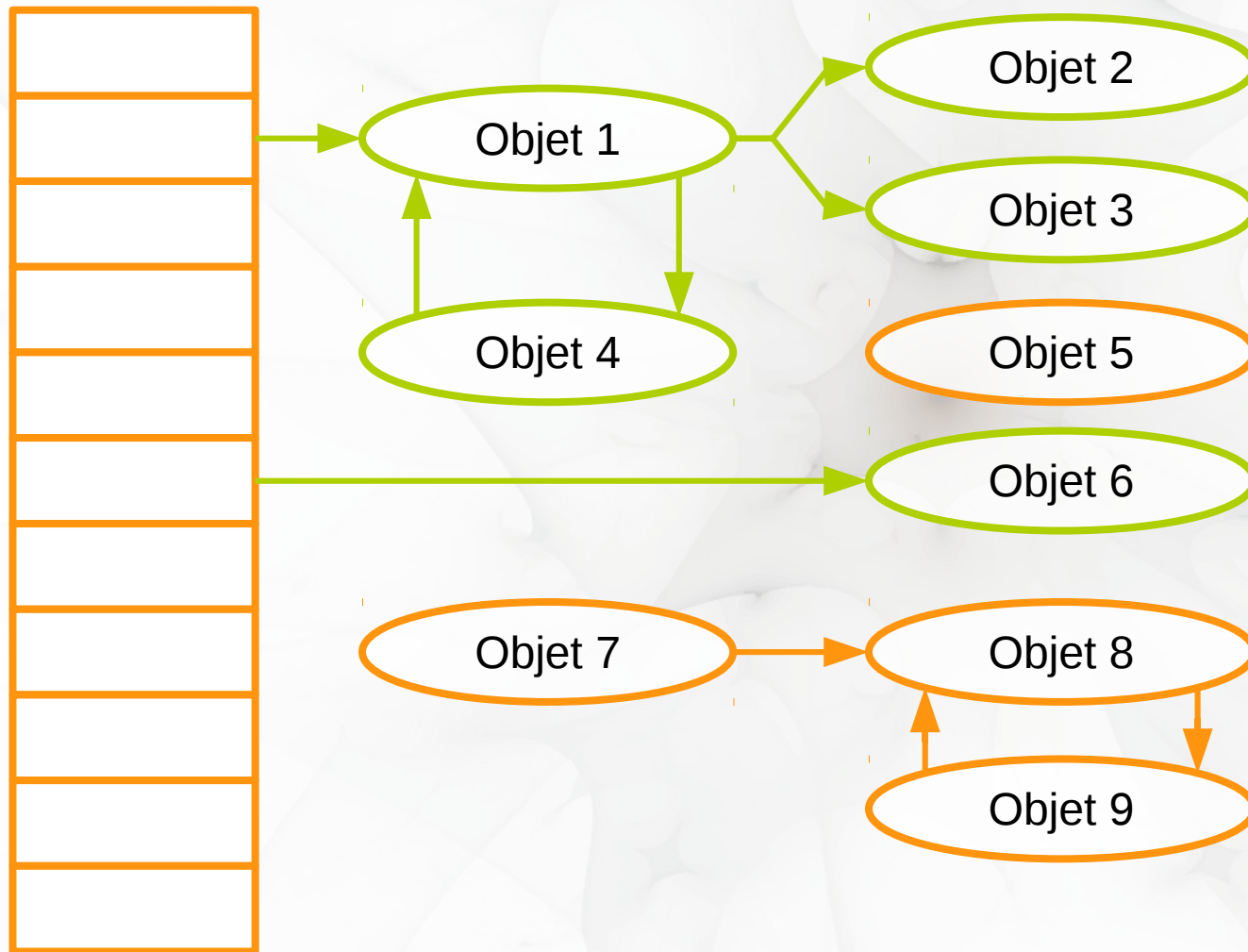
Tri-color marking



Tri-color marking



Tri-color marking



Tri-color marking

- ✓ Principe :
 - ✓ L'ensemble blanc fige l'espace de travail
 - ✓ En cas de génération de nouveaux objets durant l'exécution, ils ne seront pas pris en compte
 - ✓ Si des objets sont déréférencés durant l'exécution, ils seront évalués à la prochaine exécution
 - ✓ L'ensemble gris sert à décider si l'algorithme a fini (lorsqu'il est vide)

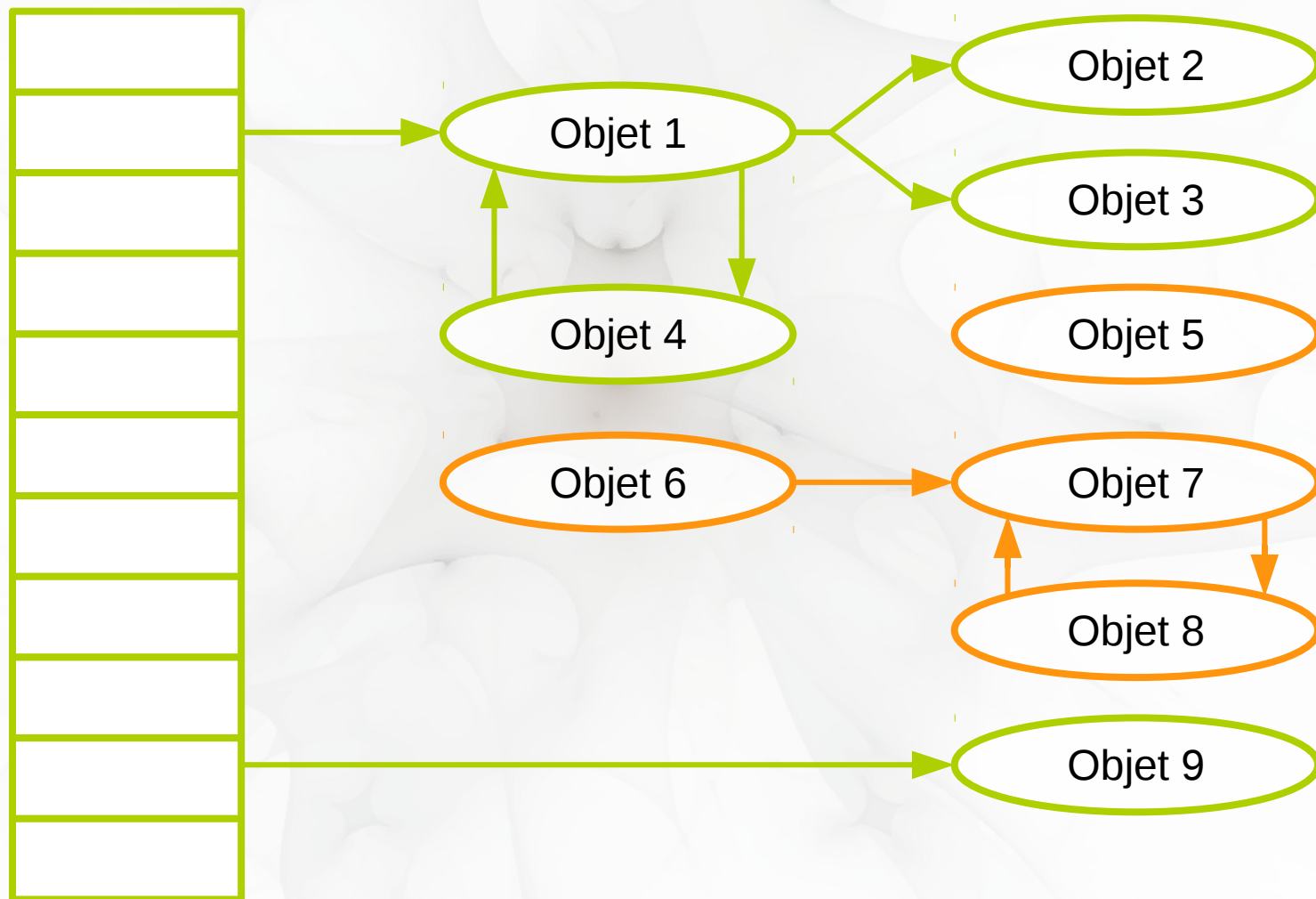
Tri-color marking

- ✓ Les objets ne peuvent être déplacés que du blanc vers le gris et du gris vers le noir
- ✓ Comme aucun objet noir ne peut référencé un objet blanc, une fois que l'ensemble gris est vide les objets peuvent êtres collectés
- ✓ Avantage :
 - ✓ Élimination des références circulaires
 - ✓ Peut être exécuté en parallèle du programme

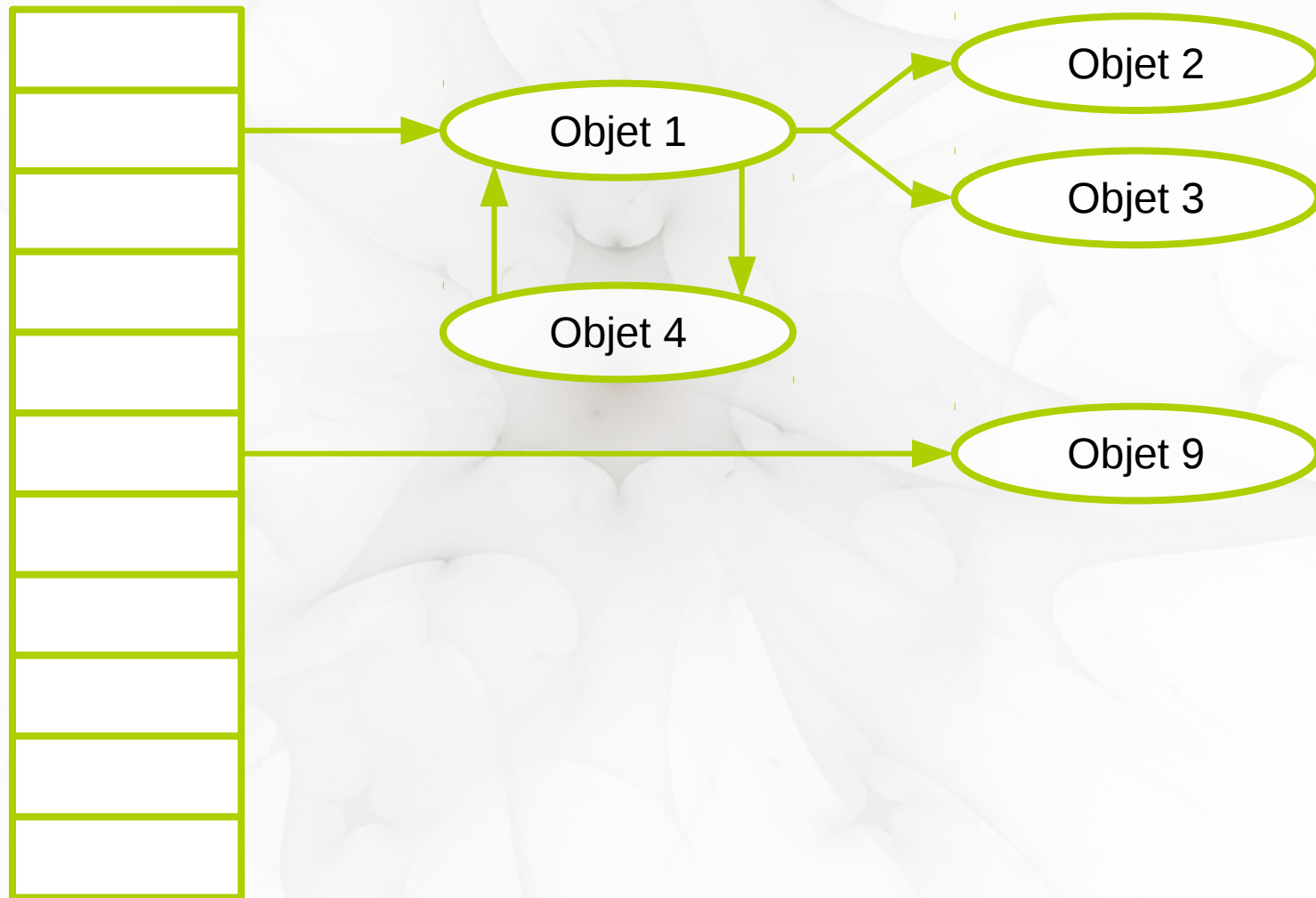
Moving vs Non-Moving

- ✓ Après la collections des objets inaccessibles, les objets restants peuvent rester sur place
- ✓ Ou être recopier dans une nouvelle zone mémoire
- ✓ But : éviter la fragmentation de la mémoire avec des zones vides
- ✓ Désavantages :
 - ✓ Coût de la copie
 - ✓ Ne permet plus l'arithmétique de pointeurs

Moving



Moving



Stop the world vs Concurrent

- ✓ Les algorithmes Stop the world, suspendent l'exécution du programme lors de la collection
- ✓ Pose des problèmes de performance
- ✓ Rende impossible les programmes Temps réel
- ✓ Les algorithmes Concurrents rendent possible la collection sans que le programme ne soit suspendu

Merci de votre attention

