

IDE - Debuggage, Profiling

V. Deslandres

IUT de LYON

CVDA, s2

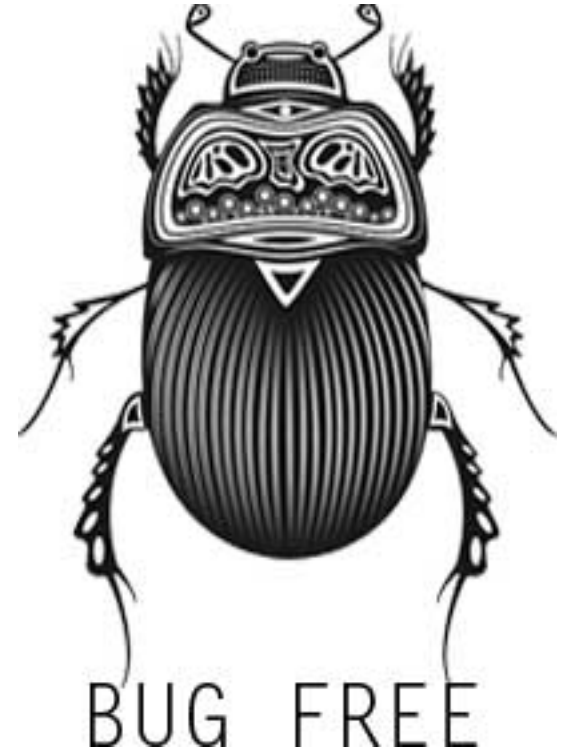


Sommaire

- Débugage
 - Sous Netbeans ----- [10](#)
- Profiling / gestion de la mémoire --- [20](#)

Où est la petite bête ?

DÉBUGGAGE



Pourquoi déboguer ?

- Pour ne plus avoir d'erreurs, OK
- Mais pourquoi est-ce un **processus à part entière** aujourd'hui ?

à votre avis ?

Evolution des langages

- **Complexité accrue**
 - Plus de possibilités
 - Plus d'ambiguïtés
 - De plus en plus d'outils sont mis à disposition pour aider le développeur : les IDE
- Un IDE regroupe des outils pour :
 - Le debugage
 - Gestion de la mémoire
 - Analyse de performances
 - Versioning (pour revenir à une version moins bugguée)

Définition

- Un « bug » est une défaut de code écrit par mégarde par un programmeur
- C'est bien souvent une erreur de logique dans son code, par exemple :
 - Une boucle *while* infinie,
 - L'oubli d'initialisation d'une variable,
 - Une valeur d'intervalle erronée dans une boucle.
- Les **tests** permettent de détecter les dysfonctionnements, mais il faut trouver les bugs associés.

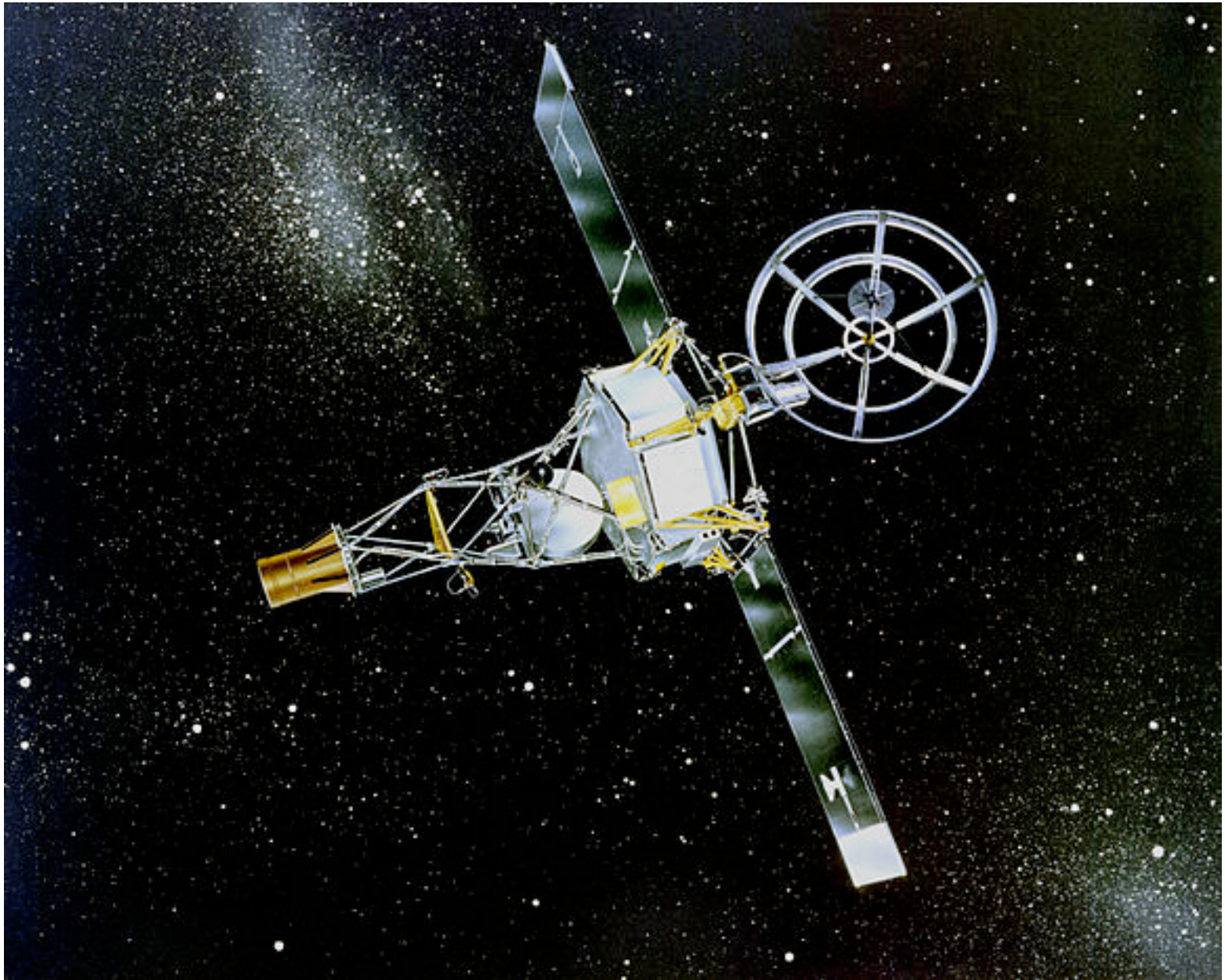
- Les bugs peuvent être difficiles à trouver
- Certains causent des catastrophes : en 1962, une mauvaise interprétation d'une équation mathématique* dans le programme de guidage de la fusée Mariner 1 l'a conduit à sa perte...

* Oubli du symbole mathématique (une barre suscrite) dans une équation : \bar{R}_n , qui signifie « la n -ième valeur **lissée** de la dérivée de R dans le temps », sans lissage, il y avait n valeurs qui, chacune, orientait la fusée différemment → sans maîtrise, la Terre a décidé de détruire la fusée

– Coût total : 18 000 000 \$ US (1962)

– http://en.wikipedia.org/wiki/Mariner_1

Mariner 1





Autre exemples

- La machine de radiothérapie Therac-25 a causé la mort de 3 patients en partie à cause d'une erreur de logique très subtile
- <http://en.wikipedia.org/wiki/Therac-25> et [http://en.wikipedia.org/wiki/Race condition](http://en.wikipedia.org/wiki/Race_condition)

Débugage sous IDE



- Les IDE proposent d'exécuter le programme pas à pas et diverses actions pour aider à la détection de bugs
- Par ex. sous Netbeans :
 - Ouvrir un projet et faire **Debug › Step Into (F7)**
 - La commande **Debug › Step Over** (ou **F8**) permet d'avancer instruction par instruction
 - Dans la fenêtre de Débugage figure un onglet « Variables »
 - Vous voyez la valeur des variables changer
 - Pour forcer une nouvelle valeur à une variable, cliquez sur la valeur après le bouton  et la modifier (cf slide suivant)
 - Pour arrêter, faire **Debug › Finish Debugger Session**, ou cliquez sur le bouton 

Onglet « Variables »

Quand le programme est en pause



	Name	Type		
▼	◆ this	ExoDebugage	...	#118
▼	◆ Static		...	
	◆ x	int	...	6
	◆ y	int	...	0
	◆ a	int	...	6
	◆ b	int	...	6

Onglet « Watches »

Quand le programme est en pause








- Les 'watches' sont les éléments à observer
 - **Debug – New watch**
- Ca peut être :
 - Une expression (un éditeur s'ouvre)
 - Une instance de classe
 - Un champ
 - Un conteneur
- L'objet 'surveillé' est contextuel : ex.: `this.champ;`
- Quels sont les autres onglets possibles ?
 - À quoi servent-ils ? Call Stack par exemple ?

Clic droit sur un
élément surveillé:
Delete All les
supprime tous

Navigation en débogage

Instructions de navigation :

- **Step into (F7)** : Prochaine instruction 
 - (niveau de la pile identique)
- **Step over (F8)** : Avancer dans la fonction appelée 
 - (niveau de la pile +1)
- **Step out (Shift F8)** : Avancer jusqu'à la fin de la fonction sans déboguer dedans (niveau de la pile -1) 
- **Continue (Ctrl-F5)**:
 - Avancer jusqu'au prochain point d'arrêt 
- Exécuter jusqu'au curseur (F4) 

Breakpoints

- On peut d'autre part ajouter des **points d'arrêt** dans le programme
 - Pour éviter de tout balayer avec F8 : c'est long
- Les points d'arrêt = lancer le débogage en pas-à-pas **qu'à partir** d'une certaine ligne du programme
 - On choisit cette ligne avec soin : là où on pense que se cache le bug, ou un peu avant.
 - Se positionner sur la ligne et faire **Debug › Toggle Line Breakpoint** (ou .. ?). La ligne devient rouge.
 - Refaire la même chose = le désactiver

Point d'arrêt

```
27  
28 public static long multiplierDeuxVecteurs(long[] lvec1, long[] lvec2) {  
29     int taille;  
30     long lRetour = 0L;  
31  
32     if ((lvec1 != null) && (lvec2 != null)) {  
33  
34         if ((taille = lvec1.length) == lvec2.length) {  
35             for (int i = 0; i < taille; i++) {  
36                 lRetour += lvec1[i] * lvec2[i];  
37             }  
38         }  
39     }  
40     return lRetour;  
41 }
```



NetBeans IDE 8.1

Démo Netbeans

Utilisation Breakpoint

- Comme outil de diagnostic pour par ex. :
 - Détecter lorsque la valeur d'un champ ou variable locale est modifiée
 - peut aider à déterminer **quelle partie du code assigne une valeur** inappropriée à un champ
 - Détecter **lorsqu'un objet est créé**
 - ce qui peut être utile, par exemple, lorsqu'on essaie de pister une fuite mémoire

Autres points d'arrêt

- On peut définir des points d'arrêts pour d'autres cas que pour une instruction, comme :
 - l'appel d'une **méthode**,
 - le lancement d'une **exception**,
 - Sur une **classe** (le débogueur suspend l'exécution lorsqu'on accède au code de la classe ou lorsque la classe est enlevée de la mémoire)
- Cf la doc Oracle (en Anglais) :

http://docs.oracle.com/cd/E40938_01/doc.74/e40142/run_debug_japps.htm#BABIIDHH

Exercice

- Implémenter le code suivant et le corriger (on suppose que y est un champs de classe (non initialisé), alors que d,e,f sont des variables) ; puis déboguer et expliquer ce qui s'affiche pour les 3 appels proposés :

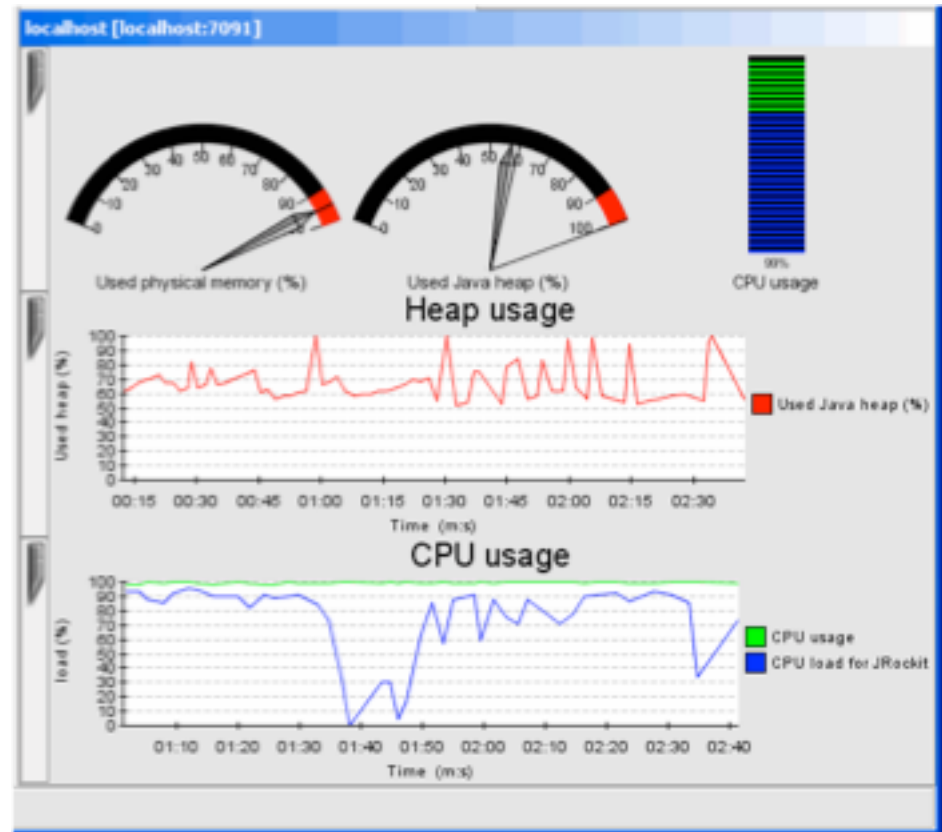
```
void myfunction ( int a , int b , int c ) {  
    if ( b < c ) {  
        d = 2*b ; f = 3*c ;  
        if ( c >= 0 ) {  
            y = f ; e = c ;  
            if ( y <= f ) {  
                a = f - e ;  
                if ( d < a )  
                    System.out.println( a ) ;  
            }  
        }  
    }  
}
```

```
myfunction(2467, 65900, 801) ;  
myfunction(2467, 549, 5904);  
myfunction(3, 16, 18) ;
```

*Ce code est-il
vraiment optimisé ?
Nous le verrons avec
la suite...*

Pour aller plus loin...

- Débugage de GUI
 - Débuguer les interfaces H/M est long et fastidieux
 - Récemment des outils d'aide au test exhaustif des composants graphiques sont apparues
- Avec Netbeans, cf cette vidéo
 - 5'20, en Anglais
 - <https://netbeans.org/kb/docs/java/debug-visual-screenshot.html>



Analyse de la performance

PROFILING

GESTION DE LA MÉMOIRE

Profiling, c'est quoi ?

Objectifs du **profiling** :

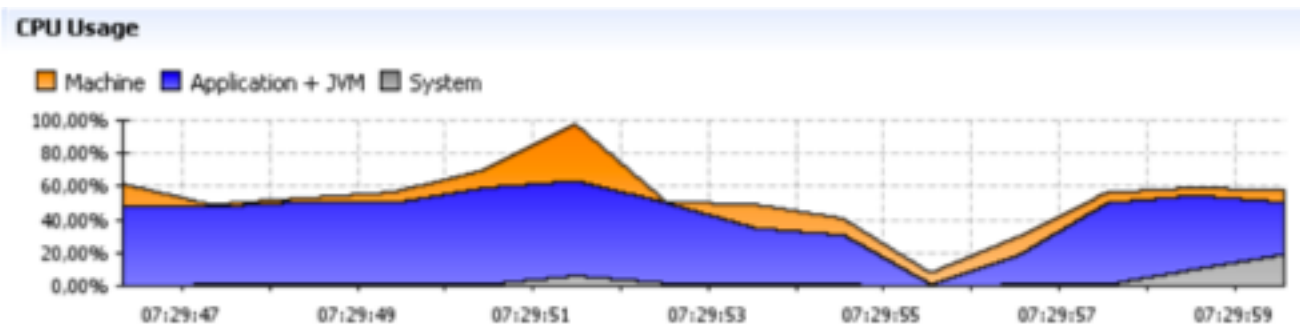
- Optimiser un temps de traitement
- Connaître le nombre d'appels (de passages) dans une fonction
- Détecter des fonctions non optimisées
- Quelle est la partie du code qui consomme le plus de temps d'exécution ? Et le plus de mémoire ?
- C'est donc l'analyse de la **performance d'exécution**

Exemple

- Aider à identifier les **goulots d'étranglement** dans le code
 - là où on ne les attend pas
 - Ou qui risquent de limiter la montée en charge de votre application
- Si une méthode xxx **représente par ex. plus de 10%** du temps total d'exécution, on peut chercher pourquoi
 - Par ex. en examinant le code source de xxx, on découvre un algorithme particulièrement inefficace qui doit être revu.

CPU / Mémoire

- Le profileur est capable de faire une analyse détaillée (aussi appelée *instrumentation*) de l'utilisation des **ressources processeur [CPU]** ainsi que de l'utilisation de la mémoire
- Par contre il ne peut pas faire les 2 au même moment !
 - En 2 temps



Pourquoi gérer la mémoire ?

- La mémoire est une ressource rare
 - Même si les matériels aujourd'hui sont tjrs plus dotés en mémoire, la prépondérance des mobiles croît de façon certaine
 - Toujours penser à développer des applications optimisant la mémoire
- Le développeur a le devoir de bien la gérer
- La mémoire dépend du système, i.e. de l'environnement d'exécution :
 - Ex. en C/C++ : il existe deux types de mémoires
 - Mémoire de la pile
 - Mémoire du tas

Pourquoi gérer la mémoire ?

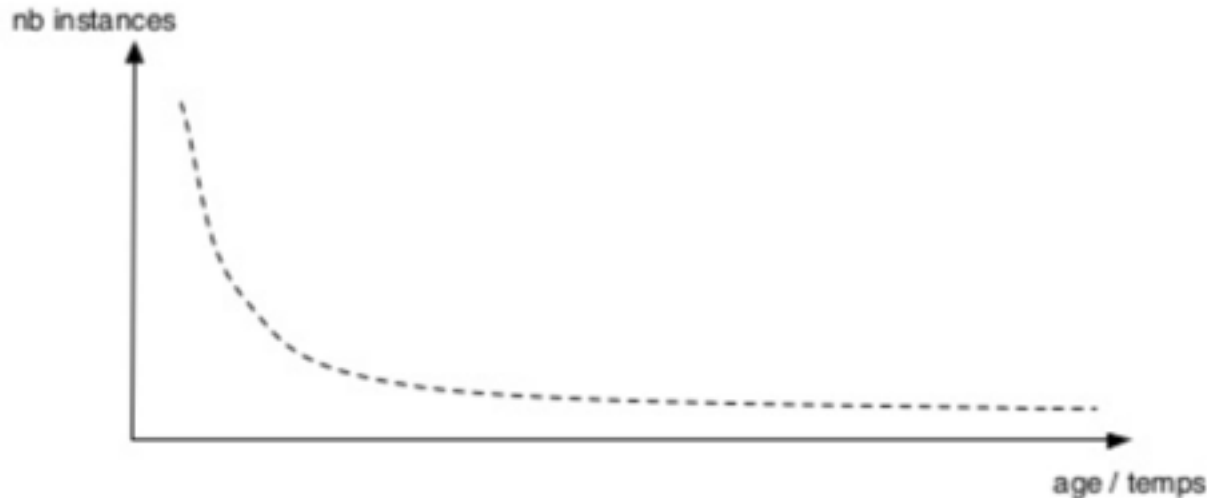
- En Java, la JVM (*Machine Virtuelle Java*) est configurée (optimisée) pour une certaine façon d'utiliser la mémoire
 - **Garbage collector (ramasse-miettes)**
 - Chaque JVM (ex. 32 ou 64 bits) implémente son propre ramasse-miettes en utilisant un ou plusieurs algorithmes
 - Les règles de gestion de la mémoire dans une JVM sont définies dans le JMM (Java Memory Model)
- L'activité du ramasse-miettes **peut dégrader fortement les performances** d'une application
 - lorsque l'utilisation de la mémoire ne correspond pas aux réglages par défaut de la JVM
 - Ex.: des objets gaspillant la mémoire (ex. : des ArrayList défini en static surdimensionnés ayant beaucoup d'éléments vides)

Principe du ramasse-miette (GC)

- Java a choisi de simplifier la vie des développeurs :
 - Il n'est pas possible d'allouer de la mémoire **explicitement** : c'est la création d'un nouvel objet avec l'opérateur *new* qui alloue la mémoire requise
 - La JVM dispose d'un ramasse-miettes qui se charge de **libérer la mémoire des objets inutilisés**
- Les objets *inutilisés* sont les objets qui **ne sont référencés** par **aucun** autre objet.

Cycle de vie des objets

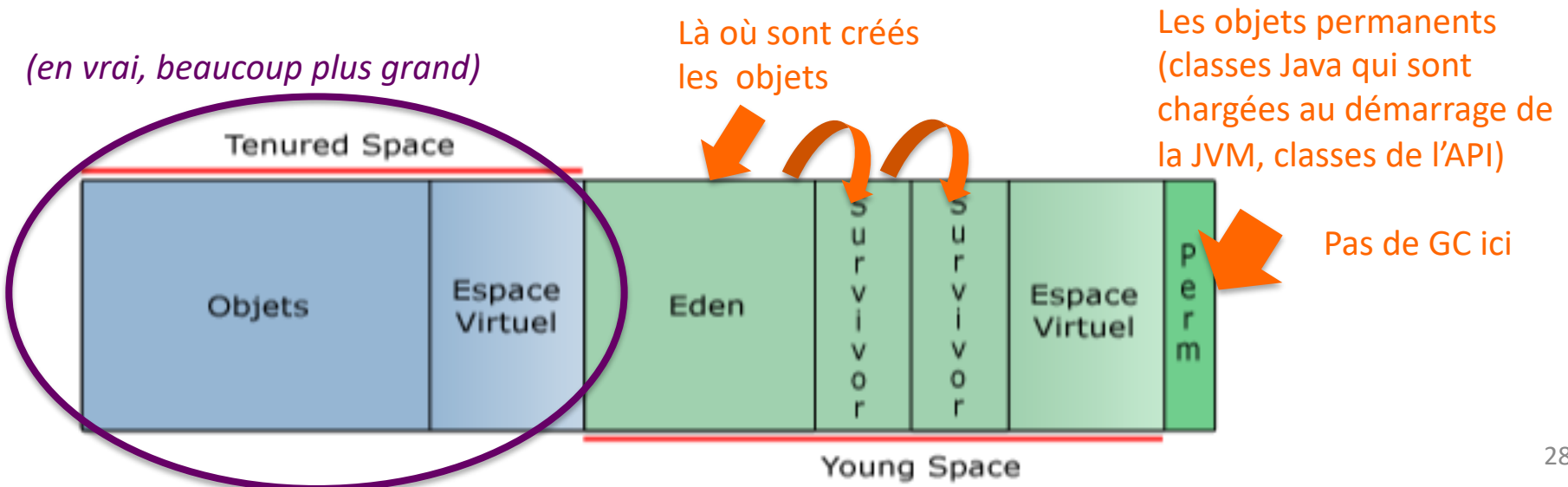
- Constat : la plupart des objets **meurent jeunes**



- Les concepteurs de la JVM chez SUN ont donc découpé la mémoire pour optimiser la recherche d'objets morts.
- Ainsi il y a une zone dédiée à la génération d'objets jeunes (*young space*) :
 - Cette zone est **nettoyée en priorité** et souvent

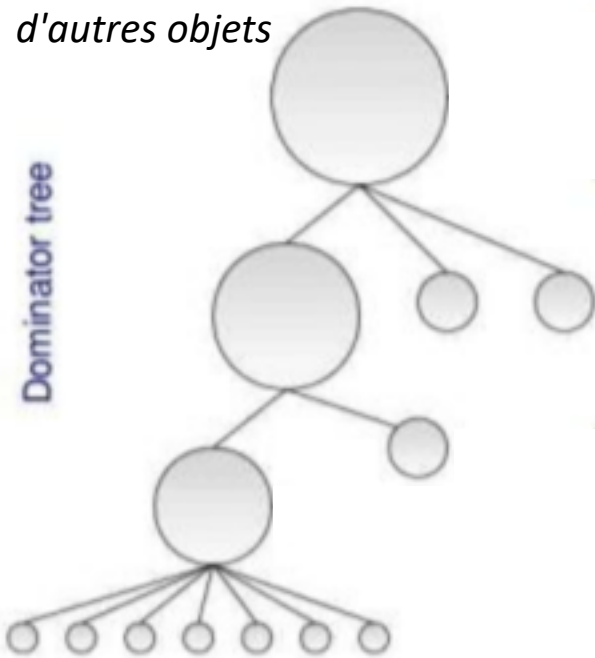
Cycle de vie des objets

- Les **survivants** (à une collecte) sont copiés dans une des sous-zones 'survivor'
- Les zones virtuelles sont des zones réservées, qui ne sont pas encore peuplées. Elles sont utilisées lorsque les zones survivor sont saturées.
- Le ramasse-miettes déclenche une collecte **mineure** lorsque la zone *young* est pleine (*eden, survivor*), une collecte **majeure** (plus coûteuse) quand le taux de remplissage de la zone *tenured* dépasse un seuil.



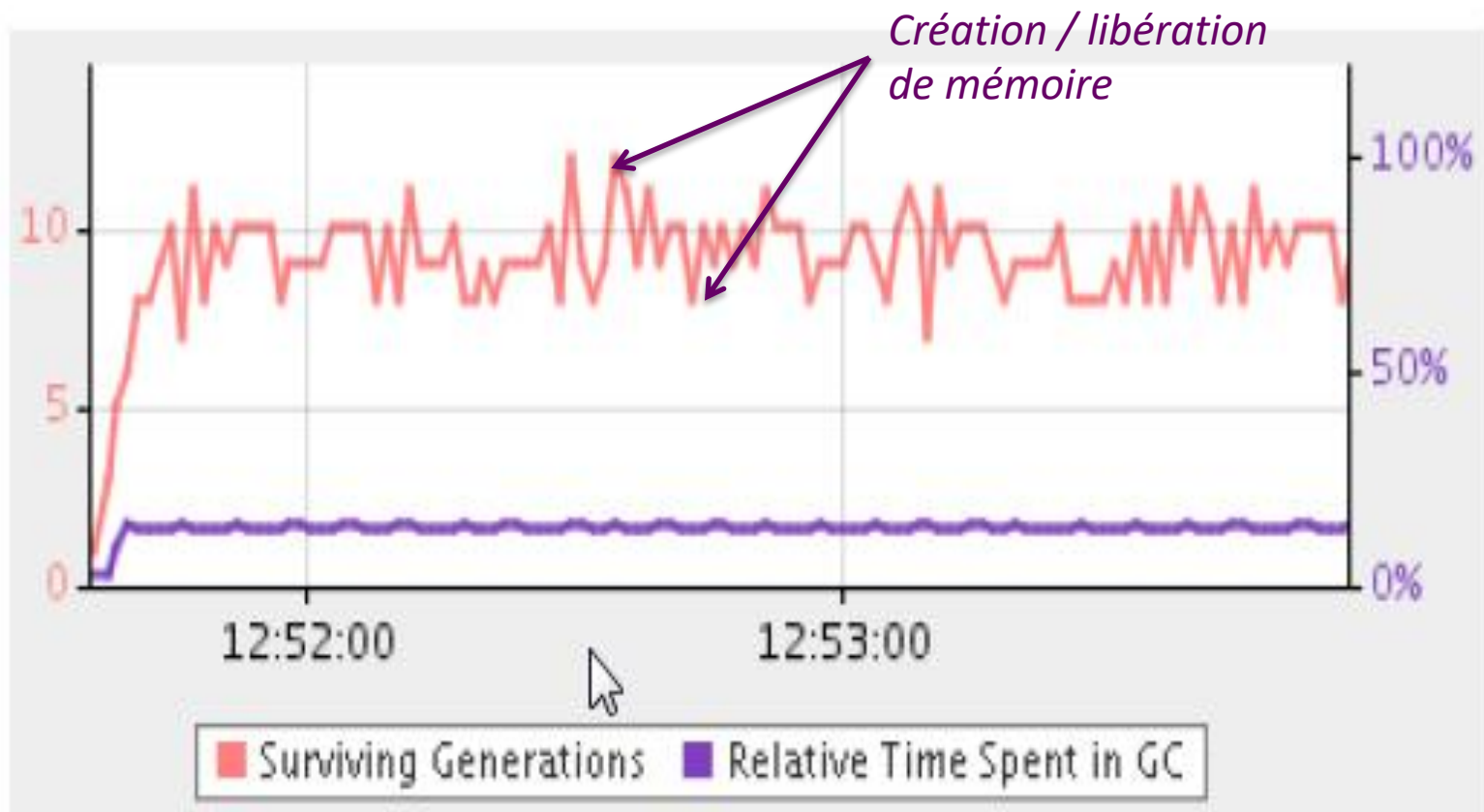
Pourquoi gérer la mémoire ?

Ici, un objet static garde des références vers une multitude d'autres objets

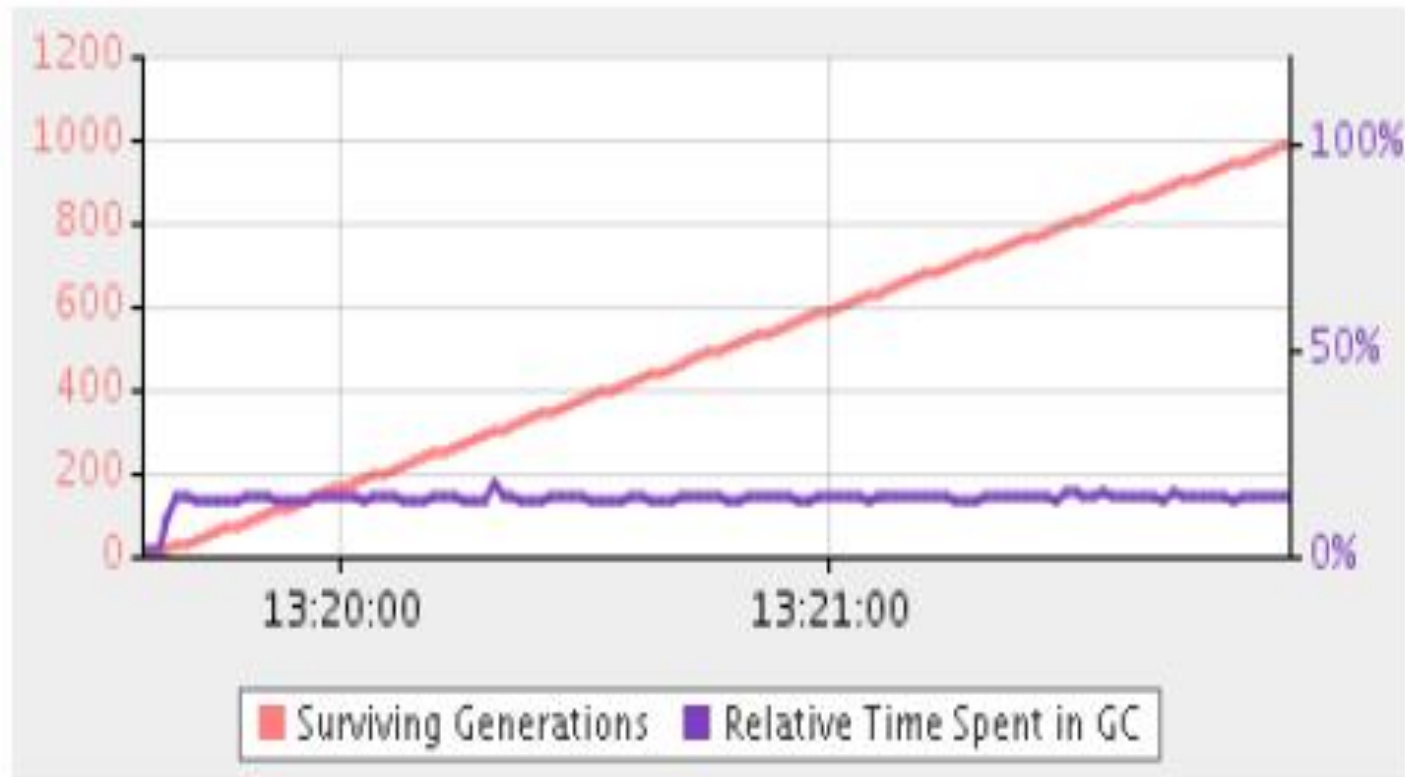


- Les **fuites mémoires** provoquent des ralentissements, et à terme des erreurs de segmentation
 - DEF une fuite de mémoire est une **occupation croissante et non contrôlée** de la mémoire pouvant mener à terme à une saturation mémoire, un « gel de l'application »
 - Ou une exception de type : **java.lang.OutOfMemoryError**
- On peut **ajuster** les paramètres de la mémoire (young / tenured)

- Dans une application à mémoire bien gérée, la courbe du nb de générations survivantes est en dents de scie :



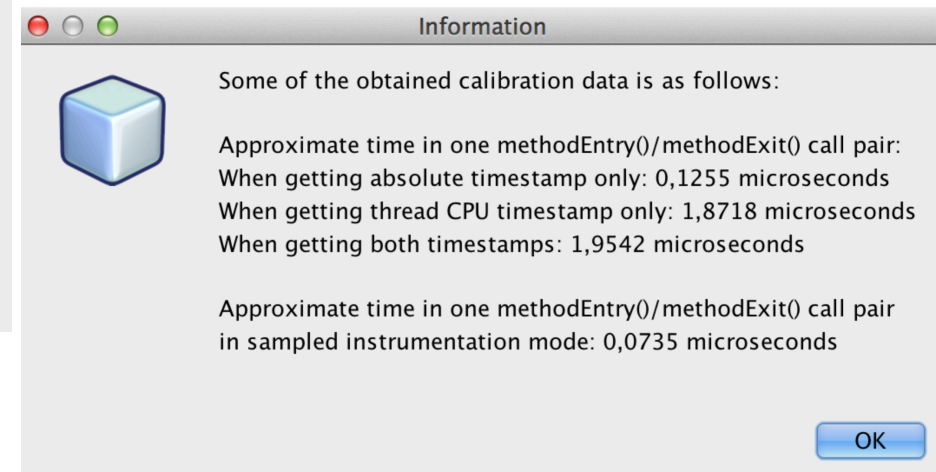
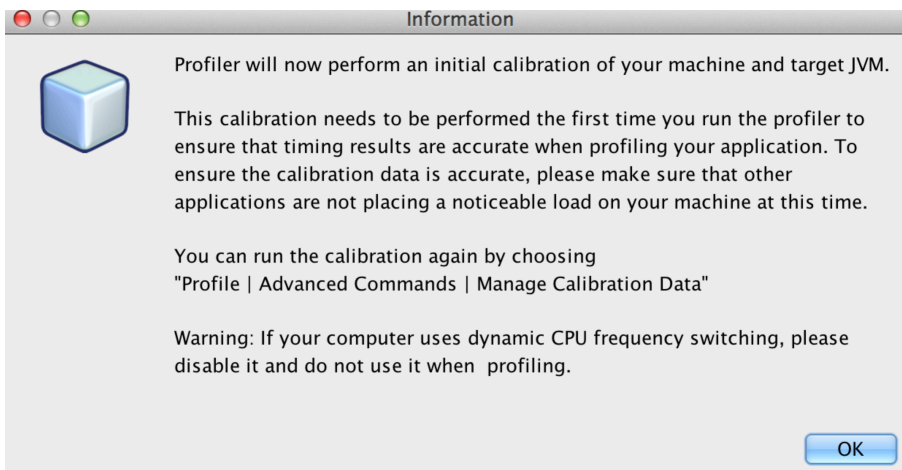
- En cas de fuite mémoire, le nombre d'objets qui survivent après chaque collecte ne cesse de croître :



Source : <http://schmitt.developpez.com/tutoriel/java/memoire/>

Outils mémoire / profiling sous IDE

- *La 1^{ère} fois qu'on lance le Profiler, il doit faire un calibrage de la JVM afin de prendre en compte la puissance de sa machine :*








Le Profiler de NetBeans

- *Menu Profile – Profile Main Project*

Configure and Start Profiling

Les analyses les plus intéressantes

1. Click the **Configure Session** button in toolbar and select the desired profiling mode:

	Telemetry	Monitor CPU and Memory usage, number of threads and loaded classes
	Methods	Profile method execution times and invocation counts, including call trees
	Objects	Profile size and count of allocated objects, including allocation paths
	Threads	Monitor thread states and times
	Locks	Collect lock contention statistics

2. Click the **Profile** button in toolbar once the session is configured to start profiling.
3. Use the Profile **dropdown arrow** to change profiling settings for the session.

Comparer des analyses

- On peut sauver ces graphiques (*save snapshot*) et les comparer lors de sessions de profiling ultérieures.
- Quand on enregistre un snapshot, NB l'enregistre dans un répertoire 'Profiler' sous Private, dans le répertoire du Projet
- *Regarder par ex.:*

<http://wiki.netbeans.org/ProfilerMethods>

Pour aller plus loin...

- Un exemple de profiling avec un site web :
 - <https://fr.netbeans.org/edi/articles/concours/nb-profiler-tutor.html#cpu>
- D'autres outils de profiling open source pour Java :
 - <http://java-source.net/open-source/profilers>

Je retiens...

- Pour le débogage :
 - Il est possible de suivre une exécution pas à pas avec un IDE
 - On peut faire varier les valeurs des variables en cours d'exécution
 - Je sais utiliser le débogueur de NetBeans
- Pour le *profiling* :
 - Je sais le définir ainsi que la *fuite de mémoire*
 - Ca sert à analyser l'utilisation du processeur et de la mémoire
 - J'ai compris le fonctionnement du GC pour la JVM
 - Je sais utiliser le profiler de NetBeans