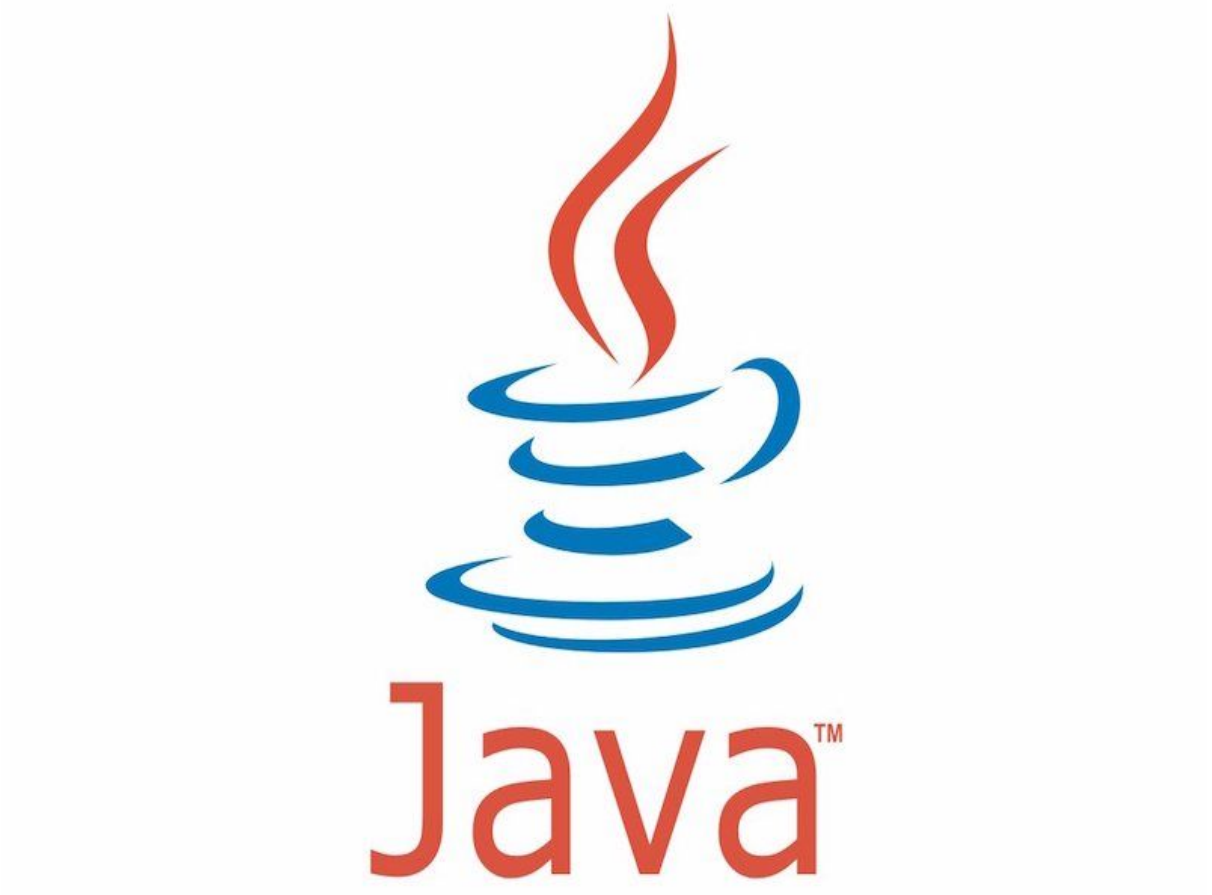
TP5

Les fourmis

Cognard Cédric et Van-Elsue Paul



Sommaire

1. Test initial

Avant d’essayer d’optimiser le programme, nous l’avons d'abord exécuté. Cela peut sembler anodin mais il serait malvenu d’essayer d’optimiser quelque chose si on ne sait pas à quoi ça correspond. On a ainsi une meilleure vue d’ensemble et parfois quelques pistes. De plus cela nous permet de voir si nos modifications ont altérées l’affichage ou non par la suite.

Ici nous avons utilisé les données “ants_default.html” puis dans un premier temps afin de savoir à quoi servait le programme et quel affichage il faisait, puis “ants_worst.html” afin de regarder les performances. Par chance, un compteur de FPS est intégré, nous avons vu que par défaut avant d’effectuer la moindre modification nous tournons à 8000 FPS.

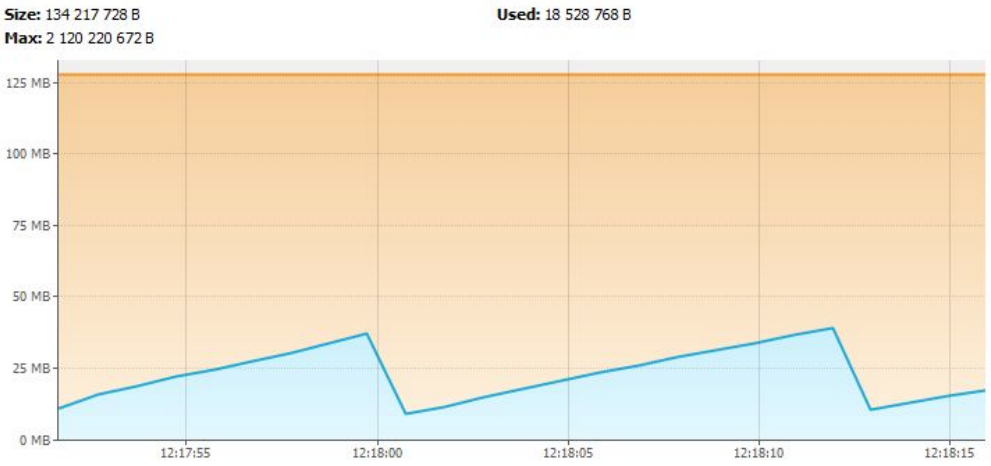
Nous voyons aussi que le programme est lent à lancer, il y a une bonne période de chargement au début. Si nous souhaitons faire une optimisation parfaite, l’idéal serait d’optimiser le nombre de FPS mais aussi le temps de chargement au début. La réalité est un peu plus compliqué car il faut parfois faire un compromis entre les deux, comme par exemple mettre des données en cache accélérant l’exécution mais ralentissant le chargement.

Dans notre cas, nous n’avons optimisé que l’exécution, face au temps que nous avions et prenant en compte que nous n’avons pas trouvé directement des solutions ainsi que le fait qu’elles n’ont pas du tout fonctionné du premier coup, nous n’avons pas pu aller plus loin.

2. Recherche des points d'amélioration

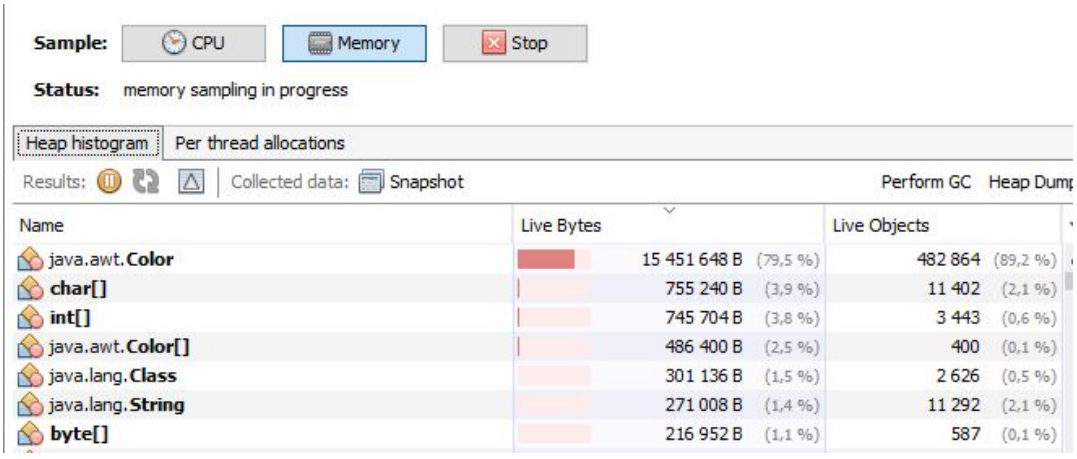
Une fois le programme lancé, nous cherchons à voir qu'est-ce qui le ralentis. Quels sont les bouts de code qui prennent le plus de temps à être exécutés.

Tout d'abord nous avons visualisé l'exécution de ce programme sur **visualvm,** on peut constater que le *garbage collector* fait bien son travail en libérant la mémoire à intervalle régulier. Sur le screenshot ci-dessous, la mémoire, en bleu, augmente car la programme l'utilise pour ses variable, puis chute régulièrement lorsque le *garbage collector* effectue sa routine.



Analyse de la mémoire sous VisualVM.

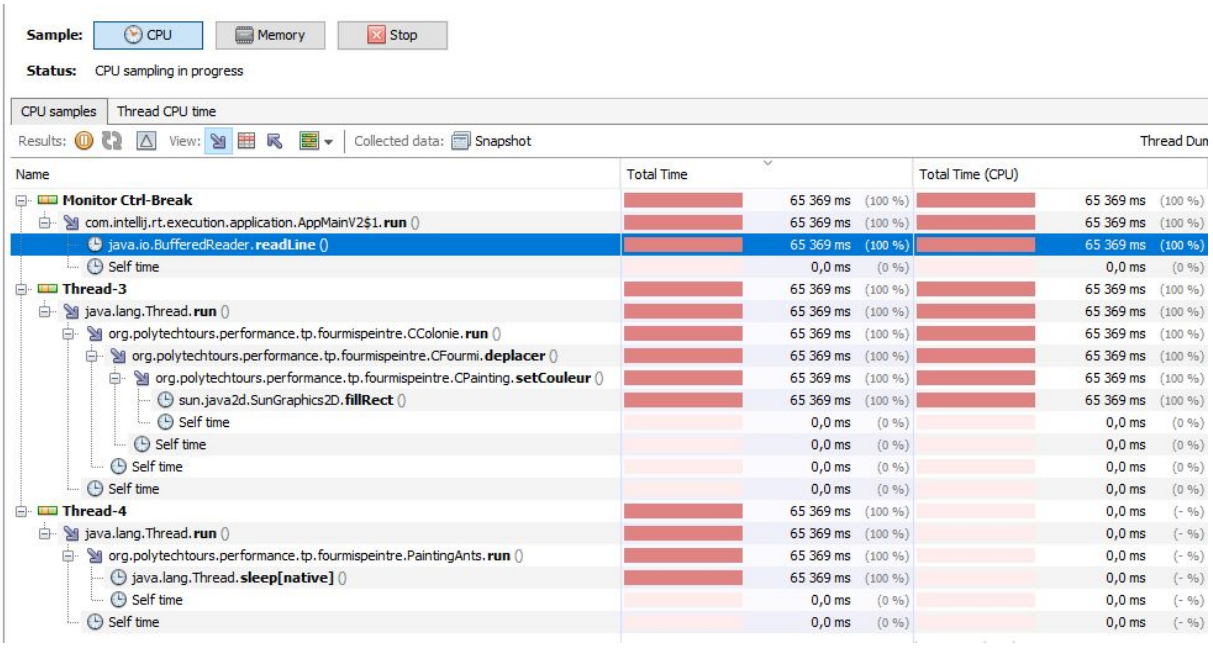
En allant dans l'onglet Memory on constate que la mémoire utilisée par le programme est composé à presque 80% d'objets de type **Color**. On pourrait donc peut être trouver un moyen de réduire le nombre d'objets allouées de ce type, c'est clairement le plus gros point faible en terme de mémoire.



Analyse de l'allocation mémoire.

De plus, si nous allons explorer l'onglet CPU nous pouvons voir dans quoi nous passons le plus de temps. On retrouve la fonction **setCouleur()** qui utilise des objets de type **Color**, c'est cette dernière qui consomme le plus le CPU. Donc nous allons donc essayer d'optimiser cette partie.

On peut effectivement voir l'analyse sur le screenshot ci-dessous, attention la ligne surlignée n'est pas la bonne.



Analyse de l'utilisation du CPU.

3. Analyse du code

Nous savons quelle est la partie du programme qui, une fois optimisée, apportera le plus de gain de performances (attention ce n'est peut être pas la seule à pouvoir être optimisée bien sûr).

Tout d'abord nous avons commencé par lire le code associé. Nous ne sommes pas à l'abri d'une grosse erreur de programmation toute simple qui ralentirait le programme, et puis de toute manière il faudra l'optimiser alors il vaut mieux le comprendre.

Nous avons remarqué que la fonction effectuait trois fois le même code selon on condition sur le paramètre **pTaille**. Ces trois morceaux de code n'ont que très peu de différence, nous avons donc choisi de factoriser tout ça et d'éviter de passer par la condition. Évidemment cela n'a pas changé les performances mais nous avons quand même testé au cas ou, mais l'avantage certain est que le code était plus lisible.

Une fois ce premier travail effectué, on continue l'analyse. Nous avons vu précédemment que c'étaient les objets de type Color qui prenaient toute la place en mémoire, et ici la

fonction cherche à en créer un et à le renvoyer. Si on analyse en réfléchissant, qu'est-ce qui ce passe ? Le code passe beaucoup de temps à faire des calculs mais surtout à créer cet objet qui est très lourd. Cela surcharge le CPU. Mais cet objet créé n'est pas supprimé tant que le *Garbage Collector* ne fait pas sa routine, cela surcharge donc la mémoire. Une fois que le *Garbage Collector* travaille, la mémoire est libérée et on recommence : cela explique parfaitement les screenshots que nous avons pu observer plus tôt !

Nous avons eu plusieurs idées pour régler ce problème. La première serait de créer son propre objet, similaire à **Color**, optimisé pour n'avoir que le strict nécessaire pour notre code. C'est probablement la meilleure manière de faire mais aussi la plus complexe, surtout qu'il faudrait modifier absolument tout le reste du code en conséquent. Cette solution a rapidement été écartée car trop longue à mettre en place, nous n'avions pas le temps.

Une autre possibilité était d'utiliser le cache : au lieu de recréer un objet **Color** à chaque fois, l'idée est de le garder en mémoire et d'aller le chercher.

4. Optimisation

Nous avons donc vu que pour optimiser le programme, la meilleure idée que nous aillons eu est d'utiliser le cache.

Utiliser le cache demande de créer les objets au début puis de les sauvegarder. C'est très lourd surtout au chargement, de plus si cela consomme trop de mémoire alors sur une machine moins puissante nous pourrions avoir des problèmes. Nous avons opté pour une création de cache "à la volée".

Chaque fois qu'un objet de type Color est appelé, si il n'existe pas il est créé puis ajouté au cache, si il existe nous n'avons qu'à le récupérer dans la mémoire. Pour savoir si un objet existe ou non nous avons tout simplement comparé les paramètres donnés pour sa création, autrement dit les paramètre de la méthode.

L'idée semble bonne ! Mais une fois implémentée la réalité en est toute autre. Nous sommes passés de 8000 FPS à seulement 500, soit un programme 16 fois moins performant, quel échec !

Mais nous n'avons pas laissé tombé, nous avons pensé que si nous sauvegardons absolument toutes les couleurs alors la mémoire utilisée serait vraiment trop importante et c'était pénalisant. C'était effectivement le cas. Pour palier à ce manque, nous avons choisi non pas de tout garder en mémoire mais seulement les couleurs les plus utilisée.

Cette idée fut la bonne, nous avions donc un tableau d'une certaine taille et un seuil indiquant que la couleur était beaucoup utilisée. Pour matérialiser tout ça, nous sommes passés par une nouvelle classe, la classe **HistoryColor**. Elle comprend 4 attributs : la taille maximale du tableau de couleurs, le minimum d'utilisation nécessaire pour rester dans le

tableau, le tableau des couleurs et enfin un dernier tableau indiquant sur chaque couleur son taux d'utilisation.

```
public static final int MAX_SIZE = 100000;
public static final int MIN_USE = 100;
private ArrayList<PaintingColor> colors;
private int usedRate[];
```

Et voilà ! Nous étions prêts à tester notre code.

5. Résultats

Il est temps de voir les résultats obtenus. L'amélioration apportée, nous exécutons notre code et… non, nous sommes toujours trop bas en terme de FPS.

Mais nous avions deux attributs pour régler la taille du tableau et le minimum d'utilisation nécessaire pour en faire parti ! Nous avons donc cherché à optimiser ces tailles cette fois, et effectivement nous avons eu une nette amélioration : en optimisant **MAX_SIZE**, nous sommes passés de 500 à 2500 FPS, c'est 5 fois mieux mais cela reste en dessous des performances de base.

Si on cherche à améliorer **MIN_USE**, cette fois nous avons une vraie amélioration ! Nous sommes enfin passés à 14000 FPS !! Ce fut un soulagement incroyable car c'est arrivé dans les 5 dernières minutes de la dernière séance, nous avions vraiment peur de n'avoir absolument rien apporté.

Désormais le code est 1.75 fois plus rapide à être exécuté. Ce n'est pas une amélioration incroyable mais c'en est une. Nous n'avions par contre malheureusement plus assez de temps pour chercher une autre grosse optimisation à faire et nous nous sommes donc arrêtés là, mais pour que cela soit parfait il y a encore beaucoup de chose à faire.

6. Conclusion

Ce projet était très intéressant. Au début nous étions vraiment perdus face à la complexité du programme mais aussi à cause de son code peu lisible. Lors des premières séances nous n'avions peu de solutions, peu de résultats et donc peu de motivation. Par contre une fois que nous avions eu notre idée du cache et que nous avons commencé, alors tout est devenu bien plus attrayant.

Nous sommes vraiment contents d'avoir finalement obtenu des résultats, car c'était mal parti.

Merci pour les cours !

Annexe : la classe HistoryColor

```java
package org.polytechtours.performance.tp.fourmispeintre.colors;

import java.util.ArrayList;


public class HistoryColor {
    public static final int MAX_SIZE = 100000;
    public static final int MIN_USE = 100;
    private ArrayList<PaintingColor> colors;
    private int usedRate[];

    public HistoryColor() {
        usedRate = new int[MAX_SIZE];
        colors = new ArrayList<>();
    }

    public boolean contains(PaintingColor testedColor) {
        return colors.contains(testedColor);
    }

    public int size() {
        return colors.size();
    }

    public int indexOf(PaintingColor testedColor) {
        return colors.indexOf(testedColor);
    }

    public PaintingColor get(int indexOf) {
        return colors.get(indexOf);
    }

    public void add(PaintingColor paintingColor) {
        if(size() >= MAX_SIZE)
            if(retract()) {
                colors.add(paintingColor);
                usedRate[colors.size() - 1] = 1;
            }
    }

    public boolean retract() {
        int new_use_rate[] = new int[MAX_SIZE];
        int min = 99999, position = -1, i;
```

```
        for(i = 0; i < usedRate.length; i++)
            if(usedRate[i] < min) {
                min = usedRate[i];
                position = i;
            }

        if(min >= MIN_USE) {
            colors.remove(position);

            for(i = 0; i < usedRate.length; i++)
                if(i != position)
                    new_use_rate[i] = usedRate[i];
            usedRate = new_use_rate;

            return true;
        }

        return false;
    }

    public PaintingColor incrementAndGet(int indexOf) {
        usedRate[indexOf]++;
        return get(indexOf);
    }
}
```