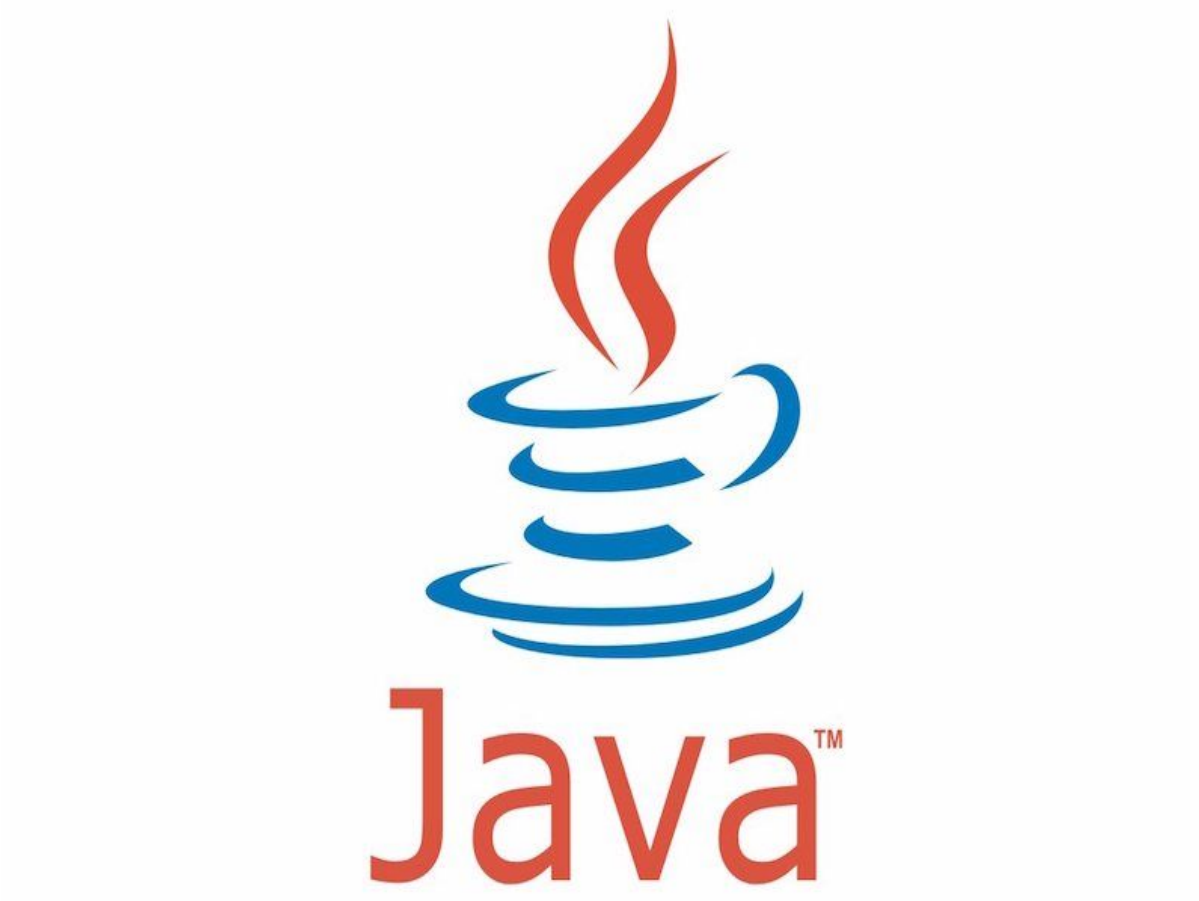
TP4

Exercices d’optimisation

Cognard Cédric et Van-Elsue Paul



Sommaire

Mode Opératoire

Dans tous nos benchmarks nous avons pris le code des tests unitaires fournis que nous avons placé dans des **Blackhole**. Ainsi au lieu de tester les méthodes fait croire au Benchmark qu'il est utilisé mais on en fait rien, permettant de tester les performances.

```
public class BenchmarkExercice2 {
  @Benchmark @Fork(1)
  public void testExercice2(Blackhole blackhole) {
     blackhole.consume(Exercice2.fibonacci(43));
  }
}
```

Pour être sûr de pouvoir revenir en arrière en cas de problème nous avons choisi de faire une nouvelle version de la fonction de base pour chaque modification apportée. De cette façon nous avons pu essayer différentes modifications dans le code et constater si nous avions amélioré ou non les performances.

La méthode consistait à commencer par nommer la fonction fournie en v01 et à incrémenter ce compteur de version à chaque modification. Dès qu'on voulait essayer quelque chose on reprenait la version la plus performante que nous avions.

Exercice 1 : Multiplication de deux matrices

Ce programme réalise le produit d'un matrice d'entiers avec la matrice *MATRIX_A* (cf. ci-dessous), pour cela on utilise 3 boucles **for** imbriquées.

```
MATRIX_A = {
        {1/42f,1/42f,2/42f,2/42f,2/42f,1/42f,1/42f},
        {1/42f,2/42f,3/42f,4/42f,3/42f,2/42f,1/42f},
        {2/42f,3/42f,4/42f,5/42f,4/42f,3/42f,2/42f},
        {2/42f,4/42f,5/42f,8/42f,5/42f,4/42f,2/42f},
        {2/42f,3/42f,4/42f,5/42f,4/42f,3/42f,2/42f},
        {1/42f,2/42f,3/42f,4/42f,3/42f,2/42f,1/42f},
        {1/42f,1/42f,2/42f,2/42f,2/42f,1/42f,1/42f}
```

Quand on fait un benchmark sur le code que nous avons par défaut nous obtenons le résultat suivant :

```
Benchmark                         Mode  Cnt       Score        Error  Units
BenchmarkExercice1.testMultiply  thrpt    5  164524.817 ± 179719.151  ops/s
```

Si nous changeons les appels Float et Integer ainsi que l'incrémentation dans les boucles et que nous sortons la déclaration du current value de la boucle nous améliorons déjà pas mal les performances (environ x6) :

```
Benchmark                          Mode  Cnt       Score        Error  Units
BenchmarkExercice1.testMultiply  thrpt    5  925811.773 ± 586380.859  ops/s
```

Si on enlève le final sur le résultat nous pouvons encore améliorer notre score, même si l'amélioration est tellement faible qu'elle peut être négligeable.

```
Benchmark                          Mode  Cnt       Score        Error  Units
BenchmarkExercice1.testMultiply  thrpt    5  994559.925 ± 112902.175  ops/s
```

Dans cette troisième version du programme nous utilisons la symétrie de la matrice.
C'est à dire que nous réduisons le nombre de tours dans la boucle mais pas les calculs à l'intérieur et au final nous perdons 10% de performance car nous utilisons 2x plus de variables.

```
Benchmark                          Mode  Cnt       Score        Error  Units
BenchmarkExercice1.testMultiply  thrpt    5  881173.324 ± 171307.359  ops/s
```

Notre score est donc descendu donc nous allons rester sur la deuxième version qui reste la plus performante. Nous avons donc un score de 994559.925 opérations par minutes ce qui est 6 fois plus rapide que le programme d'origine.

Il existe peut-être de meilleurs méthodes d'optimisation mais nous ne les avons pas trouvé. Dans la vraie vie il faudrait voir jusqu'à quel point nous avons besoin d'optimiser avant de s'arrêter.

Exercice 2 : Calcul de la suite de Fibonacci.

Dans le code de base le calcul de la suite ce fait de manière récursive. Or nous savons que c'est toujours mieux d'éviter le récursif car étant connu comme très peu performant. Nous allons donc comparer les performances avec un calcul itératif.

Récursif :

```
Benchmark                            Mode  Cnt  Score   Error  Units
BenchmarkExercice2.testExercice2  thrpt    5  0.332 ± 0.001  ops/s
```

Itératif:

```
Benchmark                            Mode  Cnt          Score         Error  Units
BenchmarkExercice2.testExercice2    thrpt    5  347664236.546 ± 2528973.846  ops/s
```

On constate que le score du benchmark en itératif est totalement incomparable au code récursif tellement les performances sont améliorées ! Nous gardons donc la solution itérative et nous nous contenterons de cette version bien plus efficace.

Exercice 4 : Condenser un nombre indéfini de tableaux de bytes dans un seul et unique tableau.

Dans cet exercice nous allons récupérer plusieurs tableaux de bytes et les fusionner dans un seul et unique tableau.

La version d'origine du code réalise le score suivant au benchmark:

```
Benchmark                            Mode  Cnt        Score       Error  Units
BenchmarkExercice4.testExercice4    thrpt    5  1322168.071 ± 77934.849  ops/s
```

Si nous remplaçons `Integer` et `Byte` par les types primitifs nous améliorons déjà les performances mais de très peu.

```
Benchmark                            Mode  Cnt        Score       Error  Units
BenchmarkExercice4.testExercice4    thrpt    5  1413873.705 ± 28725.205  ops/s
```

On peut encore retirer la mention **`final`** en paramètre de notre fonction, comme dans le premier exercice, cela permet d'améliorer encore légèrement notre score. C'est pas grand chose mais c'est toujours ça de pris.

```
Benchmark                            Mode  Cnt        Score       Error  Units
BenchmarkExercice4.testExercice4    thrpt    5  1428727.462 ± 10909.671  ops/s
```

Pour avoir une véritable amélioration des performances nous avons remplacé la liste utilisée par un tableau de bytes, en faisant ce choix nous obtenons :

```
Benchmark                            Mode  Cnt        Score       Error  Units
BenchmarkExercice4.testExercice4    thrpt    5  2290046.138 ± 37795.286  ops/s
```

Nous avons au final rendu l'exécution du programme 1.6 fois plus rapide qu'au début.

Exercice 5 : appeller une méthode get sur un objet passé en paramètre

Sans modifications, cet exemple réalise le score suivant dans notre benchmark :

```
Benchmark                          Mode  Cnt        Score       Error  Units
BenchmarkExercice5.testExercice5  thrpt    5  6345275.017 ± 70272.743  ops/s
```

Sauf que l'on remarque que la méthode donnée fait appel à un get d'un objet précis, il est donc plus efficace de directement définir une méthode get dans ce dernier. C'est ce que nous avons fait et nous pouvons constater que le résultat est vraiment meilleur :

```java
public static String getName(final Guy guy) {
  return guy.getName();
}

/**
* Méthode fournie dans le TP.
*/
public static String getName_v01(final Guy guy) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
  return (String) guy.getClass().getMethod("getName").invoke(guy);
}
```

```
Benchmark                          Mode  Cnt          Score         Error  Units
BenchmarkExercice5.testExercice5  thrpt    5  112431800.827 ± 1044398.305  ops/s
```

Nous allons donc garder cette manière de procéder qui est à peu près 17 à 18 fois plus rapide que l'originale !

Conclusion

Tout d'abord nous pouvons voir que nous n'avons pas réussi à améliorer l'exercice 3. Par contre ce TP nous aura bien permis d'appréhender les méthodes d'optimisation. Même si au final chaque programme a ses spécificités nous saurons désormais par où commencer.

C'était sympa, larloularla.