COMPTE RENDU JAVA PERFORMANCE

Compte rendu des benchmarks fait avec JMH sur les déclarations de performanceClaims.pdf

YANN MARTIN D'ESCRIENNE N°ETUDIANT 21713752



TAB	TABLE DES MATIERES						
1.1	Standard code optimizations, Claim 1	2					
1.1	Standard code optimizations, Claim 2						
1.1	Standard code optimizations, Claim 3	4					
1.1	Standard code optimizations, Claim 4	5					
1.1	Standard code optimizations, Claim 5	6					
1.1	Standard code optimizations, Claim 6	7					
1.2	Fields and variables	8					
1.3	String manipulation	9					
1.4	Storing tables of constants in arrays	10					
1.5	Methods, Claim 1	11					
1.5	Methods, Claim 2	12					
1.6	Sorting and searching, Claim 1	13					
1.6	Sorting and searching, Claim 2	14					
1.7	Exceptions	15					
1.8	Collection classes, Claim 1	16					
1.8	Collection classes, Claim 2	1 <i>7</i>					
1.8	Collection classes, Claim 3	18					
1.8	Collection classes, Claim 4	19					
1.9	Input and output	20					
1.11	Bulk array operations	21					
1 1 2	Deflection	22					



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 1

 Move loop-invariant computations out of loops. For example, avoid repeatedly computing the loop bound in a for-loop, like this:

```
for (int i=0; i<size()*2; i++) { ... }
```

Instead, compute the loop bound only once and bind it to a local variable, like this:

```
for (int i=0, stop=size()*2; i<stop; i++) { ... }
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 5,500 \pm 0,079 ms/op MyBenchmark.test2 avgt 5 5,489 \pm 0,030 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

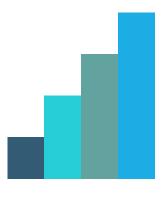
Test1: for normal.

Test2: for optimisé.

Test3: Pas de test.

CONCLUSION:

Pas de différence entre les deux boucles for.



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 2

· Do not compute the same subexpression twice:

```
if (birds.elementAt(i).isGrower()) ...
if (birds.elementAt(i).isPullet()) ...
```

Instead, compute the subexpression once, bind the result to a variable, and reuse it:

```
Bird bird = birds.elementAt(i);
if (bird.isGrower()) ...
if (bird.isPullet()) ...
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 32,323 \pm 4,073 ms/op MyBenchmark.test2 avgt 5 31,224 \pm 6,087 ms/op MyBenchmark.test3 avgt 5 \approx 10^{-6} ms/op
```

Test1: if normal.

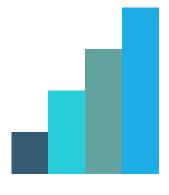
Test2: if optimisé.

Test3: Pas de test.

CONCLUSION:

Pas de différence entre les deux if (j'ai utilisé une ArrayList). Notez qu'avec une LinkedList cette optimisation à plus d'impact avec un facteur \sim 2:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 12,433 \pm 1,422 ms/op MyBenchmark.test2 avgt 5 6,402 \pm 0,428 ms/op MyBenchmark.test3 avgt 5 \approx 10^{-6} ms/op
```



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 3

Every array access requires an index check, so it is worth-while to reduce the number of array
accesses. Moreover, usually the Java compiler cannot automatically optimize indexing into multidimensional arrays. For instance, every iteration of the inner (j) loop below recomputes the
indexing rowsum[i] as well as the indexing arr[i] into the first dimension of arr:

```
double[] rowsum = new double[n];
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
    rowsum[i] += arr[i][j];</pre>
```

Instead, compute these indexings only once for each iteration of the outer loop:

```
double[] rowsum = new double[n];
for (int i=0; i<n; i++) {
  double[] arri = arr[i];
  double sum = 0.0;
  for (int j=0; j<m; j++)
    sum += arri[j];
  rowsum[i] = sum;
}</pre>
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 26,411 \pm 5,339 ms/op MyBenchmark.test2 avgt 5 25,753 \pm 2,164 ms/op MyBenchmark.test3 avgt 5 \approx 10^{-6} ms/op
```

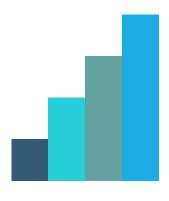
Test1: Somme des matrices normal.

Test2: Somme de la matrice optimisée.

Test3: Pas de test.

CONCLUSION:

Pas de différence entre les deux sommes de matrice.



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 4

- Declare constant fields as final static so that the compiler can inline them and precompute constant expressions.
- Declare constant variables as final so that the compiler can inline them and precompute constant expressions.

RESULTATS:

```
Benchmark
                  Mode
                        Cnt Score
                                     Error
                                            Units
MyBenchmark.test1
                          5 4,609 ± 0,368
                  avgt
                                            ms/op
MyBenchmark.test2
                          5 4,695 ± 0,290
                                            ms/op
                  avgt
MyBenchmark.test3
                  avgt
                          5 4,660 ± 0,425
                                            ms/op
```

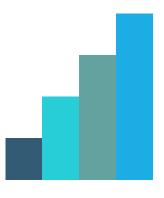
Test1: Récupération du champ normal.

Test2: Récupération du champ static.

Test3: Récupération du champ static final.

CONCLUSION:

Pas de différence majeure entre la récupération de constantes.



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 5

• Replace a long if-else-if chain by a switch if possible; this is much faster.

RÉSULTATS:

```
Benchmark
                   Mode
                         Cnt
                               Score
                                        Error
                                               Units
MyBenchmark.test1 avgt
                              19,541 ± 0,476
                                               ms/op
MyBenchmark.test2 avgt
                              14,257 ±
                                        0,258
                                               ms/op
MyBenchmark.test3 avgt
                              \approx 10^{-6}
                                               ms/op
```

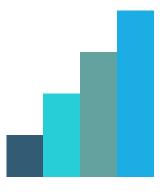
Test1: If-else.

Test2: Switch.

Test3: Pas de test.

CONCLUSION:

On constate que le switch est plus rapide que le if-else (facteur ~ 1.3).



1.1 STANDARD CODE OPTIMIZATIONS, CLAIM 6

 Nothing (except obscurity) is achieved by using 'clever' C idioms such as performing the entire computation of a while-loop in the loop condition:

```
int year = 0;
double sum = 200.0;
double[] balance = new double[100];
while ((balance[year++] = sum *= 1.05) < 1000.0);</pre>
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 0,323 \pm 0,023 ms/op MyBenchmark.test2 avgt 5 0,324 \pm 0,018 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

Test1: while classique.

Test2: while façon C.

Test3: Pas de test.

CONCLUSION:

Pas de différence entre les deux while.



1.2 FIELDS AND VARIABLES

- Access to local variables and parameters in a method is much faster than access to static or instance
 fields. For a field accessed in a loop, it may be worthwhile to copy the field's value to a local
 variable before the loop, and refer only to the local variable inside the loop.
- There is no runtime overhead for declaring variables inside nested blocks or loops in a method.
 It usually improves clarity to declare variables as locally as possible (with as small a scope as possible), and this may even help the compiler improve your program.

RESULTATS:

```
Benchmark

Mode Cnt Score Error Units

MyBenchmark.test1 avgt 5 0,161 ± 0,003 ms/op

MyBenchmark.test2 avgt 5 0,161 ± 0,002 ms/op

MyBenchmark.test3 avgt 5 0,161 ± 0,003 ms/op
```

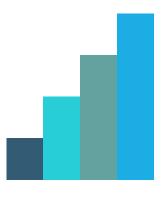
Test1: Somme avec variable statique.

Test2: Somme avec variable.

Test3: Somme avec variable locale.

CONCLUSION:

Pas de différence entre les sommes.



1.3 STRING MANIPULATION

 Do not build strings by repeated string concatenation. The loop below takes time quadratic in the number of iterations and most likely causes heap fragmentation as well (see Section 2):

```
String s = "";
for (int i=0; i<n; i++) {
    s += "#" + i;
}</pre>
```

Instead, use a StringBuilder object and its append method. This takes time linear in the number of iterations, and may be several orders of magnitude faster:

```
StringBuilder sbuf = new StringBuilder();
for (int i=0; i<n; i++) {
   sbuf.append("#").append(i);
}
String s = sbuf.toString();</pre>
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 5,314 \pm 0,427 ms/op MyBenchmark.test2 avgt 5 0,055 \pm 0,001 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

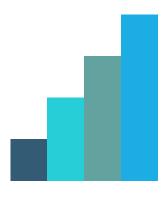
Test1: Concaténation classique.

Test2: Concaténation avec un String Builder.

Test3: Pas de test

CONCLUSION:

On constate une énorme optimisation de la concaténation avec un String Builder (facteur 100).



1.4 STORING TABLES OF CONSTANTS IN ARRAYS

Declaring an initialized array variable inside a method causes a new array to be allocated at every
execution of the method:

```
public static int monthdays(int y, int m) {
  int[] monthlengths =
      { 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31 };
  return m == 2 && leapyear(y) ? 29 : monthlengths[m-1];
}
```

Instead, an initialized array variable or similar table should be declared and allocated once and for all as a final static field in the enclosing class:

```
private final static int[] monthlengths =
    { 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31 };

public static int monthdays(int y, int m) {
    return m == 2 && leapyear(y) ? 29 : monthlengths[m-1];
}
```

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 1132,733 \pm 126,014 ms/op MyBenchmark.test2 avgt 5 64,663 \pm 1,403 ms/op MyBenchmark.test3 avgt 5 \approx 10^{-6} ms/op
```

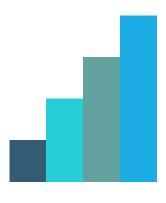
Test 1: Liste de constante interne à la méthode.

Test2: Liste de constante statique.

Test3: Pas de test

CONCLUSION:

On constate une énorme optimisation avec une liste de constante non recalculée (facteur \sim 17).



1.5 METHODS, CLAIM 1

- Declaring a method as private, final, or static makes calls to it faster. Of course, you should only do this when it makes sense in the application.
- For instance, often an accessor method such as getSize can reasonably be made final in a class, when there would be no point in overriding it in a subclass:

```
class Foo {
  private int size;
  ...
  public final int getSize() {
    return size;
  }
}
```

This can make a call o.getSize() just as fast as a direct access to a public field o.size. There need not be any performance penalty for proper encapsulation (making fields private).

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units
MyBenchmark.test1 avgt 5 13,827 ± 1,063 ms/op
MyBenchmark.test2 avgt 5 13,370 ± 0,252 ms/op
MyBenchmark.test3 avgt 5 13,352 ± 0,174 ms/op
```

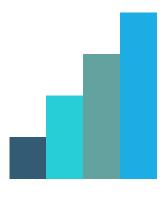
Test1: get classique.

Test2: get final.

Test3: récupération du champ directement.

CONCLUSION:

On ne constate qu'une petite différence entre la méthode de get classique et la méthode de get final. Mais ce n'est pas assez notable.



1.5 METHODS, CLAIM 2

 Virtual method calls (to instance methods) are fast and should be used instead of instanceof tests and casts.

RESULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 3,011 \pm 1,572 ms/op MyBenchmark.test2 avgt 5 3,022 \pm 1,410 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

Test1: appel de la méthode après instance of et cast.

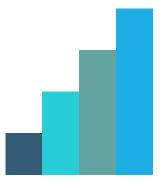
Test2: appel de la méthode virtuelle.

Test3: Pas de test.

CONCLUSION:

On ne constate pas de différence entre l'appel d'une méthode après instance of puis cast et une méthode virtuelle. Ce qui est logique car il est dit plus bas dans le document :

For example, a test (x instanceof C) conditionally followed by a cast (C) x may be optimized by a JVM so that at most one test is performed. Hence it is not worth the trouble to rewrite your program to avoid either the instanceof test or the cast.



1.6 SORTING AND SEARCHING, CLAIM 1

- Never use selection sort, bubblesort or insertion sort, except on very short arrays or lists. Use heapsort (for arrays) or mergesort (for doubly linked lists) or quicksort (for arrays; but you must make a good choice of pivot element).
- Even better, use the built-in sorting routines, which are guaranteed to be fast: $O(n \log(n))$ time for n elements, and sometimes faster if the data are nearly sorted already:

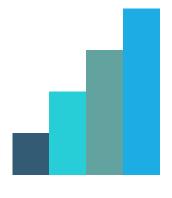
1

RÉSULTATS:

Mode	Cnt	Score	Error	Units
avgt	5	15,709 ±	0,309	ms/op
avgt	5	0,391 ±	0,007	ms/op
avgt	5	1,613 ±	0,112	ms/op
avgt	5	0,241 ±	0,003	ms/op
avgt	5	0,407 ±	0,036	ms/op
avgt	5	0,267 ±	0,004	ms/op
avgt	5	4,602 ±	0,057	ms/op
avgt	5	0,058 ±	0,001	ms/op
avgt	5	0,013 ±	0,001	ms/op
avgt	5	0,008 ±	0,001	ms/op
avgt	5	0,008 ±	0,001	ms/op
avgt	5	0,014 ±	0,001	ms/op
avgt	5	0,010 ±	0,001	ms/op
avgt	5	0,023 ±	0,001	ms/op
	avgt avgt avgt avgt avgt avgt avgt avgt	avgt 5	avgt 5 15,709 ± avgt 5 0,391 ± avgt 5 1,613 ± avgt 5 0,241 ± avgt 5 0,267 ± avgt 5 0,058 ± avgt 5 0,008 ± avgt 5 0,008 ± avgt 5 0,014 ± avgt 5 0,010 ±	avgt 5 15,709 ± 0,309 avgt 5 0,391 ± 0,007 avgt 5 1,613 ± 0,112 avgt 5 0,241 ± 0,003 avgt 5 0,407 ± 0,036 avgt 5 0,267 ± 0,004 avgt 5 4,602 ± 0,057 avgt 5 0,058 ± 0,001 avgt 5 0,008 ± 0,001 avgt 5 0,008 ± 0,001 avgt 5 0,014 ± 0,001 avgt 5 0,010 ± 0,001

CONCLUSION:

Dans des listes courtes, le tri par insertion est bon mais pas sur des longues listes. Le tri de Java quant à lui est bon dans toutes les circonstances. Viens ensuite le quickSort. Le tri à bulle est le moins bon de tous.



1.6 SORTING AND SEARCHING, CLAIM 2

- Avoid linear search in arrays and lists, except when you know that they are very short. If your program needs to look up something frequently, use one of these approaches:
 - Binary search on sorted data:

For arrays, use java.util.Arrays.binarySearch. The array must be sorted, as if by java.util.Arrays.sort. There are overloaded versions for all primitive types and for objects.

For ArrayList<T>, use java.util.Collections.binarySearch. The array list must be sorted, as if by java.util.Collections.sort.

If you need also to insert or remove elements from the set or map, use one of the approaches below instead.

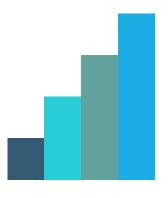
- Hashing: Use HashSet<T> or HashMap<K, V> from package java.util if your key objects have a good hash function hashCode. This is the case for String and the wrapper classes Integer, Double, ..., for the primitive types.
- Binary search trees: Use TreeSet<T> or TreeMap<K, V> from package java.util
 if your key objects have a good comparison function compareTo. This is the case for
 String and the wrapper classes Integer, Double, ..., for the primitive types.

RÉSULTATS:

Benchmark	Mode	Cnt	Score	Error	Units
SearchBenchmark.BinarySearchArray	avgt	5	0,966 ±	0,072	ms/op
SearchBenchmark.HashMapSearch	avgt	5	0,026 ±	0,001	ms/op
SearchBenchmark.TreeMapSearch	avgt	5	0,136 ±	0,017	ms/op
SearchBenchmark.regularSearchArray	avgt	5	1528,685 ±	85,134	ms/op

CONCLUSION:

La recherche d'élément de base est absolument à éviter. Si les objets à rechercher répondent aux critères cités plus haut. Il vaut mieux privilégiés une HashMap ou un TreeMap. Et dans le cas contraire une recherche dichotomique.



1.7 EXCEPTIONS

- The creation new Exception (...) of an exception object builds a stack trace, which is costly in time and space, and especially so in deeply recursive method calls. The creation of an object of class Exception or a subclass of Exception may be between 30 and 100 times slower than creation of an ordinary object. On the other hand, using a try-catch block or throwing an exception is fast.
- You can prevent the generation of this stack trace by overriding method fillInStackTrace
 in subclasses of Exception, as shown below. This makes creation exception instances roughly 10
 times faster.

```
class MyException extends Exception {
  public Throwable fillInStackTrace() {
    return this;
  }
}
```

Thus you should create an exception object only if you actually intend to throw it. Also, do not
use exceptions to implement control flow (end of data, termination of loops); use exceptions only
to signal errors and exceptional circumstances (file not found, illegal input format, and so on). If
your program does need to throw exceptions very frequently, reuse a single pre-created exception
object.

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units
MyBenchmark.test1 avgt 5 168,330 ± 12,940 ms/op
MyBenchmark.test2 avgt 5 1,221 ± 0,182 ms/op
MyBenchmark.test3 avgt 5 0,007 ± 0,001 ms/op
```

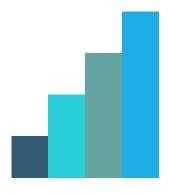
Test1: Exception classique.

Test2: Exception modifiée.

Test3: Exception modifiée créer une seule fois avant la boucle.

CONCLUSION:

Il est nécessaire d'override la méthode citée plus haut lors d'une création d'une exception manuelle qui ne nécessite pas la trace du stack. De plus si elle sera appelée souvent, il faut toujours utiliser la même exception créer une et une seule fois.



1.8 COLLECTION CLASSES, CLAIM 1

 Beware that indexing into a LinkedList<T> is not a constant-time operation. Hence the loop below takes time quadratic in the size of the list lst if lst is a LinkedList<T>, and should not be used:

```
int size = lst.size();
for (int i=0; i<size; i++)
  System.out.println(lst.get(i));</pre>
```

Instead, use the enhanced for statement to iterate over the elements. It implicitly uses the collection's iterator, so the traversal takes linear time:

```
for (T x : lst)
   System.out.println(x);
```

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 7,079 \pm 0,677 ms/op MyBenchmark.test2 avgt 5 0,017 \pm 0,002 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

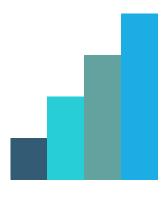
Test1: for classique avec get(i)

Test2: foreach dans la LinkedList

Test3: Pas de test.

CONCLUSION:

Le foreach est beaucoup plus rapide avec un facteur 700.



1.8 COLLECTION CLASSES, CLAIM 2

Repeated calls to add(int i, T x) or remove(int i) on LinkedList<T> should be avoided, except when i is at the end or beginning of the linked list; both perform a linear traversal to get to the i'th element.

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 0,085 \pm 0,070 ms/op MyBenchmark.test2 avgt 5 25,463 \pm 3,318 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

Test1: add et remove au début de la LinkedList

Test2: add et remove au milieu de la LinkedList

Test3: Pas de test.

CONCLUSION:

Il est beaucoup plus rapide d'ajouter à la fin ou au début d'une LinkedList plutôt qu'au milieu. (Facteur \sim 300)



1.8 COLLECTION CLASSES, CLAIM 3

Repeated calls to add(int i, T x) or remove(int i) on ArrayList<T> should be avoided, except when i is at the end of the ArrayList<T>; it needs to move all elements after i.

RÉSULTATS:

```
Benchmark
                   Mode
                                Score
                                         Error
                                                 Units
MyBenchmark.test1 avgt
                                2,562 ± 0,255
                                                ms/op
MyBenchmark.test2
                                0,107 ± 0,027
                   avgt
                                                ms/op
MyBenchmark.test3
                            5 ≈ 10<sup>-6</sup>
                   avgt
                                                 ms/op
```

Test1: add et remove au début de l'ArrayList

Test2: add et remove à la fin de l'ArrayList

Test3: Pas de test.

CONCLUSION:

Il est beaucoup plus rapide d'ajouter à la fin d'une l'ArrayList plutôt qu'au début pour les raisons citées plus haut. (Facteur \sim 25)



1.8 COLLECTION CLASSES, CLAIM 4

The collection classes can store only reference type data, so a value of primitive type such as int, double, ... must be wrapped as an Integer, Double, ... object before it can be stored or used as a key in a collection. This takes time and space and may be unacceptable in memory-constrained embedded applications. Note that strings and arrays are reference type data and need not be wrapped.

If you need to use collections that have primitive type elements or keys, consider using the Trove library, which provides special-case collections such as hash set of int and so on. As a result it is faster and uses less memory than the general Java collection classes. Trove can be found at http://trove4j.sourceforge.net/>.

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units MyBenchmark.test1 avgt 5 41,046 \pm 16,331 ms/op MyBenchmark.test2 avgt 5 337,214 \pm 70,208 ms/op MyBenchmark.test3 avgt 5 \approx 10<sup>-6</sup> ms/op
```

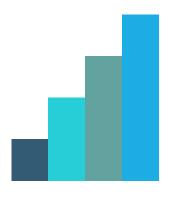
Test 1: add avec une arraylist de trove (int)

Test2: add avec une arraylist de java (Integer)

Test3: Pas de test.

CONCLUSION:

On est beaucoup plus rapide en utilisant les types primitifs comme dit plus haut (Facteur \sim 8)



1.9 INPUT AND OUTPUT

- Using buffered input and output (BufferedReader, BufferedWriter, BufferedInput-Stream, BufferedOutputStream from package java.io) can speed up input/output by a factor of 20.
- Using the compressed streams ZipInputStream and ZipOutputStream or GZIPInput-Stream and GZIPOutputStream from package java.util.zip may speed up the input and output of verbose data formats such as XML. Compression and decompression takes CPU time, but the compressed data may be so much smaller than the uncompressed data that it saves time anyway, because less data must be read from disk or network. Also, it saves space on disk.

RÉSULTATS:

```
Benchmark Mode Cnt Score Error Units
MyBenchmark.test1 avgt 5 508,403 ± 13,451 ms/op
MyBenchmark.test2 avgt 5 225,387 ± 4,552 ms/op
MyBenchmark.test3 avgt 5 473,749 ± 10,189 ms/op
```

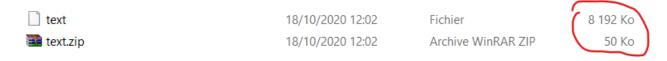
Test1: avec un Reader et Writer classique

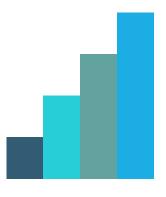
Test2: avec un BufferedReader et BufferedWriter

Test3: avec un BufferedReader et BufferedWriter et compression en Zip.

CONCLUSION:

L'utilisation des BufferedReader et BufferedWriter est beaucoup plus rapide que le reste (Facteur \sim 2), mais la compression en zip réduit la taille du fichier drastiquement.





1.11 BULK ARRAY OPERATIONS

There are special methods for performing bulk operations on arrays. They are usually much faster than equivalent for loops, in part because they need to perform only a single bounds check.

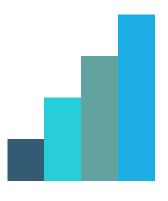
- static void java.lang.System.arrayCopy(src, si, dst, di, n) copies elements from array segment src[si..si+n-1] to array segment dst[di..di+n-1].
- static bool java.util.Arrays.equals(arr1, arr2) returns true if the arrays arr1 and arr2 have the same length and their elements are pairwise equal. There are overloads of this method for arguments of type boolean[], byte[], char[], double[], float[], int[],long[],Object[] and short[].
- static void java.util.Arrays.fill(arr, x) sets all elements of array arr to x. This method has the same overloads as Arrays.equals.
- static void java.util.Arrays.fill(arr, i, j, x) sets elements arr[i..j-1] to x. This method has the same overloads as Arrays.equals.

RÉSULTATS:

Benchmark	Mode	Cnt	Score	Error	Units
ArrayMethodBenchmark.ArrayCopy	avgt	5	13,518 ±	2,859	ms/op
ArrayMethodBenchmark.ArrayEqual	avgt	5	18,933 ±	1,102	ms/op
ArrayMethodBenchmark.ArrayFill	avgt	5	10,015 ±	0,633	ms/op
ArrayMethodBenchmark.regularCopy	avgt	5	12,821 ±	1,146	ms/op
ArrayMethodBenchmark.regularEqual	avgt	5	16,001 ±	0,796	ms/op
ArrayMethodBenchmark.regularFill	avgt	5	9,893 ±	0,982	ms/op

CONCLUSION:

Il n'y a pas de différence majeure entre les méthodes faites à la main et celle natives de java.util.Arrays



1.13 REFLECTION

- A reflective method call, reflective field access, and reflective object creation (using package java.lang.reflect) are far slower than ordinary method call, field access, and object creation.
- Access checks may further slow down such reflective calls; some of this cost may be avoided by
 declaring the class of the called method to be public. This has been seen to speed up reflective
 calls by a factor of 8.

RÉSULTATS:

Benchmark	Mode	Cnt	Score		Error	Units
MyBenchmark.test1	avgt	5	3,001	±	0,305	ms/op
MyBenchmark.test2	avgt	5	55,099	±	6,146	ms/op
MyBenchmark.test3	avgt	5	61,364	±	10,451	ms/op

Test1: appel de la méthode normalement.

Test2: appel de la méthode par réflexion.

Test3: appel de la méthode par réflexion avec un check de l'accessibilité avant.

CONCLUSION:

En passant par la réflexion, on a un facteur \sim 18. Mais en regardant l'accessibilité avant on ne perd pas énormément de temps contrairement à ce qui est dit au-dessus.

