

La récursivité terminale

Qu'est-ce que la récursivité ?

En informatique, une fonction (« méthode » en Java) ou plus généralement un algorithme qui contient un appel à elle-même est dite récursive. Cependant, il existe différentes sortes de récursivité, dont l'une dite « terminale » et l'autre non.

Quelle différence entre ces deux récursivités ?

Une invocation récursive d'une fonction f est dite « terminale » si la dernière opération effectuée lors du retour est un appel à celle-ci ; autrement dit, la valeur retournée est directement la valeur obtenue par l'invocation récursive, sans qu'il n'y ait d'opération sur cette valeur.

Méthode récursive calculant la factorielle d'un nombre

```
static long factorielle(long n) {
    if (n<0) return -1;
    return (n==0 || n==1)?1:n*factorielle(n-1);
}
```

Cette méthode est récursive, mais non terminale puisque la dernière opération effectuée avant le retour est une multiplication. Elle est donc plus coûteuse en ressource puisqu'elle occupe l'espace de la pile afin d'effectuer la multiplication et est plus lente que la méthode suivante.

Méthode récursive terminale

```
static long factorielle(long n, long r) {
    if (n<0)
        return -1;
    else if (n==0)
        return 1;
    return (n==1)?r:factorielle(n-1,n*r);
}
```

Cette méthode est récursive terminale, elle est beaucoup plus performante que la méthode précédente et autant qu'une méthode itérative. C'est ce que nous allons démontrer (au moins essayer!).

Classe définissant trois méthodes factorielle différentes et tests

```
public class Test {
    static long factR(long n, int f) {
        if (n<0)
            return -1;
        else if (n==0)
            return 1;
        return ((f & 1)==0)?factBis(n,1):factBis2(n);
    }

    static long factBis(long n, long r) {
        return (n==1)?r:factBis(n-1,n*r);
    }

    static long factBis2(long n) {
        return (n==1)?n:n*factBis2(n-1);
    }

    static long factI(long n) {
        if (n==0 || n==1)
            return 1;
        else if (n<0)
            return -1;

        long r=n;

        while(n>1)
            r*=-n;

        return r;
    }

    public static void main(String[] args) {
        long t0=System.currentTimeMillis();

        System.out.println(factR(25,1));

        long t1=System.currentTimeMillis();

        System.out.println(t1-t0+"ms.");

        t0=System.currentTimeMillis();

        System.out.println(factR(25,2));

        t1=System.currentTimeMillis();

        System.out.println(t1-t0+"ms.");

        t0=System.currentTimeMillis();

        System.out.println(factI(25));

        t1=System.currentTimeMillis();

        System.out.println(t1-t0+"ms.");
    }
}
```

On peut observer 4 méthodes:

1. **factR**: *long, int -> long*

Elle appelle la méthode récursive correspondant à l'entier donné, pair -> factBis, impair -> factBis2 et gère les conventions ainsi que les erreurs. Par convention 0! = 1 et retourne -1 en cas d'erreur (entier négatif).

2. **factBis**: *long, long -> long*

La méthode récursive terminale.

3. **factBis2**: *long -> long*

La méthode récursive.

4. **factI**: *long -> long*

La méthode itérative.

A l'exécution de la méthode *main*, on s'aperçoit alors que la méthode récursive est plus lente que la méthode récursive terminale et itérative, qui elles, sont égales.

La différence de temps observée n'est pas réellement significative dans le cas présent, elle n'est que de 1 milliseconde. Mais en changeant le type *long* en type *double* on s'aperçoit alors que le temps mis par la méthode récursive est de 6 voire 7 millisecondes alors que les deux autres sont toujours de 0 milliseconde.

```
$ java Test
7034535277573963776
1ms.
7034535277573963776
0ms.
7034535277573963776
0ms.
```

Dans cette exemple j'ai voulu respecter la définition d'une factorielle qui dit:

En [mathématiques](#), la **factorielle** d'un [entier naturel](#) n , notée $n!$, ce qui se lit soit « factorielle de n » soit « factorielle n », est le [produit](#) des nombres entiers strictement positifs inférieurs ou égaux à n .

J'ai donc conservé le type **long** qui est un entier codé sur **8 octets** (son intervalle de valeur est donc de **-9 223 372 036 854 775 808** à **9 223 372 036 854 775 807**) contrairement au type **double** qui est un **nombre à virgule** flottante double précision codé sur **8 octets**.

nb: Une classe benchmark aurait pu être écrite afin de rendre la méthode *main* plus propre et surtout plus lisible. Je pense aussi que la Suite de Fibonacci aurait fait un exemple bien plus concluant, mais tant pis !

Méthodes récursives terminales de la classe Liste : algorithme et code

algorithme	code
ajouter : <i>int x, Liste a -> Liste</i>	<pre>static Liste ajouter(int x, Liste a) { return new Liste(x, a); }</pre>
longueur : <i>Liste a -> int</i> 1) <i>longueur</i> : <i>a -> longueur_aux: a, 0</i>	<pre>static int longueur(Liste a) { return longueur_aux(a, 0); }</pre>
longueur_aux : <i>Liste a, int n -> int</i> 1) <i>a = [] -> n</i> 2) <i>longueur_aux: a, n+1</i>	<pre>static int longueur_aux(Liste a, int n) { if (a==null) return n; return longueur_aux(a, n+1); }</pre>

algorithme (suite)	code (suite)
inverser: Liste a -> Liste 1) inverser: a -> inverser_aux: a, b	<pre>static Liste inverser(Liste a) { return inverser_aux(a, null); }</pre>
inverser_aux: Liste a, Liste b -> Liste 1) a = [] -> b 2) inverser_aux: a.suivant, Liste(a.contenu, b)	<pre>static Liste inverser_aux(Liste a, Liste b) { if (a==null) return b; return inverser_aux(a.suivant, new Liste(a.contenu, b)); }</pre>
copier: Liste a -> Liste 1) copier: a -> inverser: inverser(a)	<pre>static Liste copier(Liste a) { return inverser(inverser(a)); }</pre>
concat: Liste a, Liste b -> Liste 1) concat: a, b -> concat_aux: copier(a), b	<pre>static Liste concat(Liste a, Liste b) { Liste copie = copier(a); return concat_aux(copie, b, copie); }</pre>
concat_aux: Liste a, Liste b, Liste c -> Liste 1) a = [] -> a=b; retourne c; 2) concat_aux: a.suivant, b, c	<pre>static Liste concat_aux(Liste a, Liste b, Liste c) { if (a==null) { a=b; return c; } return concat_aux(a.suivant, b, c); }</pre>

Juste pour le fun !

```
public interface Fonction {
    public long lancer();
    public String nom();
}
```

```
public class Benchmark {
    public void eval(Fonction f) {
        StringBuilder sb=new StringBuilder(f.nom()+"\n");

        long t0=System.currentTimeMillis();
        long result=f.lancer();
        long t1=System.currentTimeMillis();

        sb.append("\ndebut: "+t0+"\nfin: "+t1+"\n\nresultat: "+result+" obtenu en "+(t1-
t0)+" millisecondes.\n");

        System.out.println(sb);
    }
}
```

```
public class FiboT implements Fonction {
    long n=0L;

    FiboT(long n) {
        this.n=n;
    }

    public String nom() {
        return "méthode récursive terminale";
    }

    public long lancer() {
        return Test.fibonacci(n,2);
    }
}
```

```
public class FiboR implements Fonction {
    long n=0L;

    FiboR(long n) {
        this.n=n;
    }

    public String nom() {
        return "méthode récursive";
    }

    public long lancer() {
        return Test.fibonacci(n,1);
    }
}
```

```
public class FiboI implements Fonction {
    long n=0L;

    FiboI(long n) {
        this.n=n;
    }

    public String nom() {
        return "méthode itérative";
    }

    public long lancer() {
        return Test.fiboI(n);
    }
}
```

```

public class Test {
    static long fibonacci(long n,int f) {
        return ((f & 1)==0)?fiboT(0,1,n):fiboR(n);
    }

    static long fiboT(long nb1, long nb2, long n) {
        if (n==0)
            return 0;
        if (n<2)
            return nb2;
        return fiboT(nb2,nb1+nb2,n-1);
    }

    static long fiboR(long n) {
        if (n<2)
            return n;
        return fiboR(n-1)+fiboR(n-2);
    }

    static long fiboI(long n) {
        long a=0L,b=1L;

        if (n<=1)
            return n;

        long c=0L;

        for(int i=1;i<n;i++) {
            c = a + b;
            a = b;
            b = c;
        }

        return c;
    }

    public static void main(String[] args) {
        Benchmark bench=new Benchmark();

        bench.eval(new FiboT(50));
        bench.eval(new FiboR(50));
        bench.eval(new FiboI(50));
    }
}

```

```

$ java Test
méthode récursive terminale

debut: 1211640777474
fin: 1211640777474

resultat: 12586269025 obtenu en 0 millisecondes.

méthode récursive

debut: 1211640777475
fin: 1211640921403

resultat: 12586269025 obtenu en 143928 millisecondes.

méthode itérative

debut: 1211640921403
fin: 1211640921403

resultat: 12586269025 obtenu en 0 millisecondes.

```