# Table des matières

# Chapitre 10

# Programmation parallèle et concurrente avec Java

Ce chapitre présente les notions de base de la programmation concurrente et parallèle avec *threads* en Java, et ce entre autre à l'aide de divers exemples. Le chapitre présente aussi divers éléments de la bibliothèque `java.util.concurrent`. Toutefois, pour bien utiliser les versions plus récentes des *threads* et de cette bibliothèque, il faut au préalable présenter les lambda-expressions, introduites en Java 7.0.

## 10.1 Lambda-expressions

Nous allons illustrer les lambda-expressions à l'aide de variables et d'interfaces (explicites), pour mieux comprendre leur typage et montrer aussi que les lambda-expressions **sont des objets**.

Une lambda-expression est un objet qui représente une *méthode anonyme* et l'affectation à une variable est une façon de lui donner un nom... et *de rendre explicite la méthode associée.*

### 10.1.1 Diverses interfaces utiles pour les lambda-expressions et les *threads*

La bibliothèque `java.util.concurrent` définit un certain nombre d'interfaces de de base, utiles pour la manipulation des *threads*, alors que `java.util.function` définit diverses fonctions utiles pour des lambda-expressions.

#### Les interfaces `Runnable`, `Callable<V>` et `Future<V>`

Les extraits en anglais ci-bas sont extraits de la documentation Java d'Oracle.

### `Runnable` vs. `Callable<V>`

*A task that returns a result and may throw an exception. Implementors define a single method with no arguments called* `call`.

*The* `Callable interface` *is similar to* `Runnable`*, in that both are designed for classes whose instances are potentially executed by another thread. A* `Runnable`*, however, does not return a result and cannot throw a checked exception.*

Un `Runnable` peut être appelé avec `run()` mais ne peut pas retourner de résultat (la méthode `run` retourne le type `void`) ☹

```
interface Runnable {
   void run()
}
```

Par contre, un `Callable`, appelé avec `call()`, permet de retourner un résultat — de type `V` (générique) ☺

```
public interface Callable<V> {
  V call()
}
```

### L'interface `Future<V>`

*A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method* `get` *when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the* `cancel` *method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled.*

```
public interface Future<V>  {
  boolean cancel(boolean mayInterruptIfRunning)

  V get()

  V get(long timeout, TimeUnit unit)

  boolean isCancelled()

  boolean isDone()
}
```

*Note :* Les interfaces `Callable<V>` et `Future<V>` sont définies dans `java.util.concurrent`, donc il faut faire un `import`!

## Les interfaces de type `FunctionalInterface`

Une interface est *fonctionnelle* si elle n'exporte qu'une seule et unique méthode — sauf peut-être aussi une ou des méthodes de la classe `Object`.

*Functional interfaces provide target types for lambda expressions and method references.*
Source : `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`

De nombreuses interfaces de ce genre ($\approx$ 50) sont définies dans le *package* `java.util.function`

| | |
|---|---|
| `BiConsumer<T,U>` | Represents an operation that accepts two input arguments and returns no result. |
| `BiFunction<T,U,R>` | Represents a function that accepts two arguments and produces a result. |
| `BinaryOperator<T>` | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| `BiPredicate<T,U>` | Represents a predicate (boolean-valued function) of two arguments. |
| ⋮ | ⋮ |
| `Function<T,R>` | Represents a function that accepts one argument and produces a result. |
| ⋮ | ⋮ |
| `ToIntBiFunction<T,U>` | Represents a function that accepts two arguments and produces an int-valued result. |
| `ToIntFunction<T>` | Represents a function that produces an int-valued result. |
| `ToLongBiFunction<T,U>` | Represents a function that accepts two arguments and produces a long-valued result. |
| `ToLongFunction<T>` | Represents a function that produces a long-valued result. |
| `UnaryOperator<T>` | Represents an operation on a single operand that produces a result of the same type as its operand. |

### Un exemple : `Function<T,R>`

```
@FunctionalInterface
public interface Function<T,R> {
  // Applies this function to the given argument.
  R apply(T t)
}
```

### 10.1.2 Des exemples de lambda-expressions

**Des lambda-expressions de style `Runnable` (aucun argument ou résultat)**

```
Runnable r0
  = () -> { System.out.println( "Dans r0" ); };
r0.run();

int x = 0;
Runnable r1
  = () -> System.out.println( "x = " + x );
r1.run();

interface Fooable { void foo(); }
interface Barable { void bar(); }

Runnable r2 = () -> System.out.println( "x = " + x );
Fooable  f  = () -> System.out.println( "x = " + x );
Barable  b  = () -> System.out.println( "x = " + x );

r2.run();
f.foo();
b.bar();
```

*Par contre* :

```
int x = 0;
Runnable r2
  = () -> System.out.println( "x = " + x );
r2.run();


x = 3;
Runnable r3
  = () -> System.out.println( "x = " + x );
r3.run();

-----------------------
Lambdas.java:57: error: local variables referenced from a
                 lambda expression must be final or effectively final
   Runnable r2 = () -> System.out.println( "x = " + x );
                                                    ^
Lambdas.java:61: error: local variables referenced from a
                 lambda expression must be final or effectively final
   Runnable r3 = () -> System.out.println( "x = " + x );
```

^

**Des lambda-expressions `Callable`**

```
// interface Callable<V> { V call(); }
import java.util.concurrent.*;

Callable<Integer> c1 = () -> { return 10; };
try { ... c1.call() ... } ...


int xx = 0, yy = 22;
Callable<Integer> c2 = () -> { return xx + yy; };
try { ... c2.call() ... } ...


Callable<Integer> c3 = () -> xx + yy;
try { ... c3.call() ... } ...
```

**Des lambda-expressions qui satisfont une interface *fonctionnelle* du *package* `java.util.function`**

```
// interface Function<T,R> { R apply(T t); }
import java.util.function.*;

Function<Integer,Integer> f1
   = ( w ) -> { return w+1; };

... f1.apply(30) ...


int z = 22;
Function<Integer,Integer> f2
  = ( w ) -> { return w+z; };

... f2.apply(30) ...


Function<Integer,Integer> f3
   = w -> w+z;

... f3.apply(30) ...
```

**Des lambda-expressions qui satisfont une interface *fonctionnelle* définie par le programmeur**

**Une interface avec une seule et unique méthode**

```
interface Addable { int add( int x ); }

int y = 0;

Addable a1 = ( int w ) -> { return w + y; };
.. a1.add(20) ...

Addable a2 = ( w ) -> { return w + y; };
... a2.add(20) ...

Addable a3 = w -> w + y;
... a3.add(20) ...
```

*Par contre* :

```
int y = 0;

Addable a4 = ( int y ) -> { return y + y; };
... a4.add(20) ...

=>
Lambdas.java:111: error: variable y is already defined in
                  method main(String[])
        Addable a4 = ( int y ) -> { return y + y; };
```

**Une interface avec plusieurs méthodes mais une seule non définie dans `Object`**

```
interface MonFooable {
    void foo();
    int hashCode();
    String toString();
}

MonFooable mf0 = () -> System.out.println( "Dans mf0" );
mf0.foo();
```

**Une interface avec plusieurs méthodes non définies dans `Object`**

```
interface MesRunnables {
    void run0();
    void run1();
}

MesRunnables rs0 = () -> System.out.println( "Dans rs0" );

----
Lambdas.java:140: error: incompatible types:
             MesRunnables is not a functional interface
        MesRunnables rs0 = () -> System.out.println( "Dans rs0" );
                           ^
    multiple non-overriding abstract methods found
             in interface MesRunnables
```

## 10.2 Références à des méthodes (*method references*)

Depuis Java 8.0, il est possible de créer des *références à des méthodes — method references*. De telles références peuvent être utilisées comme argument à des méthodes qui reçoivent des objets fonctionnels, en lieu et place de lambda-expressions. Le Programme Java 10.1 présente un premier petit exemple.

- La méthode `indexOf` retourne l'index du premier élément du tableau `a` ($1^{er}$ argument) qui satisfait le `predicat` ($2^{e}$ argument).

  Ici, l'interface `Predicate<T>` est «semblable» à l'interface `Function<T,Boolean>`, à la différence que `Predicate` possède une méthode `test` alors que `Function` possède une méthode `apply`.

- Dans la méthode `main`, le premier appel à `indexOf` est fait avec une lambda-expression. Par contre, le deuxième appel est fait avec une référence à une méthode. Dans les deux cas, il s'agit du même prédicat, qui détermine si un entier est pair.

- Une référence à une méthode est indiquée par un nom de classe — ou une référence à un objet : voir plus bas —, suivi des caractères «`::`», suivi d'un *nom* de méthode... sans argument!

  Une telle référence n'est pas un appel à la méthode, puisqu'il n'y a pas d'argument. Toutefois, ultérieurement, dans le corps de la méthode appelée (ici `indexOf`), un appel à la méthode pourra être effectuée en fournissant un argument approprié.

- Une référence à une méthode peut être interprétée simplement comme une lambda-expression générée par le compilateur, et ce en fonction de la signature de la méthode référée.

  Dans le $2^{e}$ appel, la signature de la méthode référée est `static Boolean estPair(Integer)`. Elle est donc équivalente à la lambda-expression suivante, comme l'illustre le $3^{e}$ appel à `indexOf` dans le Programme Java 10.1 :

  ```
  x -> ExemplesMethodRefs.estPair(x)
  ```

**Programme Java 10.1** Un petit exemple de programme illustrant l'utilisation d'une référence à une méthode : Appel d'une méthode de classe (méthode `static`).

```
import java.util.function.*;

public class ExemplesMethodRefs {

  public static <T> int indexOf( T[] a,
                                 Predicate<T> predicat ) {
    for( int i = 0; i < a.length; i++ ) {
      if ( predicat.test(a[i]) ) return i;
    }
    return -1;
  }

  public static Boolean estPair( Integer x ) {
    return x % 2 == 0;
  }

  public static void main(String[] args) {
    Integer[] a =
        new Integer[]{ 5, 4, 3, 2, 1, 3, 2, 3, 4, 5 };

    // Appel d'une methode de classe (methode static);

    // Appel de estPair avec une lambda-expression.
    int r1 =
       indexOf( a, x -> ExemplesMethodRefs.estPair(x) );

    // Appel equivalent avec une reference de methode.
    int r2 =
       indexOf( a, ExemplesMethodRefs::estPair );

    ...
  }
}
```

Les trois sortes de références à des méthodes et leurs lambda-expressions équivalentes sont les suivantes — en supposant dans le $2^e$ cas que `arg0` soit un objet de classe `ClassName` (tiré de [UFM15]) :

| **Référence** | **Lambda-expression** |
|---|---|
| `ClassName::staticMethod` | `(args) -> ClassName.staticMethod(args)` |
| `ClassName::instanceMethod` | `(arg0, rest) -> arg0.instanceMethod(rest)` |
| `expr::instanceMethod` | `(args) -> expr.instanceMethod(args)` |

Le Programme Java 10.2 présente un petit exemple de la troisième forme, donc avec une référence à une méthode d'un objet spécifique.

Il est aussi possible de créer des références à des *constructeurs* — voir [UFM15] pour plus de détails.

**Programme Java 10.2** Un petit exemple de programme illustrant l'utilisation d'une référence à une méthode d'instance.

```
class Foo<T> {
  private T val;

  Foo( T val ) {
    this.val = val;
  }

  boolean egal( T v ) {
    return val.equals(v);
  }
}

public class ExemplesMethodRefs {

  public static <T> int indexOf( T[] a,
                                 Predicate<T> predicat ) {
    // Commme precedemment...
    ...
  }

  public static void main(String[] args) {
    ...

    // Appel d'une methode d'instance.

    Foo<Integer> foo = new Foo<>( 3 );

    // Avec lambda-expression.
    int r3 = indexOf( a, x -> foo.egal(x) );

    // Avec une reference de methode.
    int r4 = indexOf( a, foo::egal );
    ...
  }
}
```

## 10.3 Classe `Thread` vs. interface `Runnable`

La classe `Thread` est la classe de base, dont on peut hériter pour ensuite créer des objets qui sont, via leur super-classe, des *threads*. Toutefois, *il est plutôt recommandé de réaliser un* thread *en mettant en oeuvre l'interface* `Runnable`, et ce pour deux raisons : *i*) parce que Java ne supporte pas l'héritage multiple, donc si on hérite de `Thread`, alors on ne peut plus hériter d'une autre classe ; *ii*) parce qu'on ne peut faire qu'un seul appel à `start()`, i.e., on ne peut pas avoir deux instances actives du *thread* à moins de créer un nouvel objet tout à fait distinct.

L'interface `Runnable` est simplement définie comme suit :

```
interface Runnable {
   void run();
}
```

La «bonne nouvelle» en Java 8.0 : une lambda-expression est un `Runnable` ☺

```
$ cat ExempleRunnable.java
  ...
  Runnable r0
    = () -> System.out.println("Bonjour!");
  r0.run();
  ...
------------------
$ java ExempleRunnable
Bonjour!
```

La classe `Thread` est, en partie, définie comme dans le Programme Java 10.3.

**Programme Java 10.3** La classe `Thread` de Java (spécification partielle).

```
class Thread implements Runnable {
   Thread() {...}
   Thread( Runnable target ) {...}
   Thread(...) {...}

   public static Thread currentThread() {...}

   public static void sleep( long milis ) {...}
   public static void yield() {...}

   public long getId() {...}
   public long getName() {...}
   public long getThreadGroup() {...}

   public void interrupt() {...}
   public boolean isAlive() {...}
   public boolean isInterrupted() {...}

   public void join() {...}
   public void join(...) {...}

   public void run() {...}
   public void start() {...}
}
```

**Cycle de vie d'un *thread* «primitif»** Les étapes de base d'utilisation d'un *thread* sont les suivantes :

1. On crée un objet `Thread`.
2. On lance l'exécution de l'objet `Thread` avec la méthode `start()`.

   L'appel de cette méthode effectue alors, entre autre, un appel à la méthode `run()` de l'objet.
3. On attend la fin de l'exécution de l'objet `Thread` en appelant la méthode `join()`.

**Création et activation de *threads*** Les deux principales façons pour créer une instance de la classe `Thread` sont les suivantes :

- On définit une classe qui hérite de la classe `Thread` et qui définit une méthode `run` :

```
class C extends Thread {
    ...
    void run() { ... }
    ...
}

C c = new C( ... );
c.start();
```

  Possible... **mais, maintenant, on ne fait pas ça!**

- On définit une classe qui met en oeuvre l'interface `Runnable`, et on crée un *thread* en utilisant un constructeur approprié de la classe `Thread`, qui transforme un `Runnable` en un `Thread` :

```
class C implements Runnable {
  ...
  void run() { ... }
  ...
}
...
Thread t = new Thread( new C() );
t.start();
```

**Méthode à utiliser de préférence : Via `Runnable`!**

Mais en fait, de nos jours, avec les *pools de threads*, **c'est rare qu'on crée des *threads* de cette façon!**

## 10.4 Exemples simples comparant la création des *threads* en MPD, PRuby et Java

**Note :** Certains de ces exemples qui suivent sont présentés en MPD, un langage qui utilise une instruction de type `cobegin/coend` pour lancer des *threads*. Bien que nous ne verrons pas ce langage dans le cours, les exemples devraient quand même pouvoir être compris facilement — notamment parce qu'une version `Ruby`/`PRuby` est aussi présentée.

Cette section présente une brève comparaison entre MPD, Ruby et Java quant à la façon de créer des *threads*. Dans le cas de Java, plusieurs façons différentes de créer ces *threads* sont parfois présentées, notamment pour illustrer l'évolution du langage et les simplifications apportées par l'introduction des lambda-expressions.

Dans ces exemples, on suppose l'existence de la procédure et des fonctions suivantes (exprimées ici en MPD) :

```
procedure pfoo( int num_thread, int a1 )

function ffoo( int num_thread, int a1 )
         returns int

function bar( int x )
         returns int
```

La procédure `pfoo` et la fonction `ffoo` prennent en argument un numéro de thread (pour l'identification du *thread*) et un argument entier — on pourrait facilement généraliser à 0, 1 ou plusieurs arguments, entiers ou non. Quant à la fonction `bar`, elle sert simplement à représenter l'évaluation d'une expression passée en argument lors de l'appel de la procédure/fonction activée via un thread.

Trois formes d'activation de *threads* sont présentées :

1. Appel de procédure sans attente de terminaison — section 10.4.1
2. Appel de procédure avec attente de terminaison — section 10.4.2
3. Appel de fonction avec réception du résultat — section 10.4.3

### 10.4.1 Appel de procédure sans attente de terminaison (exécution en arrière-plan)

**Version MPD**

```
for [k = 0 to nb_threads-1] {
  fork pfoo( k, bar(k) );
}
```

**Version PRuby**

```
(0...nb_threads).each do |k|
  PRuby.future { pfoo( k, bar(k) ) }
end
```

**Version Java avec classe interne anonyme—avant Java 8.0**

```
for( int k = 0; k < nbThreads; k++ ) {
   final int kf = k;
   new Thread( new Runnable() {
       public void run() {
          pfoo( kf, bar(kf) );
       }
   } ).start();
}
```

**Version Java avec lambda-expression—depuis Java 8.0**

```
for( int k = 0; k < nbThreads; k++ ) {
   final int kf = k;
   new Thread(
      () -> pfoo( kf, bar(kf) )
   ).start();
}
```

### 10.4.2 Appel de procédure avec attente de terminaison

#### Version MPD

```
co [k = 0 to nb_threads-1]
  pfoo( k, bar(k) );
oc
```

#### Versions PRuby

```
# Avec pcall.
PRuby.pcall( 0...nb_threads,
             lambda { |k| pfoo( k, bar(k) ) }
)


# Avec future.
fs = (0...nb_threads).map do |k|
  PRuby.future { pfoo( k, bar(k) ) }
end
fs.each(&:join)
```

### Version Java avec classe interne anonyme

```
Thread ts[] = new Thread[nbThreads];
for( int k = 0; k < nbThreads; k++ ) {
    final int kf = k; // Var. non locale dans run.
    ts[k] = new Thread( new Runnable() {
            public void run() {
               pfoo( kf, bar(kf) );
            }
       } );
    ts[k].start();
}

for( int k = 0; k < nbThreads; k++ ) {
   try {
     ts[k].join();
   } catch( InterruptedException ie ) {
     ...
   }
}
```

### Version Java 8.0 avec lambda-expression

```
Thread ts[] = new Thread[nbThreads];
for( int k = 0; k < nbThreads; k++ ) {
    final int kf = k;
    ts[k] = new Thread(
        () -> pfoo( kf, bar(kf) )
    );
    ts[k].start();
}

for( int k = 0; k < nbThreads; k++ ) {
    try {
      ts[k].join();
    } catch( InterruptedException ie ) {
      ...
    }
}
```

### Version Java avec classe auxiliaire

```
Thread ts[] = new Thread[nbThreads];
for( int k = 0; k < nbThreads; k++ ) {
  ts[k] = new Thread(
    new ClassePourThread(k, bar(k))
  );
  ts[k].start();
}

for( int k = 0; k < nbThreads; k++ ) {
    try {
      ts[k].join();
    } catch( InterruptedException ie ) {...}
}


// Classe auxiliaire pour representer le thread,
// i.e., l'appel de procedure et les arguments
// a transmettre... pcq. run ne prend aucun argument :(
class ClassePourThread implements Runnable {
  private int k, a1;

  ClassePourThread( int k, int a1 ) {
      this.k = k;
      this.a1 = a1;
  }

  public void run() {
      pfoo( k, a1 );
  }
}
```

### 10.4.3 Appel de fonction avec résultat

#### Version MPD

```
int r[0:nb_threads-1]
co [k = 0 to nb_threads-1]
  r[k] = ffoo( k, bar(k) );
oc
```

#### Version PRuby

```
fs = (0...nb_threads).map do |k|
  PRuby.future { ffoo( k, bar(k) ) }
end
r = fs.map(&:value)
```

**Version Java 8.0 avec lambda-expression**

```
ExecutorService pool
  = Executors.newCachedThreadPool();
Future<Integer>[] fs
  = new Future[nbThreads];  // unchecked cast
int r[]
  = new int[nbThreads];

for( int k = 0; k < nbThreads; k++ ) {
    final int kf = k;
    fs[k] = pool.submit(
        () -> ffoo( kf, bar(kf) )
    );
}

for( int k = 0; k < nbThreads; k++ ) {
    try {
      r[k] = fs[k].get();
    } catch( Exception e ) {
      ...
    }
}

pool.shutdown();
```

## 10.5 Fonction pour le calcul de $\pi$ avec méthode Monte Carlo

Le Programme Java 10.4 présente un extrait d'un programme `Pi.java` avec une fonction `evaluerPi` — et une fonction auxiliaire `nbDansCercleSeq` — pour estimer la valeur de $\pi$ à l'aide d'une méthode de Monte Carlo, donc semblable aux fonctions Ruby vues précédemment.

**Programme Java 10.4** Une fonction parallèle `evaluerPi` (et sa fonction auxiliaire `nbDansCercleSeq`) pour approximer la valeur de $\pi$ à l'aide de la méthode de Monte Carlo. Il s'agit de la version Java d'une fonction Ruby vue précédemment : Programme Ruby **??** (style impératif).

```
public static int nbDansCercleSeq( int nbLancers ) {
  Random rnd = new Random();

  int nb = 0;
  for( int k = 0; k < nbLancers; k++ ) {
    double x = rnd.nextDouble();
    double y = rnd.nextDouble();
    if( x * x + y * y <= 1.0 ) {
      nb += 1;
    }
  }

  return nb;
}

//

@SuppressWarnings("unchecked")
public static double evaluerPi( final int nbLancers,
                                int nbThreads ) {
  ExecutorService pool
    = Executors.newFixedThreadPool( nbThreads );

  // On lance les threads via des futures.
  Future<Integer> lesNbs[]
    = new Future[nbThreads]; // unchecked cast
  for( int k = 0; k < nbThreads; k++ ) {
    lesNbs[k] = pool.submit(
      () -> nbDansCercleSeq(nbLancers/nbThreads)
    );
  }

  // On recoit les resultats.
  int nbTotalDansCercle = 0;
  for( int k = 0; k < nbThreads; k++ ) {
    try {
      nbTotalDansCercle += lesNbs[k].get();
    } catch( Exception e ) {
    }
  }

  pool.shutdown();

  return 4.0 * nbTotalDansCercle / nbLancers;
}
```

## Analyse des performances du programme `Pi.java`

La Figure 10.1 présente les temps d'exécution (temps moyen pour cinq (5) exécutions, en seconde) du programme Java 10.4, et ce en faisant varier le nombre de *threads* pour effectuer un total de **10 000 000 lancers** — donc 10 000 000 lancers à effectuer qui sont *partagés* entre les divers *threads*. Ces mesures de temps d'exécution ont été effectuées avec la commande `time` au niveau du *shell*.

Quant à la figure 10.2, elle présente les accélérations *relatives* (et non absolues) résultantes. On constate que les accélérations sont plutôt faibles — au mieux 2.5.

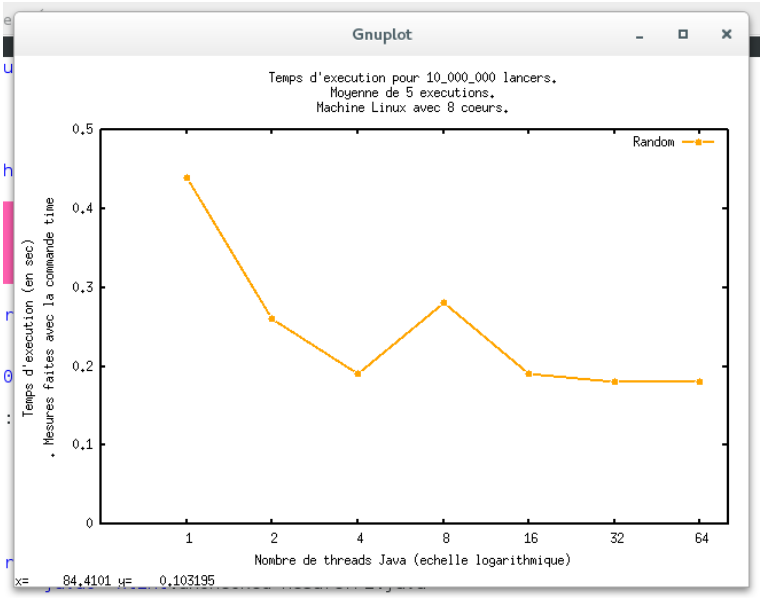



Figure 10.1: Graphe donnant les temps d'exécution (en sec.) du programme `Pi.java` pour différents nombres de *threads*— moyenne pour cinq exécutions, sur une machine Linux CentOS avec huit (8) coeurs.

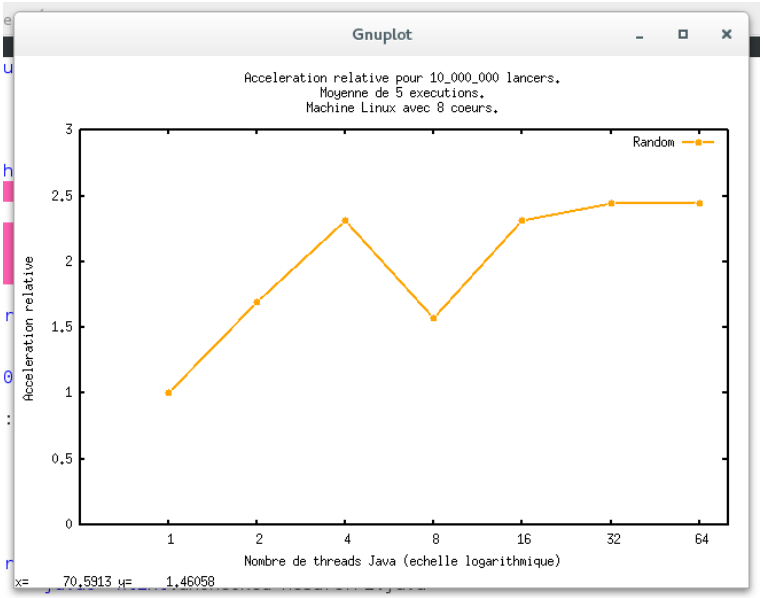



Figure 10.2: Graphe donnant les accélérations *relatives* pour `Pi.java` — moyenne pour cinq exécutions, sur une machine Linux CentOS avec huit (8) coeurs.

En consultant la documentation Java, on constate qu'il y a un problème avec le fait d'utiliser `Random` dans un programme parallèle. Les objets de classe `Random` sont *thread-safe*, donc peuvent être partagés entre *threads*, ce qui est une bonne chose. Le «hic», c'est que le fait qu'ils soient *thread-safe* introduit des surcoûts de synchronisation, ce qui ralentit grandement l'exécution ☹

Une solution alternative est d'utiliser une autre classe pour la génération de nombres pseudo-aléatoires, `ThreadLocalRandom`, qui génère correctement des nombres pseudo-aléatoires, et ce sans interférences entre *threads* et sans surcoûts significatifs en présence de *threads* multiples :

```
public static int nbDansCercleSeq( int nbLancers ) {
  ThreadLocalRandom rnd = ThreadLocalRandom.current();
  ...
```

On peut voir l'effet sur les temps d'exécution et les accélérations dans les Figures 10.3 et 10.4, qui comparent les temps et accélérations pour `Pi.java` utilisant `Random` vs. `ThreadLocalRandom`. On constate que les temps d'exécution diminuent grandement avec `ThreadLocalRandom`. Par contre, les accélérations sont nettement plus petites. Il s'agit d'une situation qu'on rencontre fréquemment : un programme *optimisé*, qui s'exécute plus rapidement, a souvent des accélérations (tant relatives qu'absolues) beaucoup plus faibles : intuitivement, c'est parce que le temps d'exécution étant plus petit, il est d'autant plus difficile de le diminuer — mais voir aussi l'exercice qui suit.

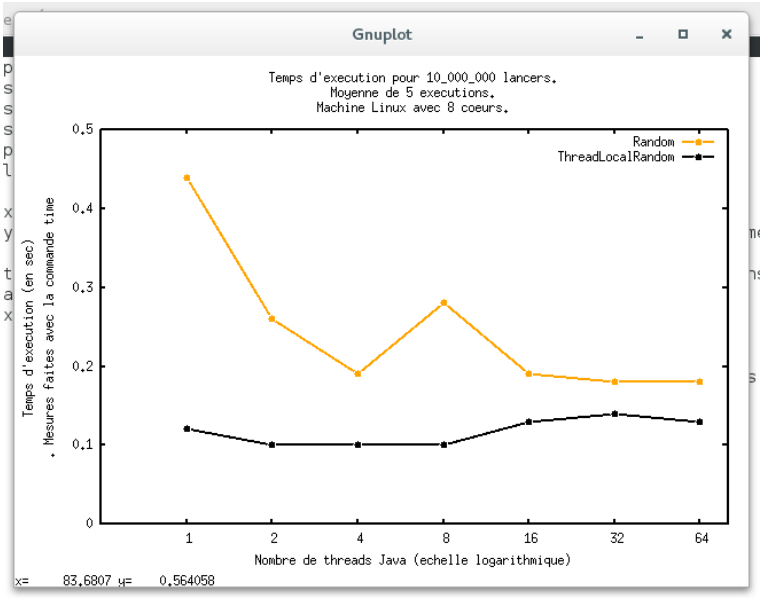



Figure 10.3: Graphe comparant les temps d'exécution (en sec.) du programme `Pi.java` en utilisant `Random` vs. `ThreadLocalRandom`.

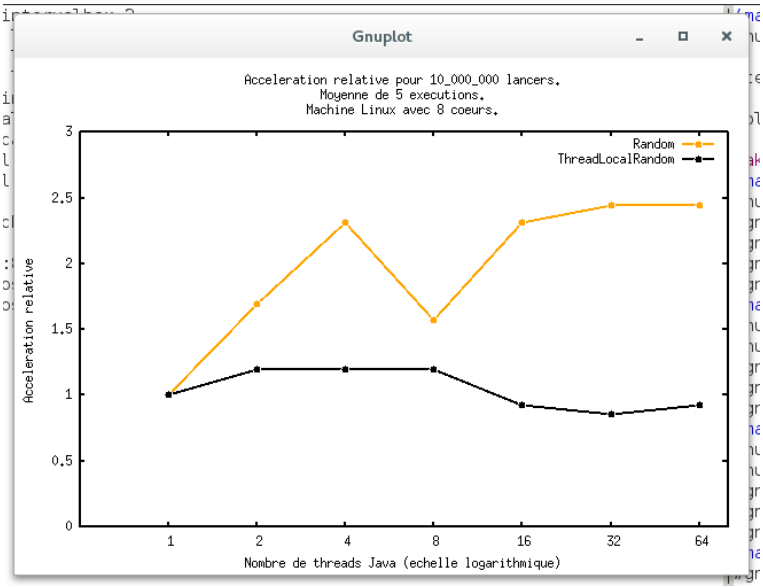



Figure 10.4: Graphe comparant les accélérations *relatives* du programme `Pi.java` en utilisant `Random` vs. `ThreadLocalRandom`.

## Analyse des performances : Mais il y a encore autre chose!

Un autre aspect important pour comprendre pourquoi les accélérations sont faibles est le suivant : les mesures de temps d'exécution présentées ci-haut ont été faites en utilisant la commande Unix `time`, au niveau du *shell*. En d'autres mots, le temps mesuré *inclue le temps pour le lancement de la machine virtuelle Java*. Or, ce temps est loin d'être négligeable — $\approx 0.03$ seconde, donc, dans le cas de la version `ThreadLocalRandom`, parfois presque 25 % du temps total d'exécution ☹

Les Figures 10.5 et 10.6 présentent les graphes comparant les temps d'exécution et les accélérations obtenus en utilisant `Random` et `ThreadLocalRandom`, mais cette fois en mesurant le temps *de l'intérieur du programme*, donc sans tenir compte du temps de lancement de la JVM.

Ces mesures sont effectuées telles qu'illustrées dans le segment de code suivant, et les temps sont donc en millisecondes :

```
long tempsDebut = System.currentTimeMillis();
double pi = evaluerPi( nbLancers, nbThreads );
long tempsFin = System.currentTimeMillis();

long tempsEcoule = tempsFin - tempsDebut;
```

On constate évidemment que les temps d'exécution avec l'utilisation de `ThreadLocalRandom` sont plus faibles qu'avec `Random`. Par contre, lorsqu'on ignore le temps de lancement de la JVM et qu'on ne mesure que le temps effectif consacré à la méthode de calcul de $\pi$, on constate que les accélérations sont plus intéressantes, et ce dans les deux cas.

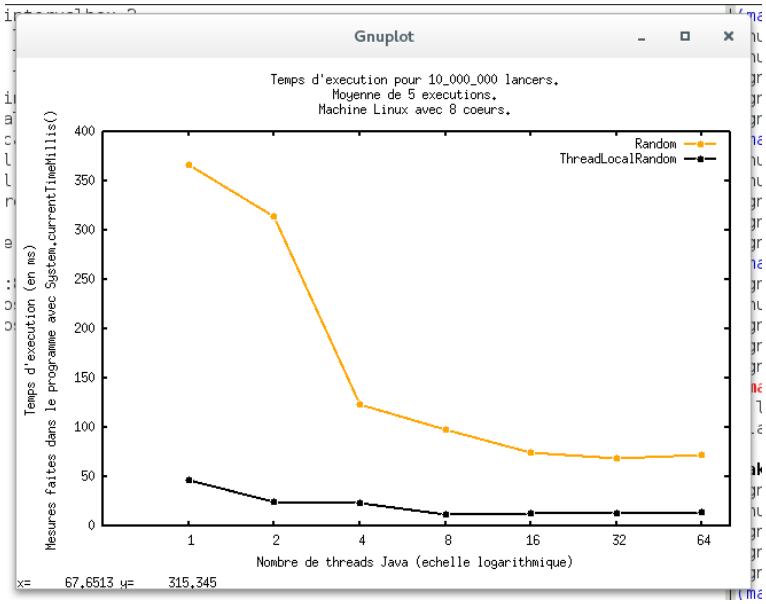



Figure 10.5: Graphe comparant les temps d'exécution du programme `Pi.java`, en ms, mesurés de l'intérieur du programme avec `System.currentTimeMillis()`, en utilisant `Random` vs. `ThreadLocalRandom`.

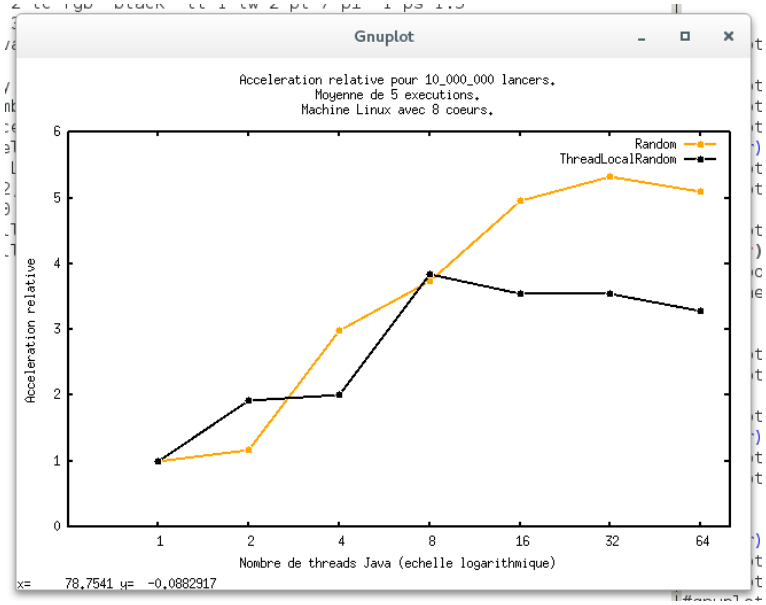



Figure 10.6: Graphe comparant les accélérations *relatives* du programme `Pi.java` en utilisant `Random` vs. `ThreadLocalRandom`, pour des temps mesurés de l'intérieur du programme.

Supposons un programme ayant le comportement suivant, où `S` indique une partie du programme qui ne peut s'exécuter que de façon *strictement séquentielle* et où `P` indique une partie qui peut s'exécuter de façon parallèle si des processeurs multiples sont disponibles :

```
S-S-P-P-P-P-P-P-P-P-S-S
```

Pour simplifier l'analyse qui suit, on suppose que *i*) chaque `S` ou `P` requiert le même temps et *ii*) les parties `P` peuvent être parallélisées de façon complète et aussi fine que nécessaire (*embarrassingly parallel*).†

1. Déterminez quelle sera l'accélération pour divers nombres de processeurs?
   (a) Quelle sera l'accélération pour 2 processeurs?
   (b) Quelle sera l'accélération pour 4 processeurs?
   (c) Quelle sera l'accélération pour 8 processeurs?
   (d) Quelle sera l'accélération pour 16 processeurs?
   (e) Quelle sera la meilleure accélération possible pour ce programme?

2. Supposons qu'on ait réussit à *optimiser* le programme initial et à réduire le temps d'exécution, *mais uniquement de la partie parallèle* et qu'on obtienne le comportement suivant :

   ```
   S-S-P-P-P-P-P-P-S-S
   ```

   (a) Quelle sera l'accélération pour 2 processeurs?
   (b) Quelle sera l'accélération pour 4 processeurs?
   (c) Quelle sera l'accélération pour 8 processeurs?
   (d) Quelle sera l'accélération pour 16 processeurs?
   (e) Quelle sera la meilleure accélération possible pour ce programme?

† Notez qu'une telle allure pour un programme parallèle est assez typique : le programme débute par une partie séquentielle (par ex., lecture des paramètres et données et initialisation), suivie d'une partie parallélisable, puis terminée par une partie séquentielle (par ex., combinaison des résultats intermédiaires ou écriture des résultats finaux).

**Exercice 10.14:** Effets des optimisations sur les accélérations.

## 10.6 Exemples des principaux patrons de programmation pour la somme de deux tableaux

Dans cette section, nous allons revoir divers patrons de programmation parallèle vus précédemment en PRuby, *mais cette fois exprimés en Java.*

Le problème (simple!) qui sera traité consistera à effectuer la somme de deux tableaux `a` et `b`, de même taille.

*Note :* Étant donné que nous utiliserons Java «de base» — donc les *packages* «standards» seulement, otamment `java.util.concurrent` — nous utiliserons donc toujours du parallélisme *fork–join*.

### 10.6.1 Version avec parallélisme récursif et seuil de récursion, avec *threads* Java et sans *threads* «inactifs»

Procédure auxiliaire :

```
private static void somme_seq_tranche( int a[], int b[],
                                       int c[],
                                       int bInf, int bSup ) {
  for ( int i = bInf; i <= bSup; i++ ) {
    c[i] = a[i] + b[i];
  }
}
```

**Programme Java 10.5** Parallélisme récursif avec création récursive d'un seul *thread*, l'autre sous-problème étant traité par le *thread* parent.

```
//
// Parallelisme recursif avec seuil utilisant du parallelisme
// fork-join avec un seul Thread.
//
public static void somme_par_rec1_ij( int a[], int b[],
                                      int c[],
                                      int i, int j,
                                      int seuil ) {
  if ( j - i + 1 <= seuil ) {
    somme_seq_tranche( a, b, c, i, j );
  } else {
    int mid = (i + j) / 2;
    Thread gauche = new Thread(
      () -> somme_par_rec1_ij( a, b, c, i, mid, seuil )
    );
    gauche.start();

    somme_par_rec1_ij( a, b, c, mid+1, j, seuil );

    try { gauche.join(); } catch( Exception e ){};
  }
}

public static int[] somme( int a[], int b[], int seuil ) {
  int n = a.length;
  int[] c = new int[n];

  somme_par_rec1_ij( a, b, c, 0, n-1, seuil );

  return c;
}
```

### 10.6.2 Version avec parallélisme embarrassant — à granularité fine

**Programme Java 10.6** Parallélisme à granularité (très!) fine avec `Threads` — un *thread* par position du tableau.

```
//
// Parallelisme style fork-join a granularite fine avec Threads.
//
public static int[] somme( int a[], int b[] ) {
  int n = a.length;
  int c[] = new int[n];

  Thread threads[] = new Thread[n];
  for( int i = 0; i < n; i++ ) {
    int fi = i;
    threads[i] = new Thread(
        () -> c[fi] = a[fi] + b[fi]
    );
    threads[i].start();
  }

  for( int i = 0; i < n; i++ ) {
    try { threads[i].join(); } catch( Exception e ){};
  }

  return c;
}
```

**Programme Java 10.7** Parallélisme à granularité (très!) fine avec des `Futures` — un *thread* par position du tableau.

```
//
// Parallelisme style fork-join a granularite fine avec Futures.
//
@SuppressWarnings("unchecked")
public static int[] somme( int a[], int b[] ) {
  int n = a.length;
  int c[] = new int[n];

  ExecutorService pool = Executors.newCachedThreadPool();
  Future<Integer>[] futures = new Future[n];
  for( int i = 0; i < n; i++ ) {
    int fi = i;
    futures[i] = pool.submit(
        () -> a[fi] + b[fi]
    );
  }

  for( int i = 0; i < n; i++ ) {
    try { c[i] = futures[i].get(); } catch( Exception e ){};
  }

  pool.shutdown();
  return c;
}
```

### 10.6.3 Version avec parallélisme à granularité grossière et attribution statique des tâches aux *threads*

**Programme Java 10.8** Parallélisme à granularité grossière avec répartition statique par blocs d'éléments adjacents.

```
//
// Parallelisme style fork-join a granularite grossiere avec
// Threads et repartition par blocs d'elements adjacents.
//
private static int inf( int i, int n, int nbThreads )
{ return i * (n / nbThreads); }

private static int sup( int i, int n, int nbThreads )
{ return (i+1) * (n / nbThreads) - 1; }

public static int[] somme( int a[], int b[], int nbThreads ) {
  assert a.length % nbThreads == 0;
  int n = a.length;
  int c[] = new int[n];

  Thread[] threads = new Thread[nbThreads];
  for( int k = 0; k < nbThreads; k++ ) {
    int bInf = inf(k, n, nbThreads);
    int bSup = sup(k, n, nbThreads);
    threads[k] = new Thread(
        () -> somme_seq_tranche( a, b, c, bInf, bSup )
    );
    threads[k].start();
  }

  for( int k = 0; k < nbThreads; k++ ) {
    try { threads[k].join(); } catch( Exception e ){};
  }

  return c;
}
```

**Programme Java 10.9** Parallélisme à granularité grossière avec répartition statique et cyclique des éléments.

```
//
// Parallelisme style fork-join a granularite grossiere avec
// Threads et repartition cyclique des elements.
//
private static void somme_seq_cyclique( int a[], int b[],
                                        int c[],
                                        int bInf,
                                        int nbThreads ) {
  for ( int i = bInf; i < c.length; i += nbThreads ) {
    c[i] = a[i] + b[i];
  }
}

public static int[] somme( int a[], int b[],
                           int nbThreads ) {
  int n = a.length;
  int c[] = new int[n];

  Thread[] threads = new Thread[nbThreads];
  for( int k = 0; k < nbThreads; k++ ) {
    int fk = k;
    threads[k] = new Thread(
        () -> somme_seq_cyclique( a, b, c, fk, nbThreads )
    );
    threads[k].start();
  }

  for( int k = 0; k < nbThreads; k++ ) {
    try { threads[k].join(); } catch( Exception e ){};
  }

  return c;
}
```

### 10.6.4 Version avec parallélisme à granularité grossière et attribution dynamique des tâches aux *threads*

**Programme Java 10.10** Parallélisme à granularité grossière avec attribution dynamique par blocs d'éléments adjacents et *pool de threads «standard»* (`newFixedThreadPool`).

```
//
// Parallelisme style Coordonnateur-Travailleurs -- donc avec une
// repartition dynamique des elements -- et utilisant au plus
// nbTravailleurs threads.
//
public static int[] somme( int a[], int b[],
                                    int tailleTache,
                                    int nbTravailleurs ) {
  assert a.length % tailleTache == 0;

  int n = a.length;
  int[] c = new int[n];

  ExecutorService pool
    = Executors.newFixedThreadPool(nbTravailleurs);

  int nbTaches = n / tailleTache;
  Future<?>[] futures = new Future[nbTaches];
  for( int k = 0; k < nbTaches; k++ ) {
    int fi = k * tailleTache;
    int fj = fi + tailleTache - 1;
    futures[k] = pool.submit(
       () -> somme_seq_tranche( a, b, c, fi, fj )
    );
  }

  // On s'assure que chaque tache a termine.
  for( int k = 0; k < nbTaches; k++ ) {
    try { futures[k].get(); } catch( Exception e ){};
  }
  pool.shutdown();

  return c;
}
```

Description de `newFixedThreadPool` :

> `public static`
> `ExecutorService newFixedThreadPool(int nThreads)`
>
> *Creates a thread pool that* **reuses a fixed number of threads operating off a shared unbounded queue**.
>
> *At any point, at most* `nThreads` *threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. [...]*
>
> *The threads in the pool will exist until it is explicitly shutdown.*

**Programme Java 10.10** Parallélisme à granularité grossière avec attribution dynamique par blocs d'éléments adjacents et *pool de threads «standard»* (`newFixedThreadPool`), avec une autre approche de terminaison.

```
//
// Parallelisme style Coordonnateur-Travailleurs -- donc avec une
// repartition dynamique des elements -- et utilisant au plus
// nbTravailleurs threads, mais avec une facon differente de
// terminer l'execution.
//
public static int[] somme( int a[], int b[],
                                    int tailleTache,
                                    int nbTravailleurs ) {
  int n = a.length;
  int[] c = new int[n];

  ExecutorService pool
    = Executors.newFixedThreadPool(nbTravailleurs);

  int nbTaches = n / tailleTache;
  Future<?>[] futures = new Future[nbTaches];
  for( int k = 0; k < nbTaches; k++ ) {
    int fi = k * tailleTache;
    int fj = fi + tailleTache - 1;
    futures[k] = pool.submit(
       () -> somme_seq_tranche( a, b, c, fi, fj )
    );
  }

  pool.shutdown();
  try {
    pool.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
  } catch (InterruptedException e) {
  }
  assert pool.isTerminated();
  return c;
}
```

**Programme Java 10.11** Parallélisme à granularité grossière avec attribution dynamique par blocs d'éléments adjacents et *pool de threads hyper-légers* (`ForkJoinPool`).

```
//
// Parallelisme style Coordonnateur-Travailleurs -- donc avec une
// repartition dynamique des elements -- et utilisant au plus
// nbTravailleurs threads.
//
public static int[] somme( int a[], int b[],
                                    int tailleTache,
                                    int nbTravailleurs ) {
  assert a.length % tailleTache == 0;

  int n = a.length;
  int[] c = new int[n];

  ForkJoinPool pool = new ForkJoinPool(nbTravailleurs);

  int nbTaches = n / tailleTache;
  Future<?>[] futures = new Future[nbTaches];
  for( int k = 0; k < nbTaches; k++ ) {
    int fi = k * tailleTache;
    int fj = fi + tailleTache - 1;
    futures[k] = pool.submit(
       () -> somme_seq_tranche( a, b, c, fi, fj )
    );
  }

  for( int k = 0; k < nbTaches; k++ ) {
    try { futures[k].get(); } catch( Exception e ){};
  }

  pool.shutdown();
  return c;
}
```

| | tailleTache | | | |
|---|---|---|---|---|
| | **1000** | **100** | **10** | **1** |
| `newFixedThreadPool` | 0.14 | 0.18 | 0.77 | 8.52 |
| `ForkJoinPool` | 0.13 | 0.17 | 0.37 | 4.45 |

Tableau 10.1: Comparaison des temps d'exécution de deux versions de la méthode `somme` : avec `newFixedThreadPool` comparé avec `ForkJoinPool`.

Description de `ForkJoinTask` et `ForkJoinPool` :

- `ForkJoinTask` :

  > *A* `ForkJoinTask` *is a thread-like entity that is* much lighter weight than a normal thread*.*
  >
  > *Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a* `ForkJoinPool`*, at the price of some usage limitations.*
  >
  > *https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html*

- `ForkJoinPool` :

  > *An* `ExecutorService` *for running* `ForkJoinTasks`*. [...]*
  >
  > *A* `ForkJoinPool` *differs from other kinds of* `ExecutorService` *mainly by virtue of employing* **work-stealing***: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most* `ForkJoinTasks`*), as well as when many small tasks are submitted to the pool from external clients.*
  >
  > *https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html*

Les temps d'exécution du tableau 10.1 ont été obtenus avec «`time -p`», pour des tableaux comportant $n$ =10 000 000 d'entiers, avec 8 *threads* (`nbTravailleurs`) sur une machine avec 8 coeurs (Linux CentOS), et ce pour différentes valeurs de `tailleTache`.

**Question** : Temps séquentiel? Temps par blocs d'éléments adjacents?

## 10.7 Les objets comme moniteurs

### 10.7.1 Verrous et exclusion mutuelle

À tout objet Java est automatiquement associé un verrou et une variable de condition. Donc, tout objet Java est automatiquement un moniteur. Ainsi, soit la classe Java suivante :

```
class Compteur {
  private int val = 0;

  public void inc() {
    val += 1;
  }

  public int valeur() {
    return val;
  }
}
```

Si deux *threads* distincts font un appel à `inc()`, alors il pourrait y avoir interférence entre les deux *threads*. Pour éviter les situations de compétition, il faut donc assurer que l'exécution de `inc()` puisse se faire de façon atomique et exclusive. La façon la plus simple consiste à annoter la méthode comme étant `synchronized` :

```
class Compteur {
  private int val = 0;

  public synchronized void inc() {
    val += 1;
  }

  public int valeur() {
    return val;
  }
}
```

Est-il nécessaire ou approprié que la méthode `valeur` soit aussi `synchronized`?

**Exercice 10.1:** Méthode `valeur` : `synchronized` ou pas?

Le mot-clé `synchronized` peut donc être utilisée de deux façons possibles :

- Dans la signature d'une méthode, de façon à indiquer que la méthode doit être exécutée de façon exclusive et atomique *relativement aux autres méthodes* `synchronized` *du même objet — donc comme une méthode de moniteur.*

- Comme une instruction explicite, dont la forme générale est la suivante :

```
synchronized( <objetQuelconque> ) {
  <instructions à exécuter de façon exclusive>
}
```

En termes d'effets, la solution qui suit est équivalente à la précédente, bien que la précédente soit plus *explicite* quant au comportement de la méthode (pas besoin de regarder sa mise en oeuvre pour voir qu'il y a exclusion mutuelle) :

```
class Compteur {
  private int val = 0;

  public void inc() {
    synchronized(this) {
      val += 1;
    }
  }

  public int valeur() {
    return val;
  }
}
```

Il est important de comprendre *qu'à tout objet est implicitement associé un verrou* utilisable par une instruction `synchronized`.

Est-ce que le comportement des deux classes suivantes diffère? Si oui de quelle façon?

```
class Compteur {
  private int val1 = 0, val2 = 0;

  public synchronized void inc1() {
    val1 += 1;
  }

  public synchronized void inc2() {
    val2 += 1;
  }
}

class Compteur {
  private int val1 = 0,
              val2 = 0;

  private Object v1 = new Object(),
                 v2 = new Object();

  public void inc1() {
    synchronized(v1) { val1 += 1; }
  }

  public void inc2() {
    synchronized(v2) { val2 += 1; }
  }
}
```

**Exercice 10.2:** Différences entre deux classes `Compteur`?

### 10.7.2 Variables de condition

De la même façon qu'à tout objet est implicitement associé un verrou, à tout objet est aussi implicitement associée une variable de condition. Trois méthodes de base peuvent être utilisées sur de telles variables de condition :

- `wait()` : Mise en attente inconditionnelle.
- `notify()` : Envoi d'un signal à un des *threads* en attente.
- `notifyAll()` : Envoi d'un signal à tous les *threads* en attente.

**Programme Java 10.12** Moniteur Java pour une classe `Semaphore`.

```
public class Semaphore {
  private int val;

  public Semaphore( int valInitiale ) {
    val = valInitiale;
  }

  public synchronized void P() {
    while ( val == 0 ) {
        try { wait(); }
        catch ( Exception e ) { ... }
    }
    val -= 1;
  }

  public synchronized void V() {
    val += 1;
    notify();
  }
}
```

Le Programme Java 10.12 présente une classe Java appelée `Semaphore` réalisée sous forme d'un moniteur. Quelques remarques concernant cet exemple :

- Les appels `wait()` et `notify()` manipulent la variable de condition implicitement associée à l'objet courant, puisque ces appels sont équivalents aux appels `this.wait()` et `this.notify()`.

  Si le moniteur requiert plusieurs variables de condition, on peut alors déclarer et allouer plusieurs objets quelconque ou, encore mieux, utiliser des objets qui satisfont l'interface `Condition` (voir section 10.A.3, p. 70).

- Un appel à la méthode `wait()` doit toujours être indiqué à l'intérieur d'une instruction `try/catch`, puisqu'il est possible que l'exception `InterruptedException` soit signalée (exception qui indique que le *thread* a été interrompu et doit terminer son exécution).

- Pour qu'un appel à la méthode `wait()`, `notify()` ou `notifyAll()` soit correct, il doit être toujours être fait alors que le *thread* qui appelle la méthode est celui qui a pris le contrôle du verrou.

1. Que fait un sémaphore?
2. Que font les opérations `P()` et `V()`?

**Exercice 10.3:** Que fait un sémaphore?

Le programme Java 10.13 illustre des exemples d'appels faits sans que le *thread* appelant soit en possession du verrou.

**Programme Java 10.13** Extrait de programme Java qui illustre les erreurs signalées par des appels à `wait()` ou `notify()` sans que le *thread* appelant soit en possession du verrou.

```
Object verrou = new Object();

public void foo() {
  verrou.notify()
}

=>

Exception in thread "Thread-0"
  java.lang.IllegalMonitorStateException


-----


public void foo() {
  try { wait(); } catch ( Exception e ) { ... }
}

=>

Exception in thread "Thread-0"
  java.lang.IllegalMonitorStateException
```

## 10.8 Interruption d'un *thread*

- Dans un programme avec plusieurs *threads*, l'avortement (*cancellation*) d'un *thread* peut être nécessaire dans certains cas, par exemple :
  - Lorsque l'usager avorte l'exécution d'une action (bouton `<cancel>`, touche `^C`, etc.).
  - Lorsque plusieurs *threads* sont lancés en parallèle pour trouver un résultat et que leur exécution peut se terminer parce que le résultat a été trouvé par l'un des *threads*, le travail des autres n'étant plus nécessaire.
  - Lorsqu'un problème a été rencontré et que l'exécution doit être avortée.
- Chaque *thread* possède un *flag* indiquant s'il a été interrompu ou non.
- `t.isInterrupted()` permet d'examiner le statut du flag du *thread* `t`, mais sans le resetter. Plus précisément, retourne `true` si le *flag* a été setté par `t.interrupt()` et n'a pas été resetté depuis par l'utilisation de `interrupted()` ou par l'exécution d'un `wait()`, `sleep()` ou `join()`.
- `Thread.interrupted()` met le *flag* **du *thread* courant** à `false` et retourne l'ancienne valeur (ne peut pas s'utiliser sur un autre *thread*).
- `t.interrupt()` : met le *flag* du *thread* `t` à true, sauf si `t` est engagé dans un `wait()`, `sleep()` ou `join()`. Dans ce cas, l'appel à `interrupt()` a plutôt pour effet que l'action interrompue va lancer l'exception `InterruptedException`.

  Ceci n'implique pas que le *thread* termine immédiatement son exécution. Entre autres, il peut être nécessaire pour le *thread* de faire un peu de nettoyage avant de vraiment se terminer. C'est au *thread* lui-même à déterminer à quelle fréquence il va examiner son *flag* et déterminer ce qu'il doit faire.

  Toutefois, une vérification du statut d'interruption est effectuée *automatiquement* lors de l'exécution des méthodes `Object.wait()`, `Thread.join()` et `Thread.sleep()`. Si le flag a été setté, alors l'exception `InterruptedException` est lancée (ce qui a alors pour effet de désactiver le *flag*).

  Autre caractéristique importante : un *thread* en attente pour l'accès à un verrou (appel d'une méthode `synchronized`) ne réagira pas à l'interruption tant qu'il sera bloqué.

## 10.9 Priorités des *threads*

- Le niveau de priorité d'un *thread* est une valeur entière comprise entre `Thread.MIN_PRIORITY` (1) et `Thread.MAX_PRIORITY` (10)
- Par défaut, un thread a la même priorité que son parent (priorité de `main` = `NORM_PRIORITY` = 5).
- Lecture et modification du niveau de priorité :

  ```
  int getPriority()
  void setPriority(int priority)
  ```

  Le maximum permis dépend du groupe de *threads* duquel le *thread* fait partie.
- Convention pour les niveaux de priorités :
  - 10 : Crise
  - 7–9 : Processus interactif, gestion d'événements
  - 4–6 : *I/O-bound*
  - 2–3 : Calcul en arrière plan
  - 1 : À exécuter seulement s'il n'y a rien d'autres à faire
- La politique exacte d'ordonnancement en fonction du niveau de priorité est «*implementation dependant*».

  De façon générale, la machine virtuelle va exécuter en premier les *threads* avec un niveau de priorité plus élevé.

  Toutefoisil faut aussi tenir compte de deux aspects importants :
  - Inversion de priorité :

    Supposons qu'un *thread* `t1` tente d'acquérir un verrou qui est actuellement sous le contrôle d'un *thread* `t2` de priorité plus faible. C'est donc comme si le *thread* `t1`, qui est maintenant bloqué en attente que `t2` libère le verrou, s'exécutait au niveau de priorité de `t2` ⇒ inversion de priorité.

    Solution : héritage de priorité ⇒ si un *thread* `t2` contrôle un verrou qui est requis par un *thread* `t1` de plus haut niveau de priorité, alors le niveau de priorité de `t2` est temporairement haussé au niveau de priorité de `t1`.
  - Niveaux de priorité de Java vs. niveaux de priorité du système d'exploitation :

    Règle générale, la priorité associée à un *thread* du SE n'est pas nécessairement identique à celle de la machine virtuelle Java. Pour éviter la «famine» (un *thread* qui ne s'exécute jamais parce tous les autres *threads* passent avant lui), la priorité d'un *thread* du SE est généralement définie par une formule ayant l'allure suivante :

```
<Vrai niveau de priorité>
   = <Niveau de priorité Java>
     +
     <Nombre de "secondes" en attente de l'UCT>
```

Donc, ceci signifie que l'on ne peut pas se baser strictement sur le niveau de priorité Java pour assurer un ordonnancement spécifique des *threads*.

## 10.10 Quelques autres méthodes de la classe `Thread`

- `Thread.currentThread()` : retourne une référence au *thread* courant (celui qui exécute l'appel de cette méthode).
- `t.sleep( long milis )` : endort le *thread* pour la durée indiquée.
- Méthodes *deprecated* : `suspend()`, `resume()`, `stop()`, `destroy()`.

**Note :** La classe `Thread` possède un (très!) grand nombre de constantes, constructeurs et méthodes :

- Constantes : `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY` ;
- Constructeurs : Huit (8) variantes différentes (possibilité d'indiquer le nom, le groupe, un `Runnable`, etc.).
- Méthodes : $\approx$ 50 méthodes!

## 10.11 Attribut `volatile`

- En Java, la lecture/écriture d'une variable **simple** (sauf pour les `long` et `double`) s'effectue de façon atomique et indivisible. Une variable simple peut donc être utilisée par deux *threads* pour s'échanger une valeur via lecture/écriture sans danger d'interférence (donc sans danger qu'il n'y ait qu'une lecture ou écriture partielle).

  Mais : le modèle de mémoire de Java permet à un *thread* de conserver dans sa mémoire locale (e.g., dans un registre de la machine) une copie de la variable (cache mémoire). Donc, si un *thread* dépend d'une valeur écrite par un autre *thread*, *il pourrait ne pas voir l'effet de l'écriture.*

- Par exemple, soit les deux méthodes suivantes définies dans une même classe — l'effet de l'écriture suite à un appel à `signalerFin()` (par un autre *thread*) pourrait ne pas être visible dans `traiter()` :

```
public class Exemple {
  ...
  private boolean termine = false;

  public void traiter() {
    while( !termine ) {
      ... faire quelque chose ...
    }
  }

  public void signalerFin() {
    termine = true;
  }
}
```

- Solution = ajouter l'attribut `volatile` :

```
private volatile boolean termine = false;
```

  Effet de `volatile` = à chaque lecture ou écriture d'une telle variable, un vrai accès à la mémoire sera effectué, plutôt qu'un accès au cache ou aux registres locaux.

  L'attribut `volatile` peut donc être interprété comme une indication à la machine virtuelle qu'elle ne doit pas faire de copie dans l'espace local du *thread.*

- Autre solution, **plus coûteuse** = protéger l'accès par un verrou :

```
private boolean termine = false;
...
public void traiter() {
    boolean termine;
    synchronized(this) {
      termine = this.termine;
    };
    while( !termine ) {
      ... faire quelque chose ...
      synchronized(this) {
        termine = this.termine;
      };
    }
 ...
}
...
public synchronized void signalerFin() {
  ...
}
```

Lors de l'accès à un verrou, toutes les copies locales des variables du *thread* sont mises à jour à partir de la mémoire. Lors de la libération du verrou, toutes les variables modifiées sont copiées en mémoire.

Citation de `http://www.javaperformancetuning.com/news/qotm030.shtml` :

> *So where* `volatile` *only synchronizes the value of one variable between thread memory and "main" memory,* `synchronized` *synchronizes the value of all variables between thread memory and "main" memory [...]. Clearly synchronized is likely to have more overhead than volatile.*

Est-il nécessaire ou approprié que la méthode `valeur` ci-bas soit `synchronized`?

```
class Compteur {
  private int val = 0;

  public synchronized void inc() {
    val += 1;
  }

  public int valeur() {
    return val;
  }
}
```

**Exercice 10.4:** Méthode `valeur` : `synchronized` ou pas?

*Autres informations sur les variables* `volatile`

> *For the purposes of the Java programming language memory model,* **a single write to a non-volatile long or double value is treated as two separate writes***: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.*
>
> Writes and reads of volatile long and double values are always atomic.
>
> Writes to and reads of references are always atomic*, regardless of whether they are implemented as 32-bit or 64-bit values.*
>
> *Source : https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html*

*Le modèle mémoire de Java et les situations de compétition*

> *A* memory model *describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. The Java programming language memory model works by examining each read in an execution trace and checking that the write observed by that read is valid according to certain rules.*
>
> The memory model describes possible behaviors of a program*. An implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model.*
>
> *Source : https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html*

Qu'est-ce qui sera imprimé par le programme ci-bas?

```
class MemoryModel {
  static int x = 0,
             y = 0,
             r1,
             r2;

  public static void main( String[] args ) {
    Thread t1 = new Thread( () -> {
        r1 = x;  // I1
        y = 1;   // I2
      } );
    Thread t2 = new Thread( () -> {
        r2 = y;  // I3
        x = 2;   // I4
      } );
    t1.start(); t2.start();

    try {
      t1.join(); t2.join();
    } catch( Exception e ){}

    System.out.println( "r1 = " + r1 +
                        "; " +
                        "r2 = " + r2 );
  }
}
```

**Exercice 10.5:** Exemple spécial illustrant le modèle de mémoire de Java.

## 10.12 Traitement des exceptions

Si durant l'exécution d'un *thread* une exception est signalée (`throw`) et que cette exception se propage au-delà de la méthode `run`, alors l'exécution du *thread* est considérée comme s'étant terminée. Par défaut, une exception qui est ainsi propagée sans être traitée dans le contexte du *thread* est traitée par la méthode `uncaughtException()` associée au groupe de *threads* (classe `ThreadGroup`) duquel le *thread* fait partie :

```
void uncaughtException(Thread t, Throwable e)
// Called by the Java Virtual Machine when
// a thread in this thread group stops
// because of an uncaught exception.
```

Par défaut, cette méthode imprime la trace de la pile d'activation. Cette méthode peut aussi être redéfinie.

## 10.13 `Streams` de Java 8.0

Déjà depuis sa version 2.0, Java fournit une riche bibliothèque de collections (`java.util.Collection`) :

- Collections ordonnées : interface `List` et classes `ArrayList`, `LinkedList`;
- Dictionnaires : interface `Map` et classes `HashMap`, `TreeMap`;
- Ensembles : interface `Set` et classes `HashSet`, `LinkedHashSet` et `TreeSet`.

La version 8.0 de Java a introduit une nouvelle forme de collections, les *streams* — l'interface `Stream` (et diverses classes concrètes) de `java.util.stream`. Une caractéristique importante de ces *streams* est que l'API est de style *fonctionnel*, semblable comme on le verra aux collections `Enumerable` de Ruby et aux `Streams` de PRuby — notamment quant à l'utilisation du *chainage de méthodes* pour exprimer des suites complexes d'opérations.

Une autre caractéristique des *streams* de Java 8.0 est qu'ils peuvent être traités tant de façon séquentielle que parallèle, le passage d'une forme à une autre se faisant par un simple appel de méthode — `parallel()` — donc sans avoir à créer ou manipuler explicitement des *threads* ou tâches.

Nous avons présenté à la section **??** des exemples illustrant les *streams* disponibles en PRuby, qui permettent d'exprimer le parallélisme de flux, et ce sans manipulation explicite de canaux de communication. Comme on le verra dans les exemples Java qui suivent, l'API PRuby pour les *streams* est semblable à celle de Java 8.0, à la différence que les *streams* de PRuby *sont toujours parallèles*, ce qui n'est pas le cas pour les *streams* de Java 8.0. On verra aussi une autre différence, à savoir qu'en Java, il faut parfois (souvent?) alterner entre collections et *streams*, puisque certaines méthodes clés sur les *streams* retournent des collections, et non des *streams*.

### 10.13.1 Tri unique des mots d'un fichier

Le Programme Java 10.14 présente une méthode qui, à partir d'un fichier de texte, produit un fichier de texte contenant les divers mots du fichier d'entrée, en ordre et sans répétition — donc comme l'exemple présenté dans le Programme PRuby **??** :

- Un *stream* source est créé à partir du contenu du fichier à l'aide de la méthode `Files.lines`.
- Par défaut, ce *stream* est traité de façon séquentielle. L'appel à `parallel()` amorce le traitement en parallèle du *stream*.
- Les appels aux méthodes `flatMap`, `filter`, `sorted` et `distinct` sont semblables aux appels équivalents du Programme Ruby **??** — la méthode Ruby équivalente à la méthode `distinct()` de Java est `uniq`.
- La méthode `forEachOrdered` traite chacune des mots produit et l'ajoute dans le fichier avec un saut de ligne — fichier `fichSortie` ouvert en écriture au début de la méthode.

**Programme Java 10.14** Version Java 8.0 d'un programme pour effectuer le tri unique des mots d'un fichier, version semblable le plus possible à la version PRuby (Programme PRuby **??**).

```
public static void trierMotsUniques( String fichEntree,
                                     String fichSortie ) {
  try {
    BufferedWriter bw = Files.newBufferedWriter(Paths.get(fichSortie));

    Files.lines( Paths.get(fichEntree) )
      .parallel()  // Si omis => traitement sequentiel!
      .flatMap( ligne -> Stream.of( ligne.split( " " ) ) )
      .filter( mot -> Pattern.compile("^\\w+").matcher(mot).find() )
      .sorted()
      .distinct()
      .forEachOrdered( mot ->
                       { try { bw.write(mot); bw.newLine(); }
                         catch(IOException ioe) { /*...*/ }; }
    );

    bw.close();
  } catch(IOException ioe) { /*...*/ }
}
```

### 10.13.2 Dénombrement des mots d'un texte

Le Programme Java 10.15 présente une méthode qui, à partir d'un tableau de `lignes` de texte, produit une liste ordonnée des divers mots (suite de caractères non blancs) trouvés dans les `lignes` avec le nombre d'occurrences associé à chaque mot. Ici, il s'agit d'une version la plus semblable possible à l'exemple présenté dans le Programme PRuby **??**. Par contre, le Programme Java 10.16 présente une autre version de cette méthode, exprimée de façon un peu plus simple en utilisant un `Collector` plus complexe.

- Alors qu'en Ruby on pouvait utiliser simplement un `Array` de taille 2 pour représenter un tuple (une paire), en Java on utilise un `SimpleEntry` — une classe concrète qui met en oeuvre l'interface `Map.Entry`, avec les méthodes `getKey()` et `getValue()` pour obtenir les éléments du tuple.

- Dans la version Ruby, tous les étages du pipeline étaient des *streams*, y compris le résultat produit par `group_by_key`. En Java, la méthode `collect`, qui permet notamment de faire des regroupements, génère une `Collection` à partir d'un *stream* — par ex., une `List` ou un `Map` selon le type d'opération de collecte. Si on veut par la suite continuer à utiliser des méthodes de *streams*, exprimées avec chaînage de méthodes, alors on doit *reconvertir* la collection en un *stream* en utilisant `stream()`.

**Programme Java 10.15** Version Java 8.0 d'un programme pour dénombrer les occurrences des mots d'un texte, version semblable le plus possible à la version PRuby (Programme PRuby **??**).

```
public static List<Map.Entry<String,Integer>>
              compterMots( String[] lignes ) {
  return
    Arrays.stream( lignes )
      .parallel()
      .flatMap( ligne -> Stream.of( ligne.split( " " ) ) )
      .map( mot -> new SimpleEntry<String,Integer>(mot, 1) )
      .collect( Collectors.groupingBy(Map.Entry::getKey) )
      .entrySet()
      .stream()
      .map( e -> new SimpleEntry<>( e.getKey(),
                                    e.getValue()
                                     .stream()
                                     .map(Map.Entry::getValue)
                                     .reduce(0, Integer::sum)) )
      .sorted( Comparator.comparing(Map.Entry::getKey) )
      .collect( Collectors.toList() );
}
```

**Programme Java 10.16** Autre version Java 8.0 d'un programme pour dénombrer les occurrences des mots d'un texte, version plus simple pour Java 8.

```
public static List<Map.Entry<String,Integer>>
              compterMots( String[] lignes ) {
  return
    Arrays.stream( lignes )
      .parallel()
      .flatMap( ligne -> Stream.of( ligne.split( " " ) ) )
      .map( mot -> new SimpleEntry<String,Integer>(mot, 1) )
      .collect( Collectors
                  .groupingBy(Map.Entry::getKey,
                              Collectors.summingInt(Map.Entry::getValue)) )
      .entrySet()
      .stream()
      .sorted( Comparator.comparing(Map.Entry::getKey) )
      .collect( Collectors.toList() );
}
```

### 10.13.3 Génération des `n` premiers nombres de Fibonacci

Une caractéristique importante des *streams* Java est que les éléments d'un *stream* sont générés *de façon paresseuse* — en d'autres mots, seuls les éléments requis *pour produire le résultat final* sont effectivement générés. Par contre, les *streams* de PRuby sont générés de façon non-paresseuse — donc de façon zêlée, impatiente, *eager* en anglais. Un *stream* Java pourrait donc — *possiblement* ou *potentiellement* — générer une série sans fin d'éléments, ce qui ne créera aucun problème de terminaison du programme si seule une partie *finie* de ces éléments est effectivement utilisée. Par contre, un *stream* PRuby qui génère une série sans fin d'éléments ne se terminera pas — ou plus précisément, il pourrait se terminer, mais il pourrait encore y avoir un *thread* actif, soit celui qui génère la série sans fin d'éléments.

Un exemple est présenté dans le Programme Java 10.17, qui génère la liste des `n` premiers nombres de Fibonacci :

- La méthode `Stream.iterate` produit un *stream*, potentiellement infini, de paires (tableaux de taille 2) de nombres, lesquels représentent des nombres adjacents dans la suite de Fibonacci.
- La méthode `map` transforme le *stream* de paires en un *stream* d'entiers, en ne conservant que le premier champ de chaque paire.
- La méthode `limit( n )` est utilisée pour limiter le *stream* produit aux `n` premiers éléments — en Ruby/PRuby, on aurait utilisé `take( n )`.
- Finalement, le *stream* est transformé en une `List` en utilisant la méthode `collect` avec le `Collectors.toList()`.

**Programme Java 10.17** Programme Java pour générer les `n` premiers nombres de Fibonacci avec un *stream* potentiellement infini.

```
public static List<Integer> fibo( int n ) {
  return
    Stream.iterate( new int[]{0, 1},
                    p -> new int[]{ p[1], p[0] + p[1] } )
    .map( p -> p[0] )
    .limit( n )
    .collect( Collectors.toList() );
}
```

De façon plus générale, la méthode `iterate` reçoit deux arguments :

1. Le premier élément qui doit être produit par le *stream*.

   Dans l'exemple ci-haut, la paire {0, 1}.

2. Une lambda-expression qui indique comment générer *le prochain élément* du *stream* à partir de l'élément courant.

   Dans l'exemple ci-haut, à partir d'une paire `p` on génère une nouvelle paire dont le $1^{er}$ élément est `p[1]` et le $2^{e}$ élément est `p[0] + p[1]`.

Voici quelques exemples simples de *streams* potentiellement infinis d'entiers :

- `Stream.iterate(0, n -> n + 1)` : tous les nombres entiers ;
- `Stream.iterate(0, n -> n + 2)` : tous les nombres pairs ;
- `Stream.iterate(1, n -> 2 * n)` : toutes les puissances de 2 ;

Quant au comportement paresseux des *streams*, on peut l'observer en examinant les paires de nombres générées par le Programme Java 10.17 en ajoutant l'appel suivant à `peek` tout de suite après l'appel à `iterate` :[1]

```
    .peek( p -> System.out.println( p[0] + ", " + p[1] ) )
```

Soit alors le segment de code suivant :

```
for( Integer n: fibo( 5 ) ) {
  System.out.println( n );
}
```

La sortie suivante serait alors produite pour ce segment de code avec `fibo` modifié par l'ajout de `peek` :

```
0, 1
1, 1
1, 2
2, 3
3, 5
0
1
1
2
3
```

[1] La méthode `peek` «*Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.*». Source : `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`

Le Programme Ruby 10.1 présente une version Ruby semblable à celle du Programme Java 10.17 — avec l'ajout de l'appel à `peek` pour illustrer les paires qui sont générées.

**Programme Ruby 10.1** Programme Ruby pour générer les `n` premiers nombres de Fibonacci avec un *stream* potentiellement infini.

```
def fibo( n )
  PRuby::Stream.generate([0, 1]) { |p0, p1| [p1, p0 + p1] }
    .peek { |p0, p1| print "#{p0}, #{p1}\n" }
    .map(&:first)
    .take( n )
    .to_a
end
```

La sortie produite par l'appel «`puts fibo( 5 )`» est présentée plus bas : on constate que plusieurs paires d'éléments sont générées *en plus des cinq paires requises pour produire le résultat*, contrairement à la version Java.

```
0, 1
1, 1
1, 2
2, 3
3, 5
5, 8
8, 13
13, 21
21, 34
34, 55
55, 89
89, 144
144, 233
233, 377
377, 610
610, 987
987, 1597
1597, 2584
2584, 4181
4181, 6765
6765, 10946
10946, 17711
17711, 28657
28657, 46368
46368, 75025
75025, 121393
121393, 196418
0
1
1
2
3
```

### 10.13.4 Éxécution séquentielle vs. parallèle des *streams* Java

Tel que mentionné précédemment, pour que les éléments d'un *stream* soient traités de façon parallèle, il suffit d'effectuer un appel à `parallel()` sur le *stream*. Inversement, si le traitement est parallèle, on peut le rendre séquentiel avec un appel à `sequential()`.

Les *streams* parallèles utilisent le `ForkJoinPool` (cf. Fig. 10.7) par défaut pour créer les *threads* qui exécuteront les opérations. Par défaut, le nombre de *threads* associés à ce *pool* est égal au nombre de coeurs de la machine — `Runtime.getRuntime()-.availableProcessors()`. Pour changer ce nombre de *threads*, c'est le nombre de *threads* de ce *pool* par défaut qu'il faut modifier ; l'effet est donc *global pour tous les streams*.

Lors d'un traitement parallèle, il faut aussi tenir compte que certaines collections se prêtent bien à la génération d'un *stream* parallèle, parce que ces collections se décomposent facilement en morceaux indépendants, alors que d'autres s'y prêtent moins bien :

| | |
|---|---|
| **Excellente** | `ArrayList` |
| **Bonne** | `HashSet`, `TreeSet` |
| **Mauvaise** | `LinkedList`, `Stream.iterate` |

De plus, les opérations qui dépendent de l'ordre des éléments ne s'exécuteront pas de façon efficace sur un *stream* parallèle, par exemple, `limit`, `findFirst`

Lorsqu'on désire utiliser des *streams*, il faut aussi tenir compte que les surcoûts associés à la création et manipulation des *streams* parallèles, bien qu'ils utilisent les *threads* léger du `ForkJoinPool`, sont non négligeables. Pour qu'un programme avec des *streams* parallèles soit plus rapide qu'avec des *streams* séquentiels, il faut que la quantité de données à traiter soit «assez grande».

Finalement, les *streams* ne devraient être utilisés que pour des traitements qui sont *compute-bound* — en fait, pour des traitements qui ne font pas d'opérations d'entrée/sortie (accès fichiers, connexions réseau, etc.), *sauf évidemment pour la génération initiale du* stream *et la collecte finale du résultat.*

## 10.A Quelques interfaces et classes disponibles dans `java.util.concurrent`

Les descriptions et exemples qui suivent sont tirés du *Web* :

- `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`

  La plupart sont disponibles dans le *package* `java.util.concurrent`.

### 10.A.1 Interface `Lock`

```
public interface Lock {
  void lock()
  // Acquires the lock.

  void lockInterruptibly()
  // Acquires the lock unless the current thread
  // is interrupted.

  Condition newCondition()
  // Returns a new Condition instance
  // that is bound to this Lock instance.

  boolean tryLock()
  // Acquires the lock only if it is free
  // at the time of invocation.

  boolean tryLock(long time, TimeUnit unit)
  // Acquires the lock if it is free within
  // the given waiting time and the current
  // thread has not been interrupted.

  void unlock()
  // Releases the lock.
}
```

Pourquoi définir une telle interface ainsi que diverses classes qui mettent en oeuvre cette interface? Réponse extraite de `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Lock.html` :

*Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.*

*[...]*

*The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.*

*While the scoping mechanism for synchronized methods and statements makes it much easier to program with monitor locks, and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. [...]*

*With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. In most cases, the following idiom should be used:*

```
Lock l = ...;
l.lock();
try {
    // Section critique ou on accede
    // a la ressource protegee par
    // le verrou l.
    ...
} finally {
    l.unlock();
}
```

[...] *Lock implementations provide additional functionality over the use of synchronized methods and statements by providing a non-blocking attempt to acquire a lock (tryLock()), an attempt to acquire the lock that can be interrupted (lockInterruptibly(), and an attempt to acquire the lock that can timeout (tryLock(long, TimeUnit)).*

*A Lock class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering,*

*non-reentrant usage, or deadlock detection. If an implementation provides such specialized semantics then the implementation must document those semantics.*

*[...]*

### 10.A.2 Classe `ReentrantLock`

*A reentrant mutual exclusion* `Lock` *with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.*

*A* `ReentrantLock` *is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking* `lock` *will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods* `isHeldByCurrentThread()`*, and* `getHoldCount()`*.*

```
public class ReentrantLock implements Lock {
  ReentrantLock()
  ReentrantLock(boolean fair)

  int getHoldCount()
  protected Thread getOwner()
  protected Collection<Thread> getQueuedThreads()
  int getQueueLength()
  protected Collection<Thread> getWaitingThreads(Condition condition)
  int getWaitQueueLength(Condition condition)
  boolean hasQueuedThread(Thread thread)
  boolean hasQueuedThreads()
  boolean hasWaiters(Condition condition)
  boolean isFair()
  boolean isHeldByCurrentThread()
  boolean isLocked()
  void lock()
  void lockInterruptibly()
  Condition newCondition()
  String toString()
  boolean tryLock()
  boolean tryLock(long timeout, TimeUnit unit)
  void unlock()
}
```

Pourquoi la bibliothèque `java.util.concurrent` définit-elle une interface `Lock` ainsi que des classes qui mettent en oeuvre cette interface — `ReentrantLock` et `ReadWriteReentrantLock` — alors que des verrous sont déjà disponibles avec `synchronized` et `synchronized(this){...}`?

**Exercice 10.6:** Interface `Lock` et classes associées.

Que se passe-t-il si un *thread* exécute une méthode `synchronized` puis tente d'appeler une autre méthode elle aussi `synchronized` de la même classe?

**Exercice 10.7:** Appels multiples à `synchronized`.

### 10.A.3 Interface `Condition`

```
public interface Condition {
  void await()
  boolean await(long time, TimeUnit unit)
  long awaitNanos(long nanosTimeout)
  void awaitUninterruptibly()
  boolean awaitUntil(Date deadline)
  void signal()
  void signalAll()
}

public interface Lock {
  ...
  Condition newCondition()
  // Returns a new Condition instance
  // that is bound to this Lock instance.
  ...
}
```

*Donc*, lorsqu'on fait `cond.await()`, c'est le verrou *parent* qui est libéré, puis réacquis après réception du `signal()`.

### 10.A.4 Classe `Semaphore`

```
public class Semaphore {
  Semaphore(int permits)
  Semaphore(int permits, boolean fair)

  void acquire()
  void acquire(int permits)
```

```
  void acquireUninterruptibly()
  void acquireUninterruptibly(int permits)

  int availablePermits()
  int drainPermits()
  protected Collection<Thread> getQueuedThreads()
  int getQueueLength()
  boolean hasQueuedThreads()
  boolean isFair()
  protected void reducePermits(int reduction)

  void release()
  void release(int permits)

  boolean tryAcquire()
  boolean tryAcquire(int permits)
  boolean tryAcquire(int permits, long timeout, TimeUnit unit)
  boolean tryAcquire(long timeout, TimeUnit unit)
}
```

| Est-ce que «`acquire(2);`» a le même effet que «`acquire(); acquire()`»? |
|---|

**Exercice 10.8:** Différences entre appels à `acquire`?

### 10.A.5 Classe `CyclicBarrier`

```
public class CyclicBarrier {
  CyclicBarrier(int parties)
  // Creates a new CyclicBarrier that will trip
  // when the given number of parties (threads)
  // are waiting upon it, and does not perform
  // a predefined action upon each barrier.
  ...

  int await()
  // Waits until all parties have invoked await on this barrier.

  int await(long timeout, TimeUnit unit)
  // Waits until all parties have invoked await on this barrier.

  int getNumberWaiting()
  // Returns the number of parties currently waiting at the barrier.
```

```
  int getParties()
  // Returns the number of parties required to trip this barrier.

  ...

  void reset()
  // Resets the barrier to its initial state.
}
```

### 10.A.6 Classe `CountDownLatch`

*A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.*

*A* `CountDownLatch` *is initialized with a given* `count`*. The* `await` *methods block until the current count reaches zero due to invocations of the* `countDown()` *method, after which all waiting threads are released and any subsequent invocations of* `await` *return immediately. This is a one-shot phenomenon — the count cannot be reset. If you need a version that resets the count, consider using a* `CyclicBarrier`*.*

```
public class CountDownLatch {
  CountDownLatch(int count)
  // Constructs a CountDownLatch initialized
  // with the given count.

  void await()
  // Causes the current thread to wait until
  // the latch has counted down to zero,
  // unless the thread is interrupted.

  boolean await(long timeout, TimeUnit unit)
  // Causes the current thread to wait until the latch has counted down
  // to zero, unless the thread is interrupted, or the specified waiting
  // time elapses.

  void countDown()
  // Decrements the count of the latch, releasing all waiting
  // threads if the count reaches zero.

  long getCount()
  // Returns the current count.
  ...
}
```

### 10.A.7 Classe Exchanger<V>

*A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return.*

```
public class Exchanger<V> {
  Exchanger()
  // Creates a new Exchanger.

  V exchange(V x)
  // Waits for another thread to arrive at this
  // exchange point (unless the current thread
  // is interrupted), and then transfers the given
  // object to it, receiving its object in return.

  V exchange(V x, long timeout, TimeUnit unit)
}
```

### 10.A.8 Classes pour des objets atomiques

Diverses classes — `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference` — permettent de manipuler des objets (par opposition à des valeurs de types primitifs tels `int`, `boolean`) de façon atomique. Dans le cas des classes `AtomicInteger` et `AtomicReference`, les opérations sont les suivantes :

```
public class AtomicInteger {
  AtomicInteger()
  AtomicInteger(int initialValue)

  int addAndGet(int delta)
  // Atomically adds the given value to the current value.
  // Returns: the updated value

  boolean compareAndSet(int expect, int update)
  int decrementAndGet()
  int get()

  int getAndAdd(int delta)
  // Atomically adds the given value to the current value.
  // Returns: the previous value

  int getAndDecrement()
```

```
  int getAndIncrement()
  int getAndSet(int newValue)
  int incrementAndGet()
  int intValue()
  long longValue()
  void set(int newValue)
  String toString()
  ...
}
```

**Exemples :**

- Supposons un seul et unique *thread* :

```
AtomicInteger ai = new AtomicInteger( 17 );

assert ai.get() == 17;
assert !ai.compareAndSet( 19, 23 );
assert ai.get() == 17;

assert ai.compareAndSet( 17, 23 );
assert ai.get() == 23;
```

- Supposons deux *threads* :

```
AtomicInteger ai = new AtomicInteger( 17 );

Thread t0 = new Thread( () -> {
  int x = ai.get(); ai.compareAndSet( x, x+1 );
} );

Thread t1 = new Thread( () -> {
  int y = ai.get(); ai.compareAndSet( y, y+1 );
} );

t0.start(); t1.start();

try { t0.join(); t1.join(); } catch( Exception e ){ ... }

ai.get() == ? ?;
```

Quelle est la valeur — ou *les valeurs* — qui peut être retournée par `ai.get()`?

**Exercice 10.9:** Valeur possible pour un `AtomicInteger`

Supposons que l'on ait la classe `AtomicInteger` avec `compareAndSet()` **mais sans** `increment()` :

```
public final boolean compareAndSet( int expect,
                                    int update )
```

**Voici alors une mise en oeuvre de** `increment()` :

```
public int increment() {
  int valeurAvant,
      valeurApres;

  do {
    valeurAvant = valeur.get();
    valeurApres = valeurAvant + 1;
  } while( !valeur.compareAndSet( valeurAvant,
                                  valeurApres ) );

  return valeurApres;
}
```

Dans la méthode `increment`, quand l'appel à `compareAndSet` peut-il retourner `false`?

**Exercice 10.10:** Méthode `compareAndSet`.

```
public class AtomicReference<V> {
  AtomicReference()

  AtomicReference(V initialValue)

  boolean compareAndSet(V expect, V update)
  // Atomically sets the value to the given updated
  // value if the current value == the expected value.

  V get()

  V getAndSet(V newValue)
  // Atomically sets to the given value
  // and returns the old value.

  void set(V newValue)
```

```
  String toString()
}
```

Pour manipuler des `Doubles` de façon atomique, on doit définir soi même une classe appropriée :

```
class AtomicDouble {
  private AtomicReference<Double> value;

  public AtomicDouble( double initVal ) {
    value = new AtomicReference<Double>(
                      new Double(initVal) );
  }

  public double get() {
    return value.get().doubleValue();
  }

  public boolean compareAndSet( double expect,
                                double update ) {
   Double origVal, newVal;

   newVal = new Double(update);
   while( true ) {
    origVal = value.get();

    if (Double.compare(origVal.doubleValue(), expect) == 0) {
      if ( value.compareAndSet(origVal, newVal) ) {
        return true;
      }
    } else {
      return false;
    }
   }
  }
}
```

### 10.A.9 Interfaces `Callable<V>` et `Future<V>`

`Callable<V>`

*A task that returns a result and may throw an exception. Implementors define a single method with no arguments called* `call`.

*The* `Callable` `interface` *is similar to* `Runnable`*, in that both are designed for classes whose instances are potentially executed by another thread. A* `Runnable`*, however, does not return a result and cannot throw a checked exception.*

Un `Runnable` ne peut pas retourner de résultat `:(`

```
interface Runnable {
   void run();
}
```

Un `Callable` permet de retourner un résultat :)

```
public interface Callable<V> {
  V call()
  // Computes a result, or throws an exception
  // if unable to do so.
}
```

`Future<V>`

*A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method* `get` *when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the* `cancel` *method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled.*

```
public interface Future<V>  {
  boolean cancel(boolean mayInterruptIfRunning)
  // Attempts to cancel execution of this task.

  V get()
  // Waits if necessary for the computation
  // to complete, and then retrieves its result.

  V get(long timeout, TimeUnit unit)
  // Waits if necessary for at most the given time
  // for the computation to complete, and
  // then retrieves its result, if available.

   boolean isCancelled()
   // Returns true if this task was cancelled
   // before it completed normally.

   boolean isDone()
   // Returns true if this task completed.
}
```

**Exemple d'utilisation** :

```
public interface ArchiveSearcher {
  String search(String target);
}

class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...

  void showSearch(final String target)
                  throws InterruptedException {
    Future<String> future = executor.submit(
            new Callable<String>() {
               public String call() {
                 return searcher.search(target);
               }
            });
    displayOtherThings(); // do other things while searching

    try {
      displayText(future.get()); // use future
    } catch ( ExecutionException ee ) {
      cleanup(); return;
    }
  }
}
```

### 10.A.10 *Pool de threads*

L'utilisation d'un *pool de threads* permet de créer un certain nombre de *threads* qui vont rester «vivants» aussi longtemps que le *pool* existera, si nécessaire en restant «inactifs» en attente de travail à faire. Lorsqu'une tâche sera donnée à exécuter au *pool* (via un `Runnable` ou un `Callable`, décrit dans la section précédente), un des *threads* du *pool* l'exécutera, puis retournera en attente de nouvelles tâches.

On utilise un *pool de threads* lorsqu'on veut avoir un nombre maximum de *threads* actifs à un instant donné, tout en ayant une certaine flexibilité quant à leur affectation (i.e., les *threads* vont effectuer des tâches diverses tout au long de leur durée de vie).

Pour des tâches de fine granularité, l'utilisation d'un *pool de threads* évite de créer de façon répétitive de nouveaux *threads*, qui ne s'exécuteraient que pour une courte période, puis se termineraient — donc avec un *overhead* élevé de création puis destruction du *thread* par rapport au travail effectué par le *thread*.

### Interface `Executor`

L'interface de base pour les *pool de threads* est la suivante :

```
public interface Executor {
  public void execute(Runnable task);
}
```

### Interface `ExecutorService`

Extraits de l'interface :

```
public interface ExecutorService {
  boolean awaitTermination(long timeout, TimeUnit unit)
  // Blocks until all tasks have completed execution after
  // a shutdown request, or the timeout occurs, or
  // the current thread is interrupted, whichever happens first.

  <T> List<Future<T>>
      invokeAll(Collection<Callable<T>> tasks)
  // Executes the given tasks, returning a list of Futures
  // holding their status and results when all complete.

  <T> T
      invokeAny(Collection<Callable<T>> tasks)
  // Executes the given tasks, returning the result of one that has
  // completed successfully (i.e., without throwing an exception), if any do.

  boolean isShutdown()
   // Returns true if this executor has been shut down.

  boolean isTerminated()
  // Returns true if all tasks have completed following
  // shut down.

 void shutdown()
 // Initiates an orderly shutdown in which previously
 // submitted tasks are executed,
 // but no new tasks will be accepted.

 List<Runnable> shutdownNow()
 // Attempts to stop all actively executing tasks,
 // halts the processing of waiting tasks, and
 // returns a list of the tasks that were awaiting execution.
```

```
<T> Future<T> submit(Callable<T> task)}
// Submits a value-returning task for execution
// and returns a Future representing the pending
// results of the task.

Future<?> submit(Runnable task)
// Submits a Runnable task for execution and returns a
// Future representing that task.

<T> Future<T> submit(Runnable task, T result)
// Submits a Runnable task for execution and returns
// a Future representing that task that will upon
// completion return the given result
```

### Classe de base pour un `ExecutorService` : `ThreadPoolExecutor`

Extraits de la classe — comporte approx. 40 méthodes :

```
Class ThreadPoolExecutor implements Executor, ExecutorService {

  ThreadPoolExecutor(int corePoolSize,
                     int maximumPoolSize,
                     long keepAliveTime,
                     TimeUnit unit,
                     BlockingQueue<Runnable> workQueue)
  // Creates a new ThreadPoolExecutor with the given
  // initial parameters and default thread factory
  // and handler.
  ...

  void execute(Runnable command)
  // Executes the given task sometime
  // in the future.
  ...

  int getActiveCount()
  // Returns the approximate number of threads that are
  // actively executing tasks.

  long getCompletedTaskCount()
  // Returns the approximate total number of tasks that have
  // completed execution.
  ...
```

```
  void purge()
  // Tries to remove from the work queue all Future tasks
  // that have been cancelled.

  boolean remove(Runnable task)
  // Removes this task from the executor's internal queue
  // if it is present, thus causing it not to be run
  // if it has not already started.

  ...

  void shutdown()
  // Initiates an orderly shutdown in which previously submitted
  // tasks are executed, but no new tasks will be accepted.

  List<Runnable> shutdownNow()
  // Attempts to stop all actively executing tasks, halts the
  // processing of waiting tasks, and returns a list of the
  // tasks that were awaiting execution.

}
```

**Remarque :** `ThreadPoolExecutor` est une sous-classe de `AbstractExecutorService`, laquelle classe définit, entre autres, la méthode suivante :

```
<T> Future<T> submit(Callable<T> task)
```

Effets des paramètres (du constructeur) sur le comportement des *threads* et du *pool* :

- *A* `ThreadPoolExecutor` *will automatically adjust the pool size according to the bounds set by* `corePoolSize` *and* `maximumPoolSize`.

  *When a new task is submitted in method* `execute`*, and fewer than* `corePoolSize` *threads are running, a new thread is created to handle the request, even if other worker threads are idle.*

  *If there are more than* `corePoolSize` *but less than* `maximumPoolSize` *threads running, a new thread will be created only if the queue is full.*

  *By setting* `corePoolSize` *and* `maximumPoolSize` *the same, you create a fixed-size thread pool. By setting* `maximumPoolSize` *to an essentially unbounded value such as* `Integer.MAX_VALUE`*, you allow the pool to accommodate an arbitrary number of concurrent tasks.*

- *If the pool currently has more than* `corePoolSize` *threads, excess threads will be terminated if they have been idle for more than the* `keepAliveTime`*. This provides*

*a means of reducing resource consumption when the pool is not being actively used. If the pool becomes more active later, new threads will be constructed.*

- *Any* `BlockingQueue` *may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:*
  - *If fewer than* `corePoolSize` *threads are running, the* `Executor` *always prefers adding a new thread rather than queuing.*
  - *If* `corePoolSize` *or more threads are running, the* `Executor` *always prefers queuing a request rather than adding a new thread.*
  - *If a request cannot be queued, a new thread is created unless this would exceed* `maximumPoolSize`*, in which case, the task will be rejected.*

### Méthodes de fabrication pour créer un `ExecutorService`

Les **trois** principales (parmi une vingtaine d'autres) :

```
class Executors {
  // Methods that create and return an ExecutorService
  // set up with commonly useful configuration settings.
  ...
  static ExecutorService
         newCachedThreadPool()
  // Creates a thread pool that creates new threads
  // as needed, but will reuse previously constructed
  // threads when they are available.

  static ExecutorService
         newFixedThreadPool(int nThreads)
  // Creates a thread pool that reuses a fixed set
  // of threads operating off a shared unbounded queue.

  static ExecutorService
         newSingleThreadExecutor()
  // Creates an Executor that uses a single worker
  // thread operating off an unbounded queue.
  // Tasks are guaranteed to execute sequentially,
  // and no more than one task will be active
  // at any given time.
}
```

Quelques informations supplémentaires :

- `newCachedThreadPool` : *Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools*

*will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created.*

- `newFixedThreadPool` : *Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.*

### 10.A.11 Classe `ThreadLocal<T>`

*This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).*

```
public class ThreadLocal<T> {
  ThreadLocal()

  Object get()
  // Returns the value in the current thread's copy
  // of this thread-local variable.

  protected Object initialValue()
  // Returns the current thread's initial value
  // for this thread-local variable.

  void set(Object value)
  // Sets the current thread's copy of this thread-local
  // variable to the specified value.
}
```

Exemple d'utilisation : *For example, in the class below, the private* `static ThreadLocal` *instance (*`serialNum`*) maintains a "serial number" for each thread that invokes the class's static* `SerialNum.get()` *method, which returns the current thread's serial number. (A thread's serial number is assigned the first time it invokes* `SerialNum.get()`*, and remains unchanged on subsequent calls.)*

*Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread*

*goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).*

```
public class SerialNum {
    // The next serial number to be assigned
    private static int nextSerialNum = 0;

    private static ThreadLocal serialNum
                    = new ThreadLocal() {
        protected synchronized Object
                                initialValue() {
            return new Integer(nextSerialNum++);
        }
    };

    public static int get() {
        return ((Integer) (serialNum.get())).intValue();
    }
}
```

### 10.A.12 Les collections concurrentes

Les principales collections concurrentes disponibles en Java 8 — package `java.util.concurrent` :

- `ArrayBlockingQueue<E>`
- `ConcurrentHashMap<K,V>`
- `ConcurrentLinkedDeque<E>`
- `ConcurrentLinkedQueue<E>`
- `ConcurrentSkipListMap<K,V>`
- `ConcurrentSkipListSet<K,V>`

`ConcurrentHashMap<K,V>` :

> A *hash table* supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as `Hashtable` and includes versions of methods corresponding to each method of `Hashtable`.However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access.

Retrieval operations (including get) generally do not block, so *may overlap with* update operations (including put and remove).

Source : `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`

- `ConcurrentHashMap` *:*
  - *It is thread safe without synchronizing the whole map.*
  - *Reads can happen very fast while write is done with a lock.*
  - *There is no locking at the object level.*
  - *The locking is at a much finer granularity at a hashmap bucket level.*
- `SynchronizedHashMap` *(ancienne version!)*
  - *Synchronization at* `Object` *level.*
  - *Every read/write operation needs to acquire lock.*
  - *Locking the entire collection is a performance overhead.*
  - *This essentially gives access to only one thread to the entire map & blocks all the other threads.*

*Source :* `http://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashm`

## 10.B Comparaison des performances entre `synchronized`, `ReentrantLock` et `AtomicInteger`

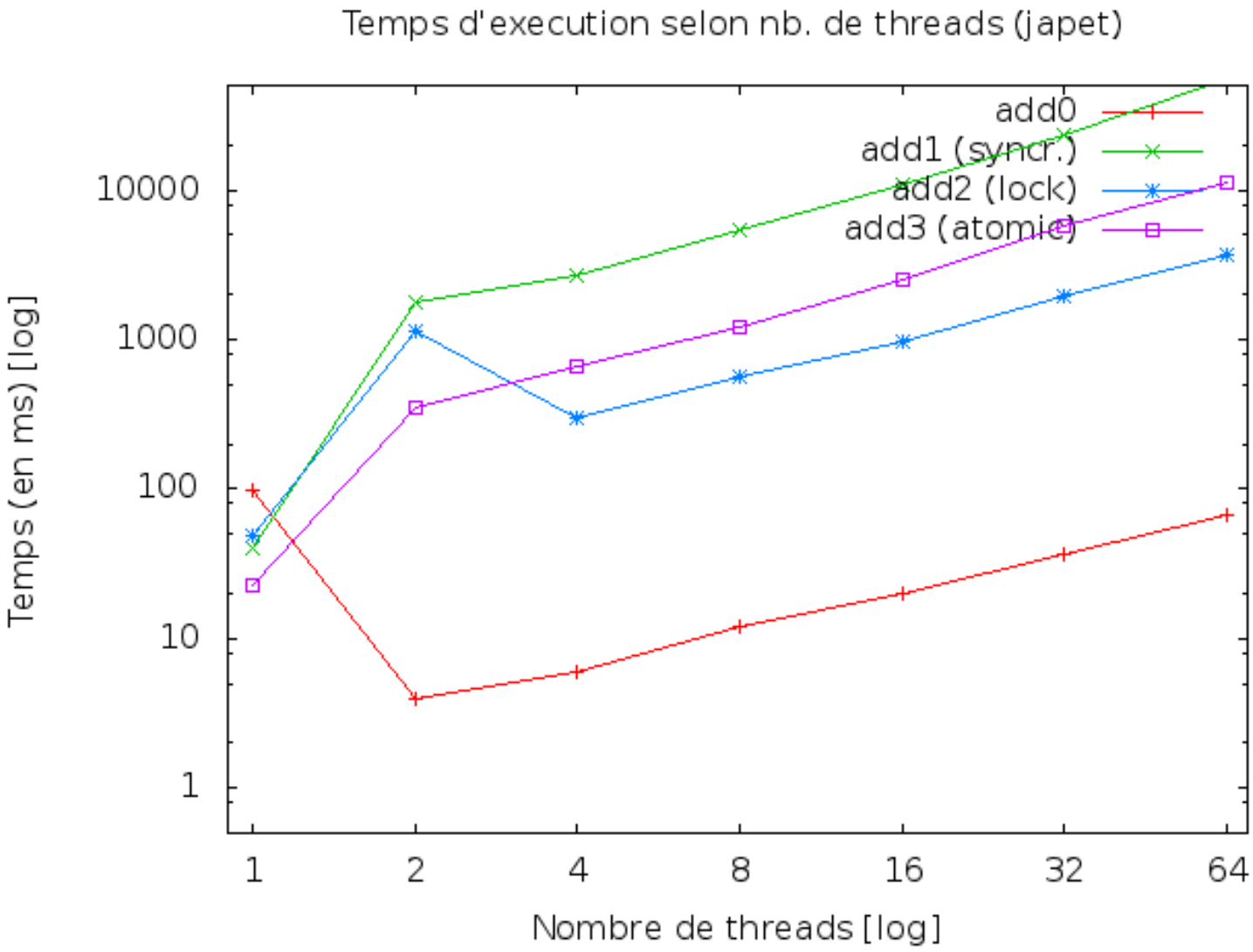
Dans cette section, nous allons comparer les performances relatives entre `synchronized`, `ReentrantLock` et `AtomicInteger`.

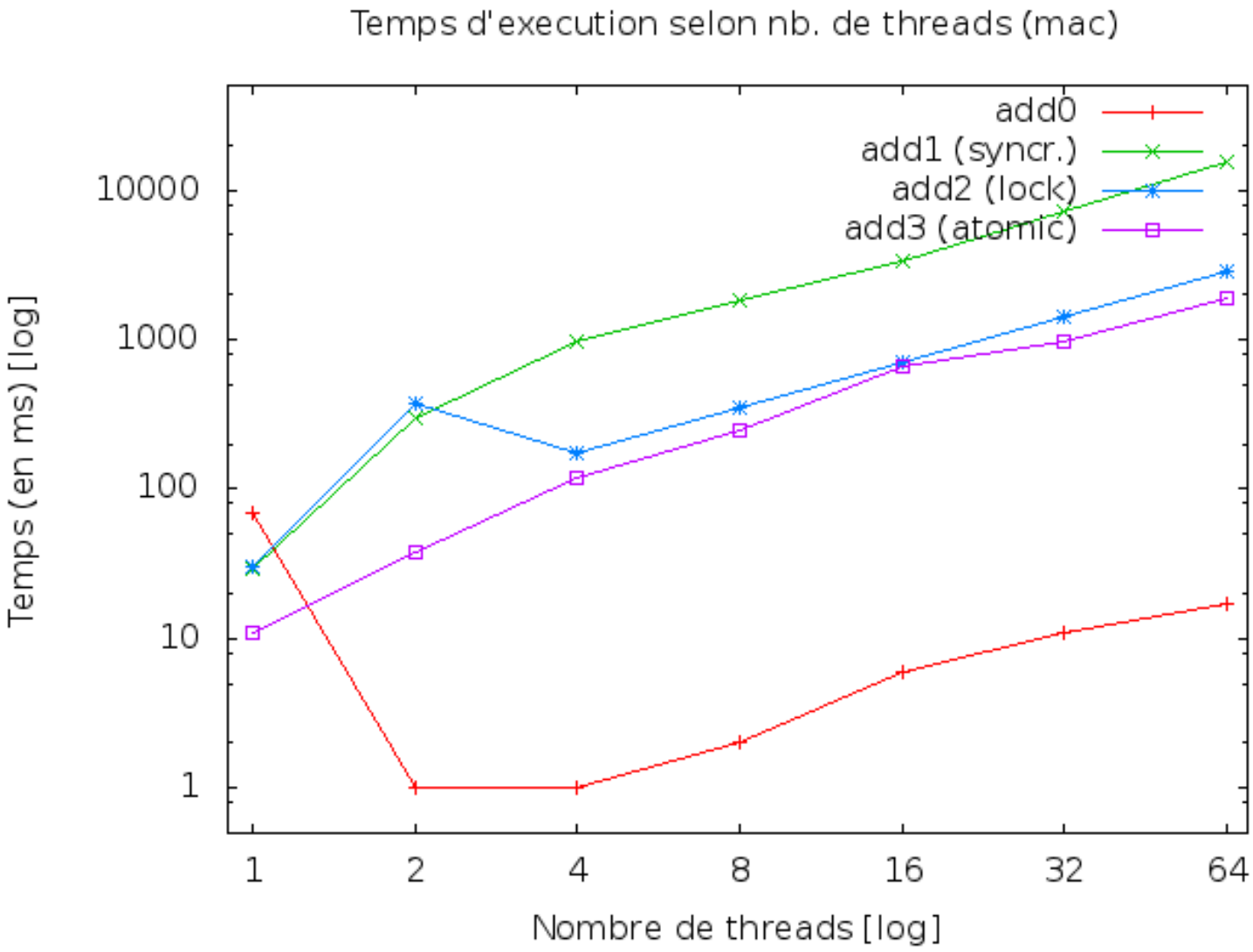
**Programme Java 10.18** Une classe `VarInteger`.

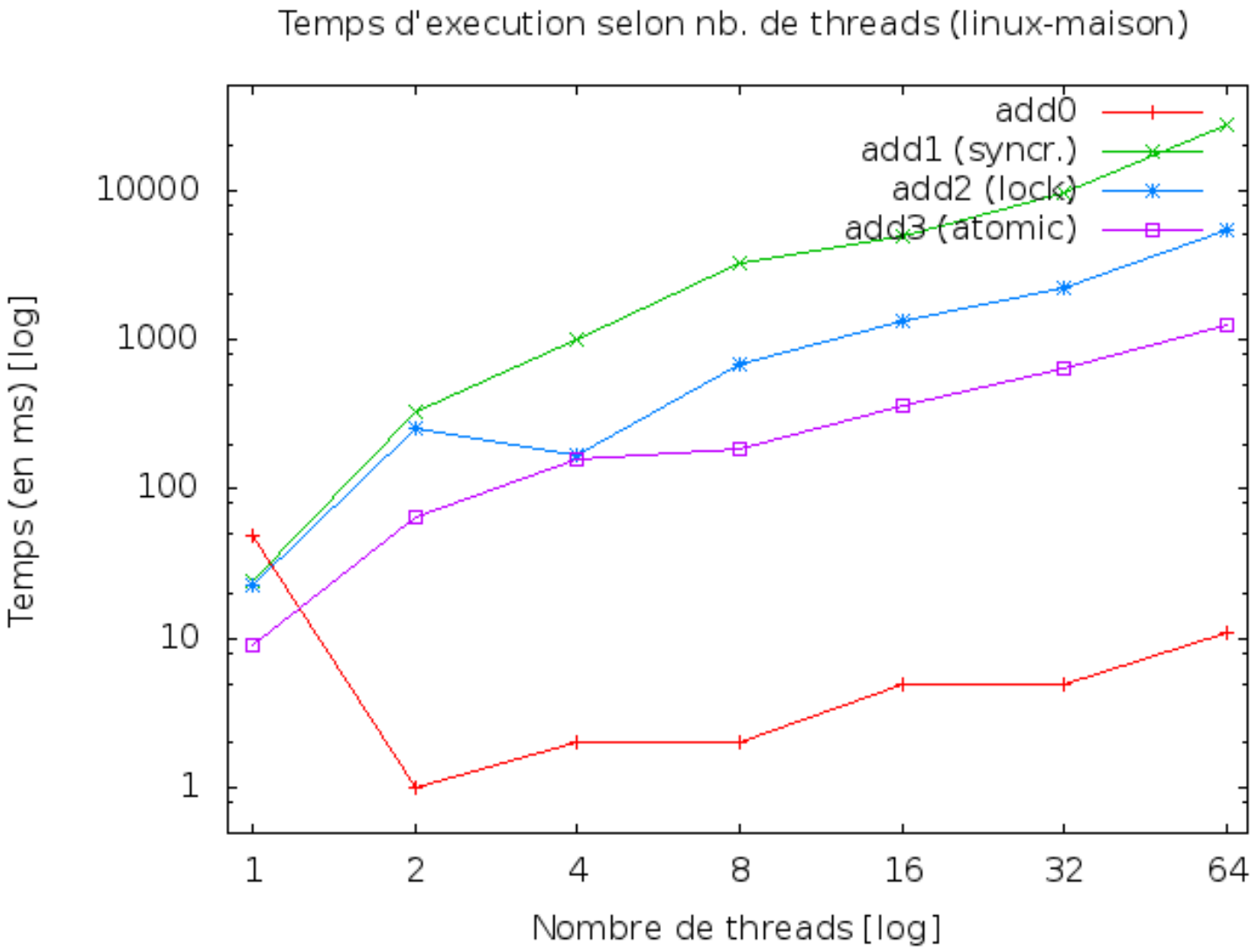
```
// Modelise une variable contenant un entier.
class VarInteger {
  private int val = 0;
  private final ReentrantLock lock; // Pour add2.
  private AtomicInteger atomicVal; // Pour add3.

  VarInteger() {
    lock = new ReentrantLock();
    atomicVal = new AtomicInteger( 0 );
  }

  // Methode sans exclusion mutelle.
  void add0( int x ) { val += x; }

  // Methode avec verrou "primitif".
  synchronized void add1( int x ) { val += x; }

  // Methode avec ReentrantLock.
  void add2( int x ) { lock.lock(); val += x; lock.unlock(); }

  // Methode avec AtomicInteger.
  void add3( int x ) { val = atomicVal.addAndGet( x ); }

  // Methode pour dispatcher.
  void add( int numMethode, int x ) {
    switch ( numMethode ) {
    case 0: add0(x); break;
    case 1: add1(x); break;
    case 2: add2(x); break;
    case 3: add3(x); break;
    }
  }

  public int value() { return val; }
}
```

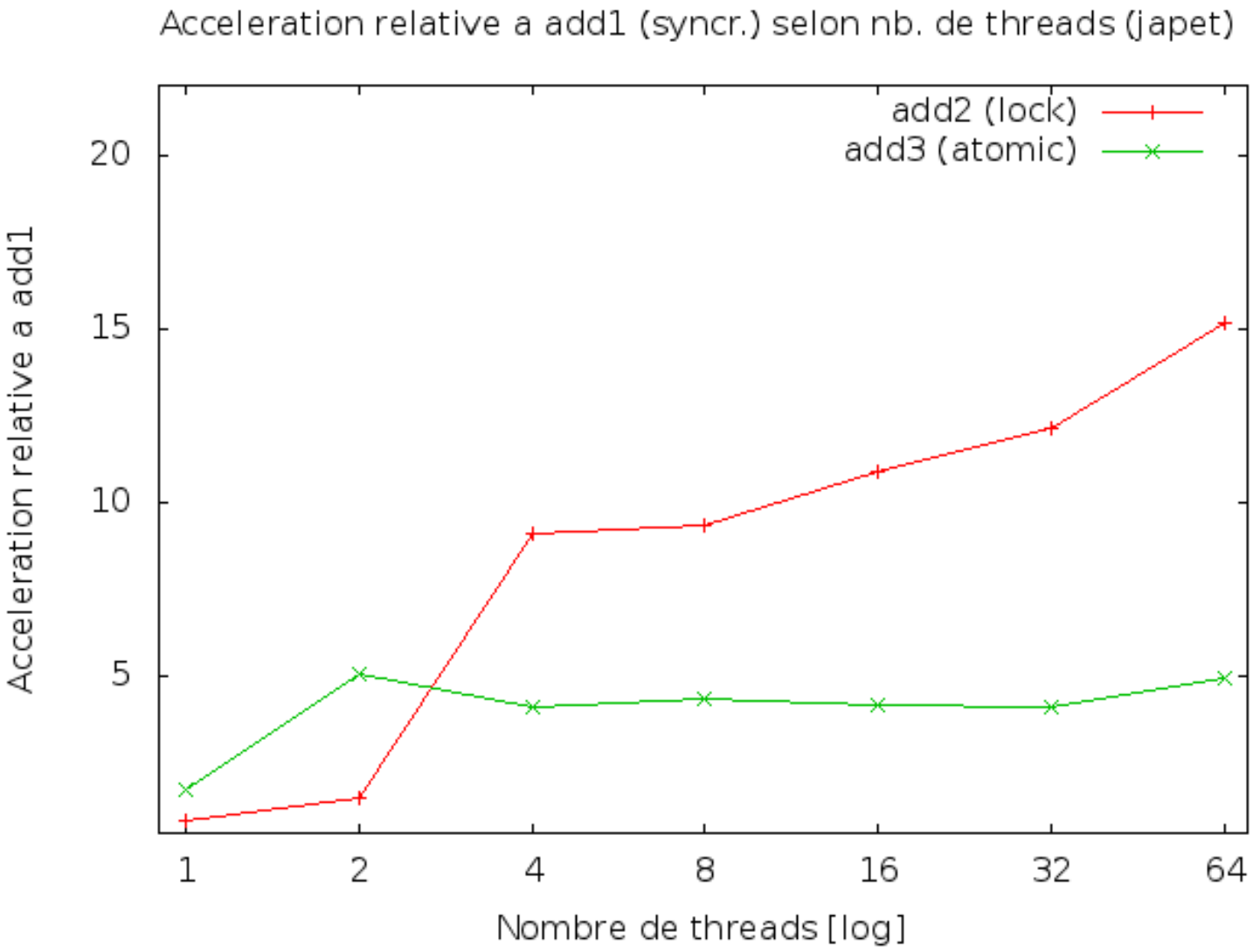
**Programme Java 10.19** Un programme de *benchmark* utilisant `VarInteger`.

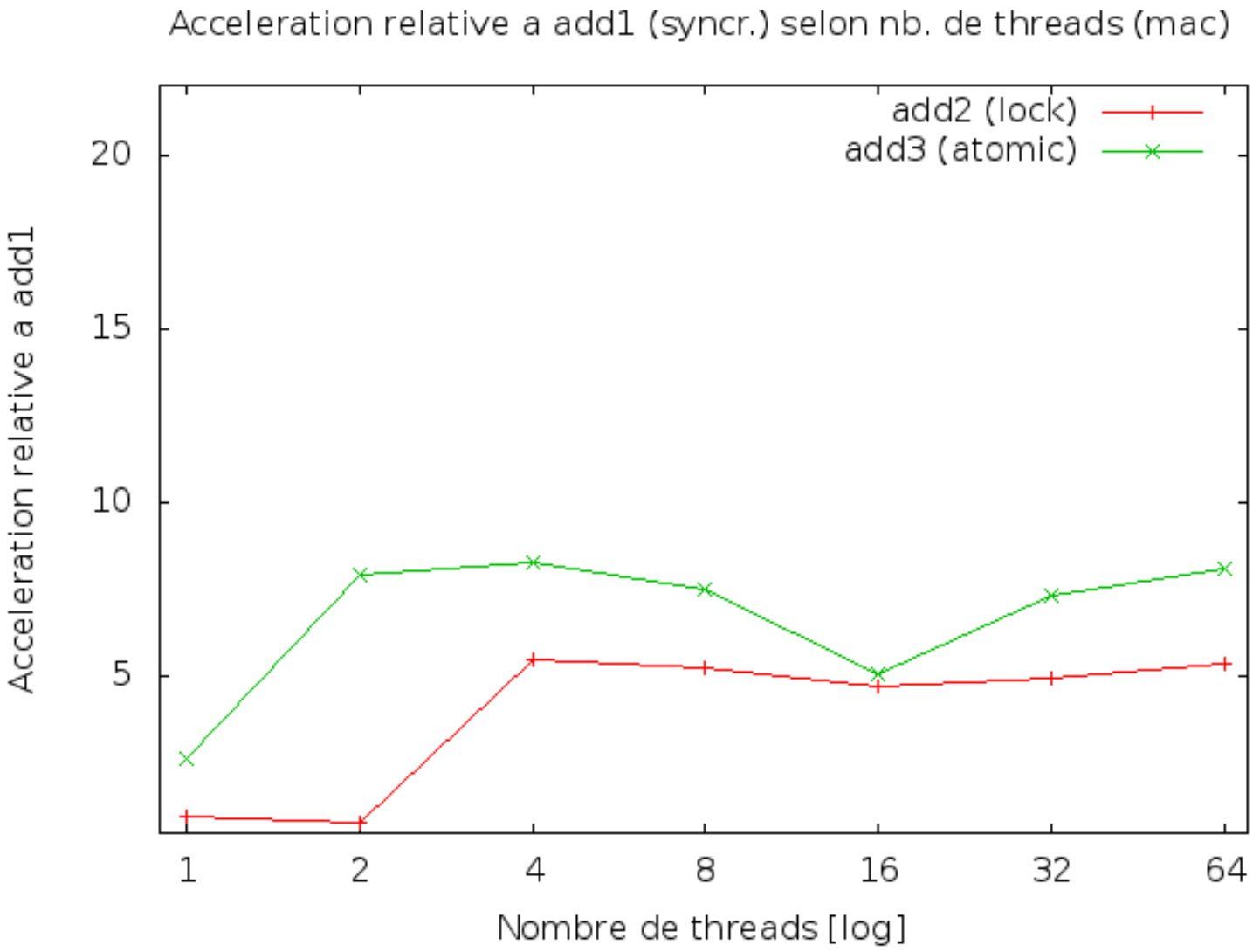
```
// On alloue un tableau.
int a[] = new int[nbElements];
for( int i = 0; i < nbElements; i++ ) { a[i] = 1; }

// On va executer pour divers nombres de threads.
int[] lesNbsThreads = {1, 2, 4, 8, 16, 32, 64};

System.out.format( "#%s %7s %7s %7s %7s\n",
       " nb.thr.", "add0", "add1", "add2", "add3" );
for( int nbThreads: lesNbsThreads ) {
  System.out.format( "%8d ", nbThreads );
  for( int method = 0; method <= 3; method++ ) {
    int numMethode = method;

    // On lance la minuterie.
    long tempsDebut = System.currentTimeMillis();

    // On lance les threads, selon la methode demandee.
    VarInteger total = new VarInteger();
    Thread[] threads = new Thread[nbThreads];
    for( int i = 0; i < nbThreads; i++ ) {
      threads[i] = new Thread( () -> {
          for( int x:  a ) { total.add( numMethode, x ); };
        } );
      threads[i].start();
    }
    for( Thread t: threads ) {
      try { t.join(); } catch( InterruptedException e ) {}
    }

    // On arrete la minuterie.
    long tempsFin = System.currentTimeMillis();
    System.out.format( "%7d ", (tempsFin - tempsDebut) );

  }
  System.out.format( "\n" );
}
```

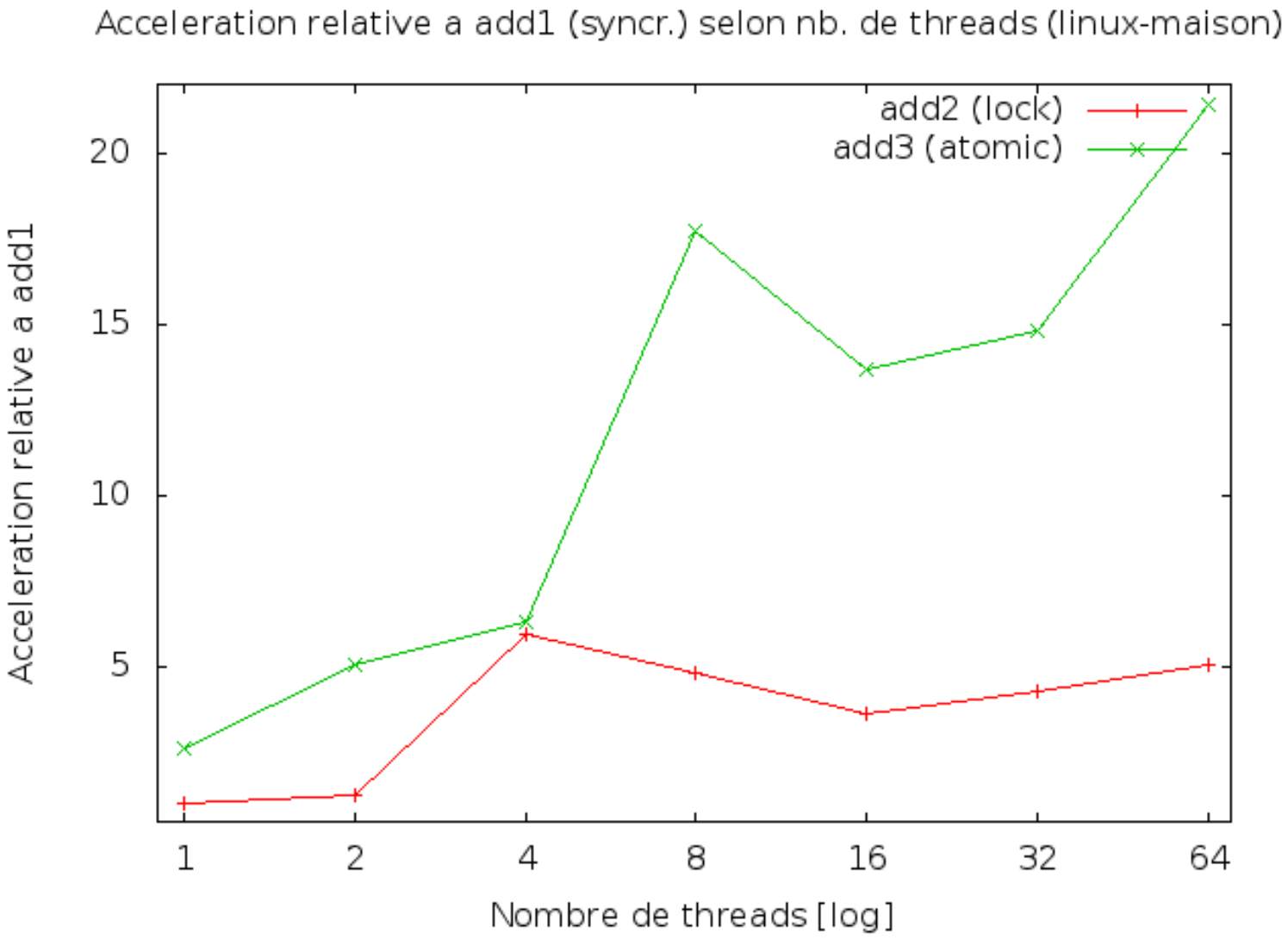

Temps d'execution selon nb. de threads (japet)
add0
add1 (syncr.)
add2 (lock)
add3 (atomic)
10000
1000
100
10
1
Temps (en ms) [log]
1
2
4
8
16
32
64
Nombre de threads [log]


Temps d'execution selon nb. de threads (mac)
add0
add1 (syncr.)
add2 (lock)
add3 (atomic)
10000
1000
100
10
1
Temps (en ms) [log]
1
2
4
8
16
32
64
Nombre de threads [log]


Temps d'execution selon nb. de threads (linux-maison)
add0
add1 (syncr.)
add2 (lock)
add3 (atomic)
10000
1000
100
10
1
Temps (en ms) [log]
1
2
4
8
16
32
64
Nombre de threads [log]


Acceleration relative a add1 (syncr.) selon nb. de threads (japet)
add2 (lock)
add3 (atomic)
Acceleration relative a add1
20
15
10
5
1
2
4
8
16
32
64
Nombre de threads [log]


Acceleration relative a add1 (syncr.) selon nb. de threads (mac)
add2 (lock)
add3 (atomic)
Acceleration relative a add1
20
15
10
5
1
2
4
8
16
32
64
Nombre de threads [log]


Acceleration relative a add1 (syncr.) selon nb. de threads (linux-maison)
add2 (lock)
add3 (atomic)
Acceleration relative a add1
20
15
10
5
1
2
4
8
16
32
64
Nombre de threads [log]

## 10.C Exemple avec *streams* (paquetage `java.util.stream`)

Le Programme Java 10.20 présente un exemple d'utilisation des *streams* de Java 8.0. Il s'agit d'une fonction pour trier les mots d'un fichier, de façon unique, donc semblable aux fonctions présentées dans les Programmes Ruby **??** et **??** — semblable mais pas identique, notamment puisque les mots triés sont simplement émis sur `stdout` plutôt que dans un fichier texte spécifique.

- L'appel à la méthode `parallel()` aurait pu être omis et le même résultat aurait été produit, mais sans exécution parallèle.
- Le tri avec `sorted` se fait, par défaut, avec la méthode `String::compareTo`.
- Le *stream* étant parallèle, son contenu doit être examiné avec `forEachOrdered` pour que l'impression se fasse en respectant l'ordre des éléments triés.

Pour plus de détails sur les *streams* Java, voir la documentation en ligne d'Oracle : `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

Effet de `flat_map` (version Ruby) :

```
>> [[10, 20, 30], [], [88, 99]].
   flat_map { |x| x }
=> [10, 20, 30, 88, 99]


>> ["abc def", "xy", "1 2 3"].
     flat_map { |x| x.split(" ") }
=> ["abc", "def", "xy", "1", "2", "3"]
```

**Programme Java 10.20** Fonction pour trier les mots d'un fichier, en s'assurant que chaque mot apparaît au plus une fois — version avec *streams*de Java 8.0.

```
private static Stream<String>
               lignes( String nomFichier ) {
  try {
    return
      new BufferedReader(
          new FileReader(nomFichier)
      ).lines();
  } catch (Exception e) {
    ...
  }
}

private static Stream<String>
               genererMots( String ligne ) {
  return Stream.of( ligne.split(" ") );
}

private static boolean
               motValide( String mot ) {
  Pattern pattern = Pattern.compile("\\w+");
  Matcher matcher = pattern.matcher(mot);
  return matcher.find();
}
```

```
public static void
       trierMotsUniques( String fichEntree ) {
  lignes(fichEntree)
    .parallel()
    .flatMap( l -> genererMots( l ) )
    .filter( m -> motValide( m ) )
    .sorted()
    .distinct()
    .forEachOrdered( System.out::println );
}
```

## 10.D Allocation dynamique de tableaux génériques

L'allocation directe de tableaux génériques n'est pas «naturelle» en Java.

Soit la classe générique suivante :

```
class Pair<T,U> {
  T first;
  U second;

  Pair( T f, U s ) {
    first = f;
    second = s;
  }

  T first()  { return first; }
  U second() { return second; }
}
```

On peut utiliser le «*diamond*» — <> — pour éviter de spécifier de façon explicite certains types, qui peuvent être *inférés* par le compilateur Java (inférence de types) :

```
// Forme explicite.
Pair<String,String> p
  = new Pair<String,String>("10", "abc");

// Forme implicite (inference de type).
Pair<String,String> p
  = new Pair<>("10", "abc");
```

Par contre, on ne peut pas utiliser le *diamond* pour allouer dynamiquement un tableau générique.

Les fragments de code suivants génèrent des *erreurs* de compilation :

```
Pair<String,String>[] a = new Pair<>[N];
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Generiques.java:31: error: cannot create array with '<>'
    Pair<String,String>[] a = new Pair<>[N];
```

---

```
Pair<String,String>[] a = (Pair<>[]) new Pair<String,String>[N];
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
Generiques.java:32: error: illegal start of type
    Pair<String,String>[] a = (Pair<>[]) new Pair<String,String>[N];
```

---

```
Pair<String,String>[] a = new Pair<String,String>[N];
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
Generiques.java:42: error: generic array creation
    Pair<String,String>[] a = new Pair<String,String>[N];
```

---

Les fragments de code suivants génèrent des *avertissements* de compilation :

```
Pair<String,String>[] a = (Pair<String,String>[]) new Pair[N];
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Generiques.java:55: warning: [unchecked] unchecked cast
    Pair<String,String>[] a = (Pair<String,String>[]) new Pair[N];
                                                           ^
  required: Pair<String,String>[]
  found:    Pair[]
```

---

```
Pair<String,String>[] a = new Pair[N];
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Generiques.java:47: warning: [unchecked] unchecked conversion
    Pair<String,String>[] a = new Pair[N];
                                  ^
  required: Pair<String,String>[]
  found:    Pair[]
```

---

On peut supprimer ces avertissements à l'aide d'une *annotation* appropriée devant la méthode qui contient le fragment de code en question :

```
@SuppressWarnings("unchecked")
void foo( ... ) {
  ...
  Pair<String,String>[] a = new Pair[N];
  ...
}
```

## 10.E Exercices additionnels

### 10.E.1 Les différentes formes de *pools de threads*

Soit le segment de code suivant :

```
int n = 1000;
double[] a = new double[n];
for( int i = 0; i < n; i++ ) {
  a[i] = 1.0;
}

ExecutorService pool = ?          ?;

double r
  = sommation_pr(a, 0, a.length-1, 100, pool);
System.out.println( r );
```

---

Qu'est-ce qui sera imprimé par le segment de code ci-haut selon les différentes valeurs possibles suivantes pour `pool` :

1. `Executors.newCachedThreadPool()`
2. `new ForkJoinPool(4)`
3. `Executors.newFixedThreadPool(4)`

**Exercice 10.11:** Méthodes `foo_tranche` et `foo_pr`.

**Programme Java 10.21** Méthodes `foo_tranche` et `foo_pr`.

```
public static double foo_tranche( double[] a,
                                  int inf,
                                  int sup ) {
  double somme = 0.0;
  for( int i = inf; i <= sup; i++ ) {
    somme += a[i];
  }
  return somme;
}

public static double foo_pr( double[] a,
                             int inf, int sup,
                             int seuil,
                             ExecutorService pool ) {
  if ( sup - inf + 1 <= seuil ) {
    return foo_tranche(a, inf, sup);
  }

  int mid = (sup + inf) / 2;
  Future<Double> gauche = pool.submit( () ->
     foo_pr(a, inf, mid, seuil, pool)
  );
  double droite = foo_pr(a, mid+1, sup, seuil, pool);

  try {
    return gauche.get() + droite;
  } catch( Exception e ) {
    assert false : "*** Exception : " + e;
    return 0.0D;
  }
}
```

Soit le segment de code suivant :

```
int n = 100000;
double[] a = new double[n];
for( int i = 0; i < n; i++ ) {
  a[i] = 1.0;
}

ExecutorService pool = ?          ?;

double r
  = sommation_pr(a, 0, a.length-1, 2, pool);
System.out.println( r );
```

---

Qu'est-ce qui sera imprimé par le segment de code ci-haut selon les différentes valeurs possibles suivantes pour `pool` :

1. `Executors.newCachedThreadPool()`
2. `new ForkJoinPool(4)`

**Exercice 10.12:** Méthodes `foo_tranche` et `foo_pr` (bis).

**Note :** Dans l'exemple qui précède, on aurait pu utiliser `gauche.get()`, mais il aurait alors fallu utiliser un `try/catch`.
**Note :** Il existe aussi une classe `RecursiveAction`, pour les tâches sans résultat — donc résultat de type `void`.

**Programme Java 10.22** Sommation des éléments d'un tableau avec des `RecursiveTask` et un `ForkJoinPool`.

```
class SommationRT extends RecursiveTask<Double> {
  private double[] a;
  private int inf, sup, seuil;

  SommationRT( double[] a, int inf, int sup, int seuil ) {
    this.a = a; this.inf = inf;
    this.sup = sup; this.seuil = seuil;
  }

  @Override
  public Double compute() {
    if ( sup - inf + 1 <= seuil ) {
      return sommation_tranche(a, inf, sup);
    }

    // On decompose en deux sous-problemes.
    int mid = (sup + inf) / 2;
    SommationRT gaucheRT = new SommationRT(a, inf, mid, seuil);
    SommationRT droiteRT = new SommationRT(a, mid+1, sup, seuil);

    // Tache pour sous-probleme gauche mais on garde droite.
    gaucheRT.fork();
    double droite = droiteRT.compute();

    return gaucheRT.join() + droite;
  }
  ...
```

```
// Utilisation.
ForkJoinPool pool = new ForkJoinPool(nbThreads);
SommationRT rt = new SommationRT(a, 0, a.length-1, seuil);
double r = pool.invoke(rt);
```

```
public abstract class RecursiveTask<V>
                      extends ForkJoinTask<V>
                      implements Future<V> {

  protected abstract V compute()
  // The main computation performed by this task.
}
```

---

```
public abstract class ForkJoinTask<V>
                      implements Future<V> {
  ForkJoinTask<V> fork()
  // Arranges to asynchronously execute this task
  // in the pool the current task is running in,
  // if applicable, or using the ForkJoinPool.commonPool()
  // if not inForkJoinPool().

  V get()
  // Waits if necessary for the computation to complete,
  // and then retrieves its result.


  V join()
  // Returns the result of the computation when it is done.
  // This method differs from get() in that abnormal completion
  // results in RuntimeException or Error, not ExecutionException,
  // and that interrupts of the calling thread do not cause
  // the method to abruptly return by throwing InterruptedException.
}
```

Figure 10.7: Quelques méthodes des classe `RecursiveTask` et `ForkJoinTask`.

**Remarques concernant `ForkJoinPool`**

Quelques remarques additionnelles concernant `ForkJoinPool` [UFM15]) :

- Lorsqu'un *thread* éxécute la méthode `join()` d'un autre *thread*, le premier *thread* bloque si le $2^e$ *thread* n'a pas terminé. Il faut donc faire tous les appels requis à des `fork()` **avant** de faire un `join()`.
- La méthode `invoke()` **ne doit pas** être appelée par une `ForkJoinTask` (ou une `RecursiveTask`). C'est plutôt la méthode `fork()` (ou `compute()`) qui doit être appelée. La méthode `invoke()` ne doit être appelée que pour le premier appel, effectué à partir d'un segment de code séquentiel.
- Comme on le ferait en Ruby lorsqu'on utilise des `future`s, si on doit traiter $n$ sous-problèmes, il est inutile de créer $n$ `ForkJoinTask` : il suffit d'en creér $n - 1$, l'autre tâche pouvant être traitée par le *thread* initial.

### 10.E.2 Un moniteur pour des `IStructures`

Une I-structure est une forme de *tableau* composé d'un certain nombre de **cellules**, où chaque cellule assure une *synchronisation* entre producteur (unique) et consommateurs (multiples) de la cellule par l'intermédiaire d'opérations `get` et `put` : voir l'interface `IStructure` dans le Programme Java 10.23.

Une classe concrète `IStructureMonitor` mettant en oeuvre cette interface est présentée dans le Programme Java 10.24. La taille effective de la `IStructure` — le nombre de cellules — est spécifiée au moment de l'allocation de la `IStructure`, dans le constructeur. Les cellules d'une `IStructure` possèdent le comportement suivant :

- Initialement, toutes les cellules sont **vides**.
- Une cellule d'une `IStructure` peut être lue avec `get`. Toutefois, lorsque la cellule est vide, le *thread* appelant *est mis en attente* jusqu'à ce que la cellule ait été remplie par un autre *thread*.
- Une cellule d'une `IStructure` ne peut être écrite, avec `put`, **qu'une seule et unique fois**. C'est une *erreur* (`AlreadyFullException`) d'écrire plus d'une fois dans une cellule donnée d'une `IStructure`. Si des lecteurs sont attentes au moment de l'écriture, ils sont réactivés de façon à obtenir la valeur qui vient d'être écrite.

Le Programme Java 10.25 présente un petit cas de test, avec le processeur lecteur (*thread*`getter`) mis en attente et réactivé lors de l'écriture par un autre *thread*.

| Complétez la mise en oeuvre des méthodes `put` et `get` de la classe `IStructureMonitor`, partiellement définie dans le Programme Java 10.23. |
|---|

**Exercice 10.13:** Méthodes `put` et `get` des `IStructureMonitor`.

**Programme Java 10.23** Interface pour une `IStructure`.

```
interface IStructure<T> {
  /**
   * Ecrit a la ieme position de la I-Structure.
   *
   * @param i Index
   * @param v Valeur a ecrire
   *
   * @requires 0 <= i && i < taille de la I-Structure
   */
  void put( int i, T v )
       throws AlreadyFullException;

  /**
   * Obtient le ieme element de la I-Structure.
   *
   * @requires 0 <= i && i < taille de la I-Structure
   *
   * @param i Index
   * @return L'element a la position indiquee
   */
  T get( int i );
}

class AlreadyFullException
      extends RuntimeException{}
```

**Programme Java 10.24** Une classe concrète `IStructureMonitor` qui met en oeuvre l'interface `IStructure`.

```
import java.util.concurrent.locks.*;

public class IStructureMonitor<T>
             implements IStructure<T> {
  private T[] elems;
  private ReentrantLock[] verrous;
  private Condition[] pleins;

  @SuppressWarnings("unchecked")
  public IStructureMonitor( int n ) {
    elems = (T[]) new Object[n];
    verrous = new ReentrantLock[n];
    pleins = new Condition[n];

    for( int i = 0; i < n; i ++ ) {
      elems[i] = null;
      verrous[i] = new ReentrantLock();
      pleins[i] = verrous[i].newCondition();
    }
  }
```

**Programme Java 10.25** Un cas de test pour `IStructureMonitor`.

```
@Test public void exemple_exercice() {
  IStructure<Integer> is
    = new IStructureMonitor<Integer>(1);

  Thread getter = new Thread( () -> {
      assertEquals( (Integer) 10, is.get(0) );
    });
  getter.start();

  new Thread( () -> {
      try { Thread.sleep(500); } catch(Exception e) {};
      is.put( 0, 10 );
  }).start();

  try { getter.join(); } catch( Exception e ){}
}
```

# Références

[Lea00] D. Lea. *Concurrent Programming in Java—Design Principles and Patterns (Second Edition)*. Addison-Wesley, 2000.

[OW04] S. Oaks and H. Wong. `Java` *Threads—Third Edition*. O'Reilly, 2004.

[UFM15] R.-G. Urma, M. Fusco, and A. Mycroft. *Java 8 In Action*. Manning, 2015.