10. Programmation parallèle et concurrente avec Java

Notions de base de la programmation concurrente et parallèle en Java.

Notamment la bibliothèque `java.util.concurrent`.

Il faut au préalable présenter les lambda-expressions, introduites en Java 7.0.

10.1 Lambda-expressions

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*

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

`Runnable` vs. `Callable<V>`

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

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

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

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

L'interface `Future<V>`

```
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.

De nombreuses interfaces de ce genre (≈ 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 effective
   Runnable r2 = () -> System.out.println( "x = " + x );
                                                    ^
Lambdas.java:61: error: local variables referenced from a
                  lambda expression must be final or effective
   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 r
                                 ^
    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** qui peuvent être utilisées comme argument à des méthodes qui reçoivent des objets fonctionnels au lieu de lambda-expressions.

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 );

    ...
  }
}
```

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

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 créer des objets qui sont des *threads*.

Toutefois, *il est plutôt recommandé de réaliser un* thread *en mettant en oeuvre l'interface* `Runnable`

```
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!
```

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»

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 un appel à la méthode `run()`.
3. On attend la fin de l'exécution de l'objet `Thread` en appelant `join()`.

Création et activation de *threads*

- 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 :

```
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

On suppose la procédure et les fonctions suivantes :

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

function ffoo( int num_thread, int a1 )
         returns int

function bar( int x )
         returns int
```

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.map(&:value)
```

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 π avec méthode Monte Carlo

```
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;
}

//
```

```java
@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;
}
```

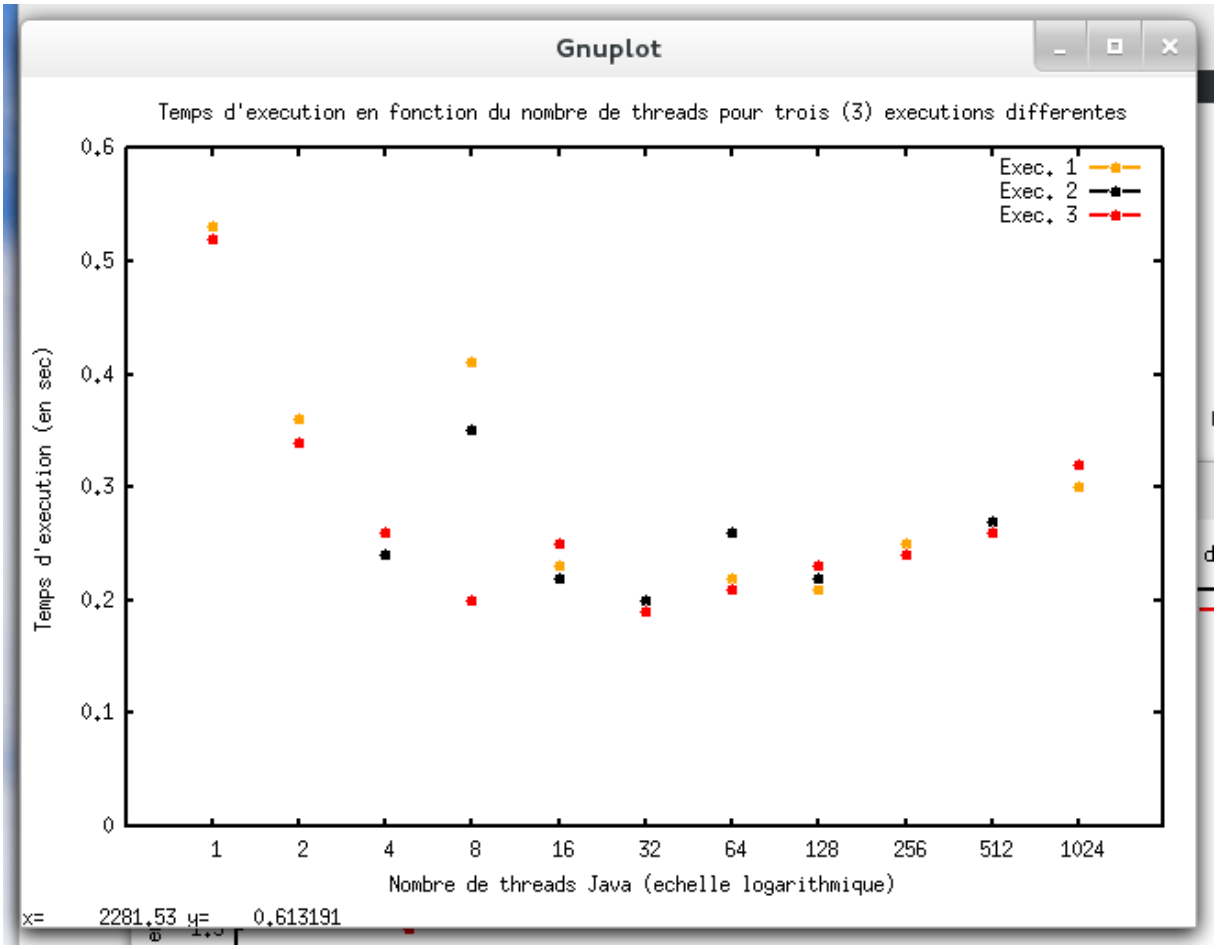
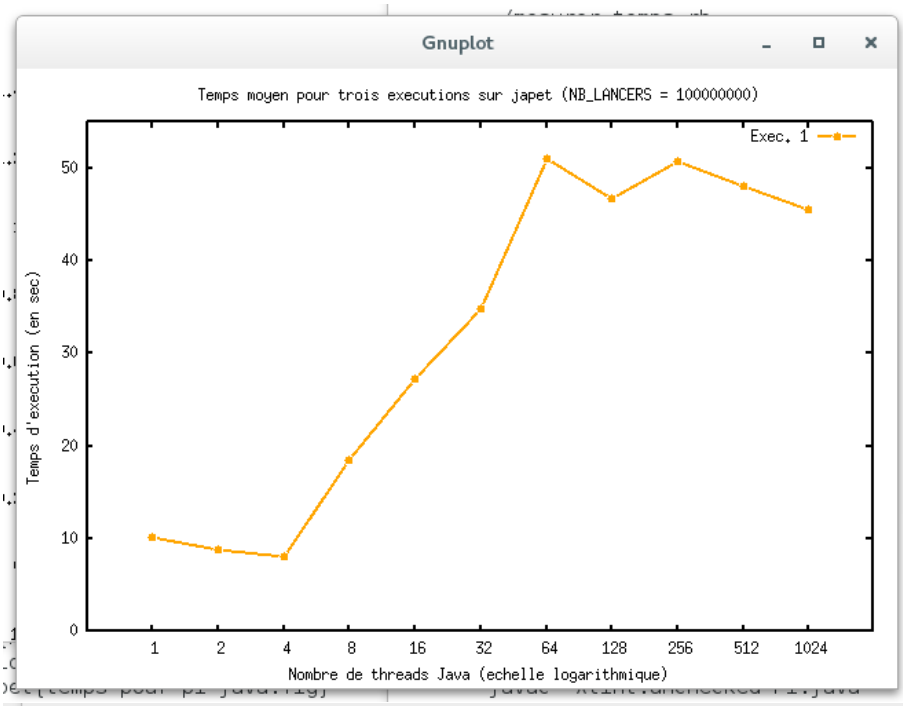
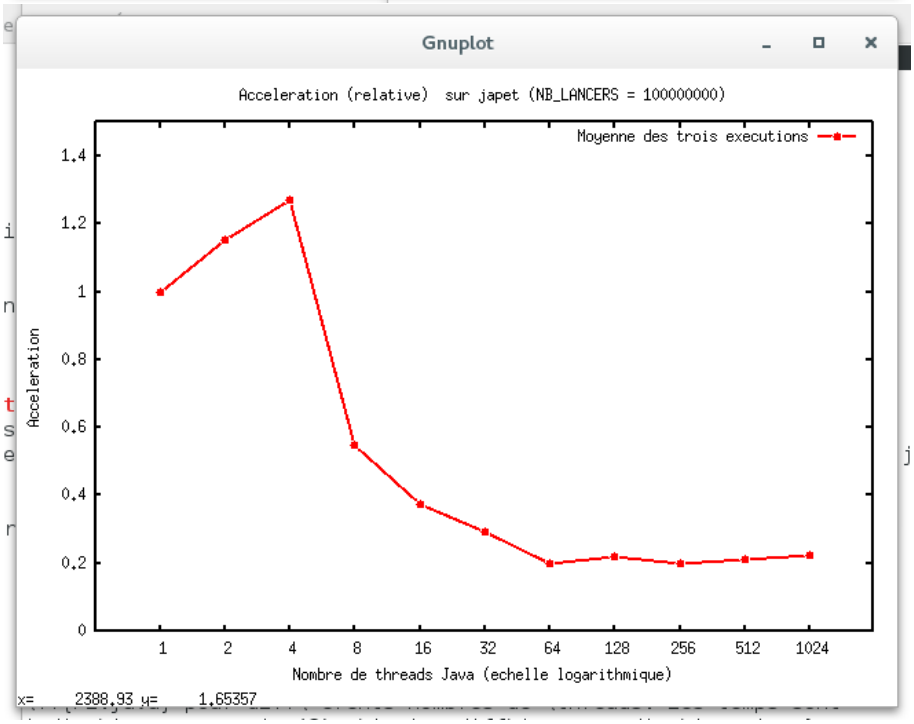


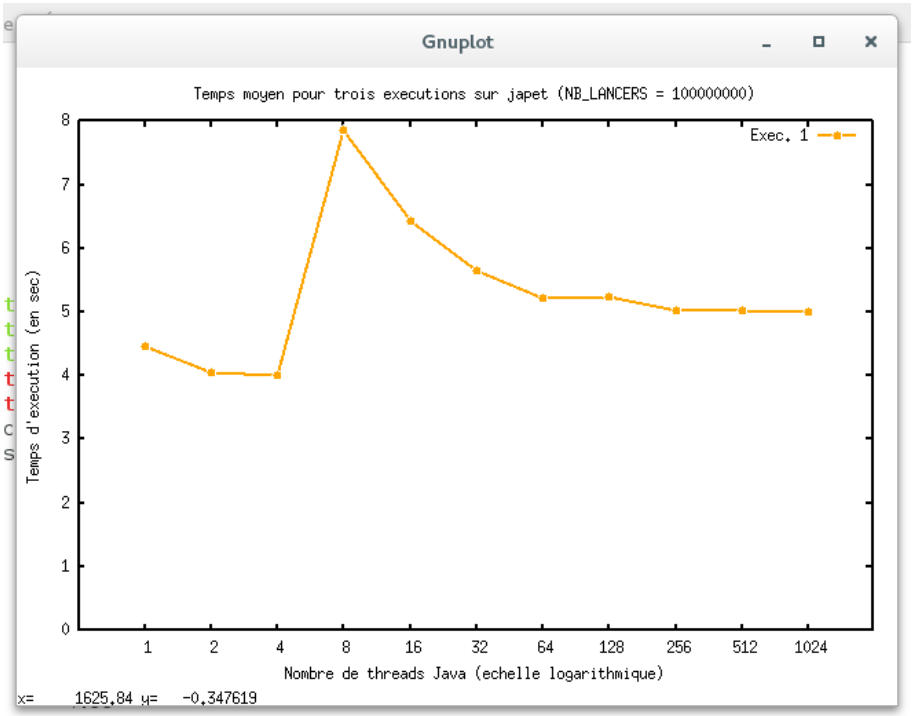
Figure 10.1: Graphe donnant le temps d'exécution du programme `Pi.java` pour différents nombres de *threads*. Les temps sont indiqués pour trois (3) séries différentes d'exécution.

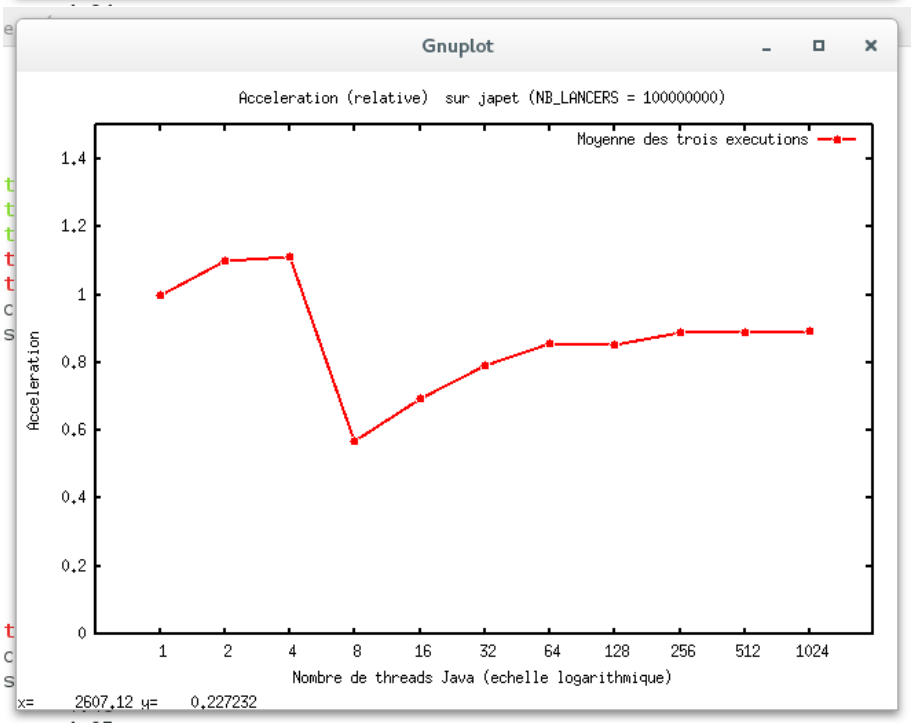
Autres graphes : Temps et accélération sur `japet` pour 100 000 000 lancers





Autres graphes : Temps et accélération sur `linux` pour 100 000 000 lancers



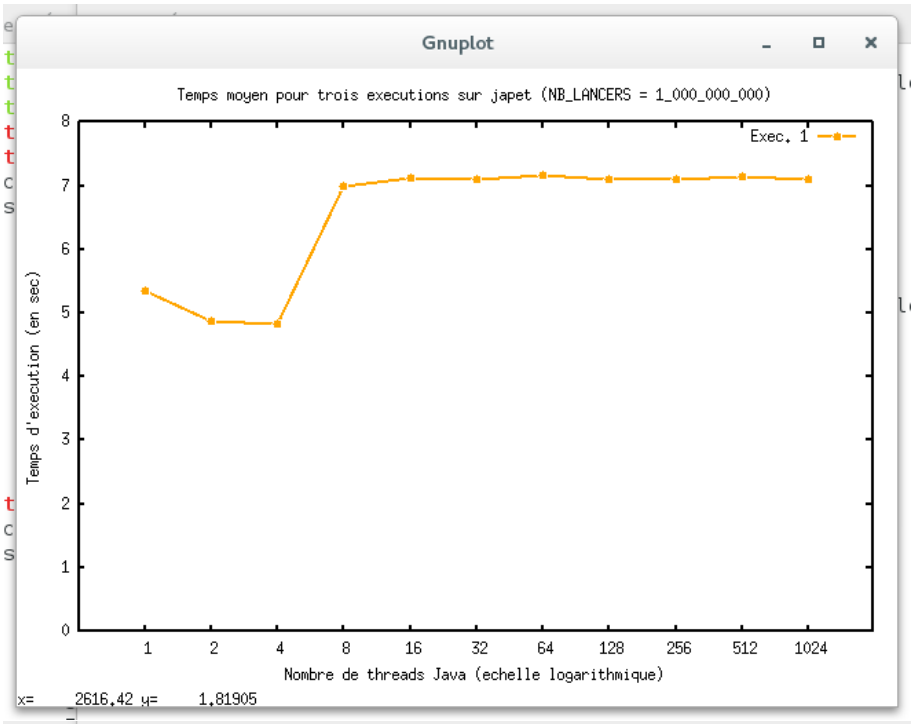


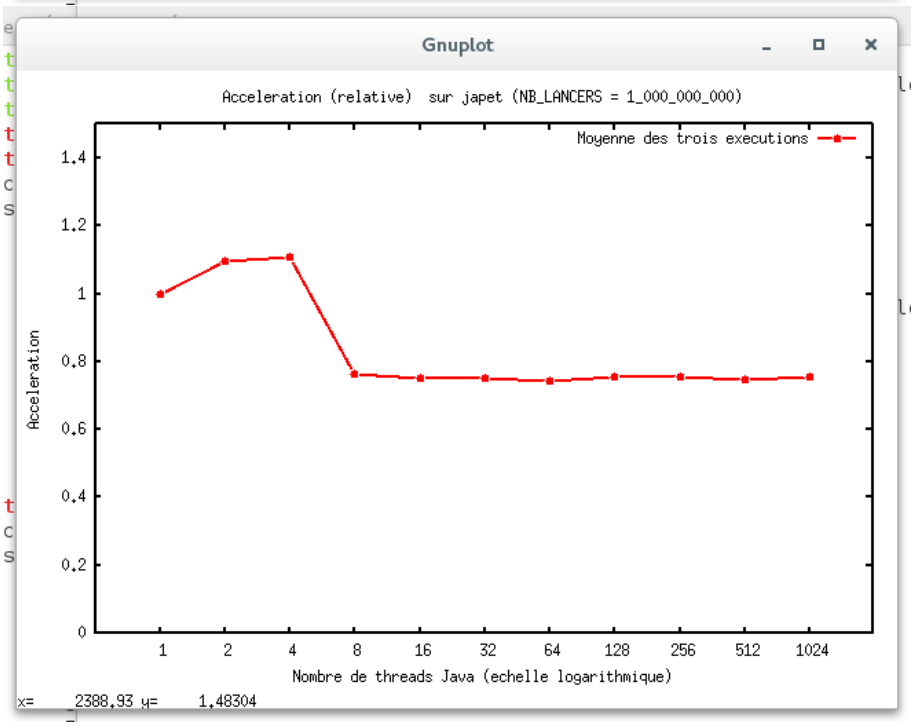
Un problème = `Random` est *thread-safe*... mais est partagé par les différents *threads* ☹

Solution :

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

Autres graphes : Temps et accélération sur `linux` pour 1 000 000 000 lancers





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

Nous allons revoir divers patrons de programmation parallèle vus en PRuby, **mais cette fois en Java**.

Le problème = somme de deux tableaux `a` et `b`.

Note : Étant donné que nous utiliserons Java «de base» — notamment `java.util.concurrent` — nous utiliserons donc du parallélisme *fork–join*.

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

Question : Comment?

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 10.4 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 paralleli
// 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

Question : Comment?

Programme Java 10.5 Parallélisme à granularité (très!) fine avec `Threads` — un *thread* par position du tableau.

```
//
// Parallelisme style fork-join a granularite fine avec Th
//
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 10.6 Parallélisme à granularité (très!) fine avec des `Futures` — un *thread* par position du tableau.

```
//
// Parallelisme style fork-join a granularite fine avec Fu
//
@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*

Question : Comment?

Programme Java 10.7 Parallélisme à granularité grossière avec répartition statique par blocs d'éléments adjacents.

```
//
// Parallelisme style fork-join a granularite grossiere av
// 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 10.8 Parallélisme à granularité grossière avec répartition statique et cyclique des éléments.

```
//
// Parallelisme style fork-join a granularite grossiere av
// 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*

Question : Comment?

Programme Java 10.9 Parallélisme à granularité grossière avec attribution dynamique par blocs d'éléments adjacents et *pool de threads «standard»* (`newFixedThreadPool`).

```
//
// Parallelisme style Coordonnateur-Travailleurs -- donc a
// repartition dynamique des elements -- et utilisant au p
// 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 10.9 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 a
// repartition dynamique des elements -- et utilisant au p
// nbTravailleurs threads, mais avec une facon differente
// 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 10.10 Parallélisme à granularité grossière avec attribution dynamique par blocs d'éléments adjacents et *pool de threads hyper-légers* (`ForkJoinPool`).

```
//
// Parallelisme style Coordonnateur-Travailleurs -- donc a
// repartition dynamique des elements -- et utilisant au p
// 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;
}
```

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

	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`.

Temps obtenus avec «`time -p`», pour 10 000 000 entiers, avec 8 *threads* sur une machine avec 8 coeurs.

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

Séquentiel : 0.08
Adjacents : 0.13

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.

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

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

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

Question : Ok ou pas ok si utilisé par plusieurs *threads*?

Pour éviter les situations de compétition, il faut assurer que `inc()` puisse se faire de façon atomique et exclusive.

```
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 être utilisée de deux façons possibles :

- Dans la signature d'une méthode
- Comme une instruction explicite

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

La solution qui suit est équivalente, bien que la précédente soit plus **explicite** (signature!) :

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

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

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

Note : Semblable, donc, à ce qu'on ferait en Ruby.

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

À tout objet est aussi implicitement associée une variable de condition.

Trois méthodes de base :

- `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 10.11 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();
  }
}
```

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

Exercice 10.3: Que fait un sémaphore?

Quelques remarques :

- `wait() = this.wait()`

  `notify() = this.notify()`

- Un appel à `wait()` doit se faire dans un `try`!

- Lors d'un appel à `o.wait()`, `o.notify()` ou `o.notifyAll()`, le *thread* doit avoir le contrôle du verrou de `o`.

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

Programme Java 10.12 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*

- L'avortement d'un *thread* peut être nécessaire :
  - l'usager avorte l'exécution
  - plusieurs *threads* sont lancés pour trouver un résultat et le résultat a été trouvé
- 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`, sans le resetter
- `Thread.interrupted()` met le *flag* **du *thread* courant** à `false` et retourne l'ancienne valeur
- `t.interrupt()` : met le *flag* à true, sauf si `t` est engagé dans un `wait()`, `sleep()` ou `join()`.

  Ceci n'implique pas que le *thread* termine immédiatement son exécution.

  Une vérification du statut est effectuée lors de `wait()`, `join()` et `sleep()`.

  ET : un *thread* en attente pour un verrou ne réagira pas à l'interruption tant que bloqué.

10.9 Priorités des *threads*

- Valeur comprise entre `Thread.MIN_PRIORITY` (1) et `Thread.MAX_PRIORITY` (10)
- Par défaut, un thread a la priorité de son parent (`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*.

- 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 s'il n'y a rien d'autres à faire

- La politique exacte d'ordonnancement est «*implementation dependant*».

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

  Toutefois :

  – Inversion de priorité :
  – Niveaux de priorité de Java vs. niveaux de priorité du système d'exploitation :

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

10.10 Quelques autres méthodes de la classe `Thread`

- `Thread.currentThread()` : retourne une référence au *thread* courant
- `t.sleep( long milis )` : endort le *thread*

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

- Constantes : `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORI`
- Constructeurs : Huit (8) variantes différentes
- 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.

  Mais : le modèle de mémoire de Java permet à un *thread* de conserver dans sa mémoire locale 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**.

- Exemple, où un *thread* exécute `traiter()` et un autre *thread* exécute `signalerFin()` :

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

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

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

Question : Ok? Pas ok?

- Solution = ajouter l'attribut `volatile` :

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

  Effet de `volatile` = à chaque lecture ou écriture, un vrai accès à la mémoire sera effectué.

- 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, les copies des variables du *thread* sont mises à jour.

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 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`

10.12 Traitement des exceptions

Si une exception est signalée et que cette exception se propage au-delà de la méthode `run`, alors l'exécution du *thread* est considérée comme terminée.

Par défaut, une exception ainsi propagée sans être traitée est traitée par la méthode `uncaughtException()` :

```
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, imprime la trace de la pile d'activation.

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.

10.13 `Streams` de Java 8.0

10.13.1 Tri unique des mots d'un fichier

10.13.2 Dénombrement des mots d'un texte

10.13.3 Génération des n premiers nombres de Fibonacci

```
        .peek( p -> System.out.println( p[0]

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

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

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

```ruby
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
```

```
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

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

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

Descriptions et exemples tirés du *Web* :

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

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.
}
```

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

10.A.2 Classe ReentrantLock

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

  int getHoldCount()
  protected Thread getOwner()
  protected Collection<Thread> getQueuedThreads()
  int getQueueLength()
  protected Collection<Thread> getWaitingThreads(C
  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.
  ...
}
```

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, Ti
  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

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

  int getNumberWaiting()
  // Returns the number of parties currently waiti

  int getParties()
  // Returns the number of parties required to tri

  ...

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

10.A.6 Classe CountDownLatch

```
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 l
  // to zero, unless the thread is interrupted, or
  // time elapses.

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

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

10.A.7 Classe Exchanger<V>

```
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

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

  int addAndGet(int delta)
  // Atomically adds the given value to the curren
  // 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 curren
  // 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 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 update
  // value if the current value == the expected va

  V get()

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

  void set(V newValue)

  String toString()
}
```

```
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>

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

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

Future<V>

```
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

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

10.A.10 *Pool de threads*

Interface `Executor`

```
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 f

  <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
  // completed successfully (i.e., without throwing an excep

  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

```
Class ThreadPoolExecutor implements Executor, Exec

  ThreadPoolExecutor(int corePoolSize,
                     int maximumPoolSize,
                     long keepAliveTime,
                     TimeUnit unit,
                     BlockingQueue<Runnable> workQ
  // Creates a new ThreadPoolExecutor with the giv
  // 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 tha
  // actively executing tasks.

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

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

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

    ...

    void shutdown()
    // Initiates an orderly shutdown in which previo
    // tasks are executed, but no new tasks will be

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

}
```

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

Méthodes de fabrication pour créer un `ExecutorService`

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

```
class Executors {
  // Methods that create and return an ExecutorSer
  // set up with commonly useful configuration set
  ...
  static ExecutorService
         newCachedThreadPool()
  // Creates a thread pool that creates new thread
  // as needed, but will reuse previously construc
  // 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 q

  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.
}
```

10.A.11 Classe ThreadLocal<T>

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

  Object get()
  // Returns the value in the current thread's cop
  // 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
  // variable to the specified value.
}
```

Exemple :

```
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` [. . .]
>
> However, even though all operations are thread-safe, retrieval operations do not entail locking.
>
> Retrieval operations generally do not block, so **may overlap with** update operations.
>
> 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-ha`

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

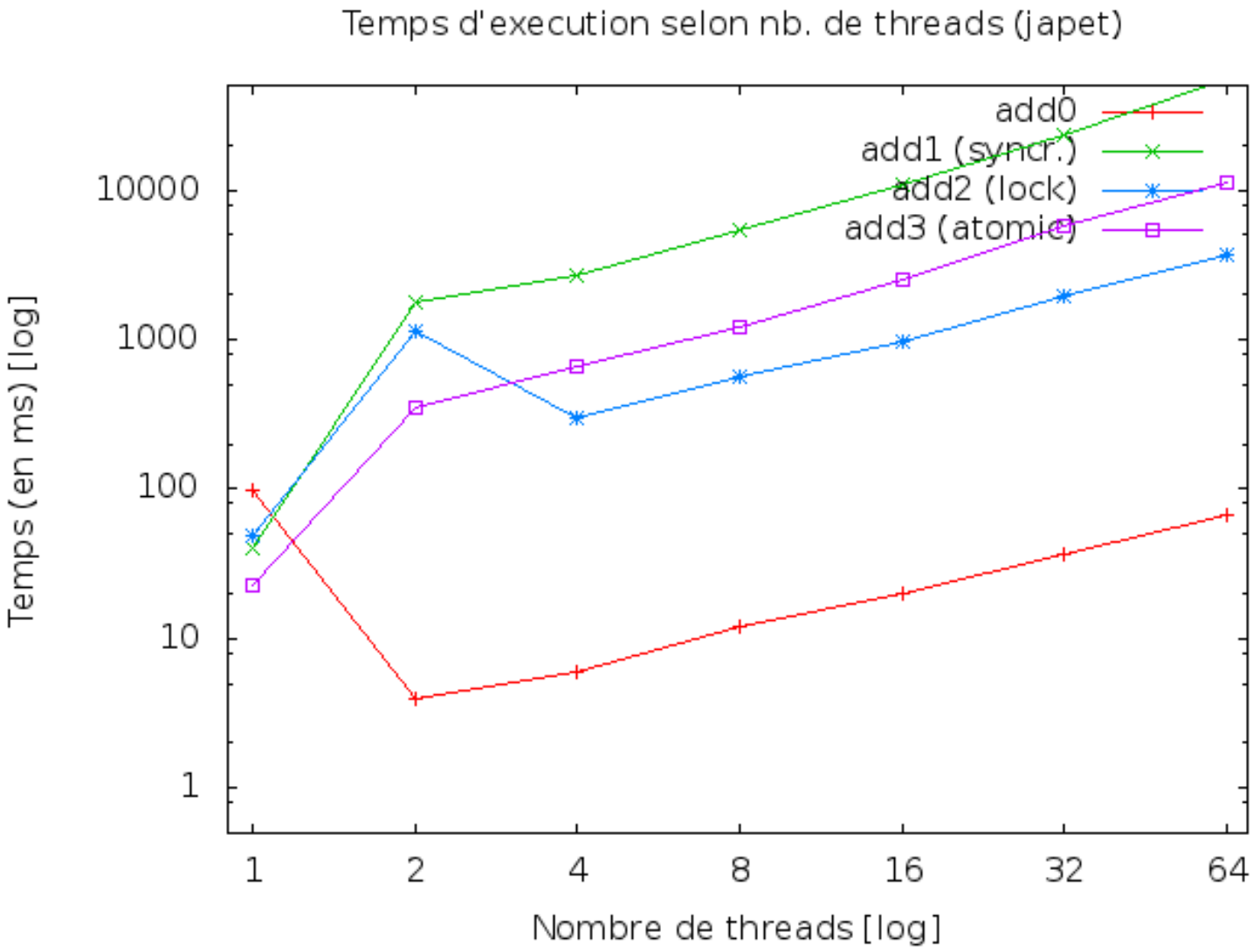
Compare les performances relatives entre `synchronized`, `ReentrantLock` et `AtomicInteger`.

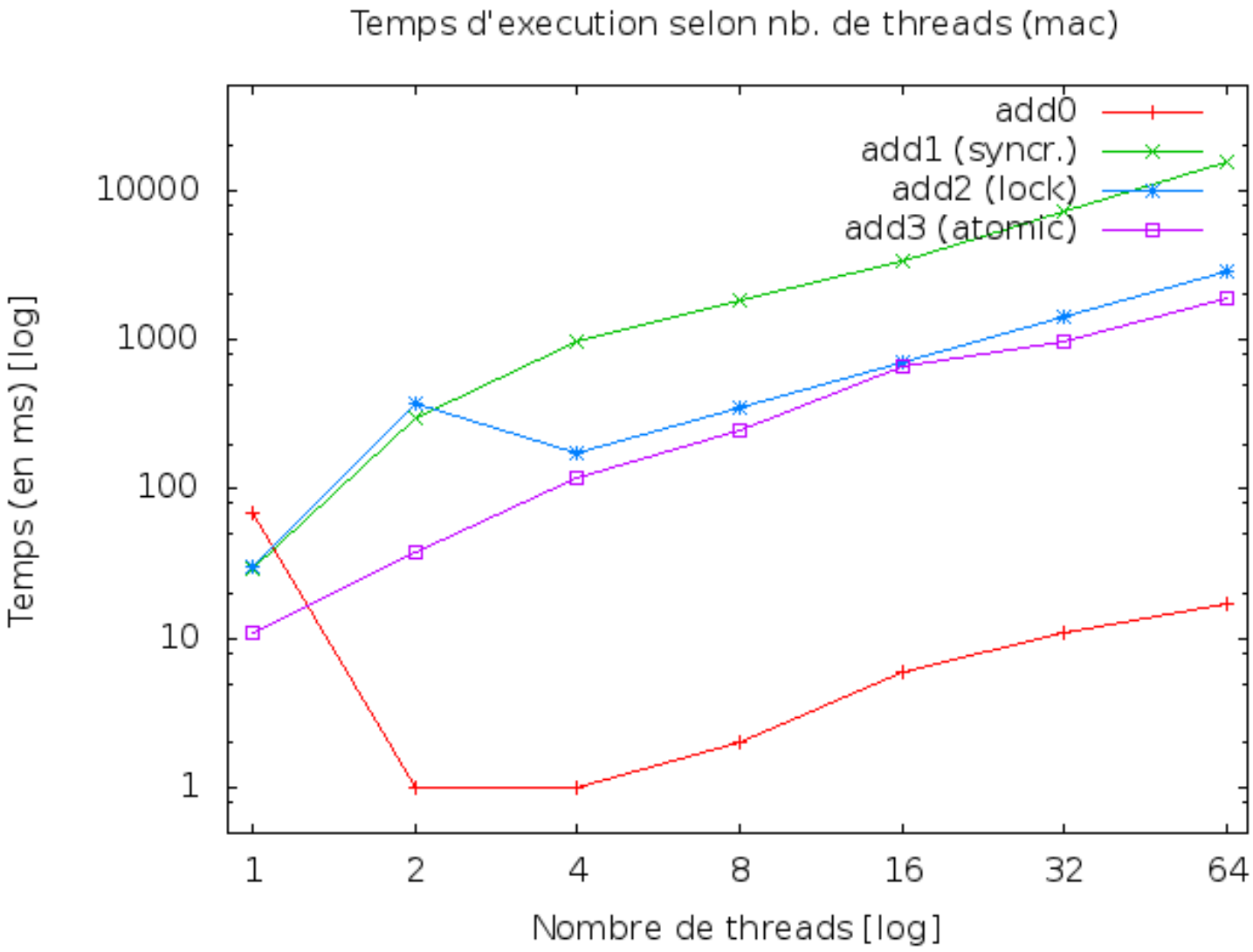
Programme Java 10.13 Une classe `VarInteger`.

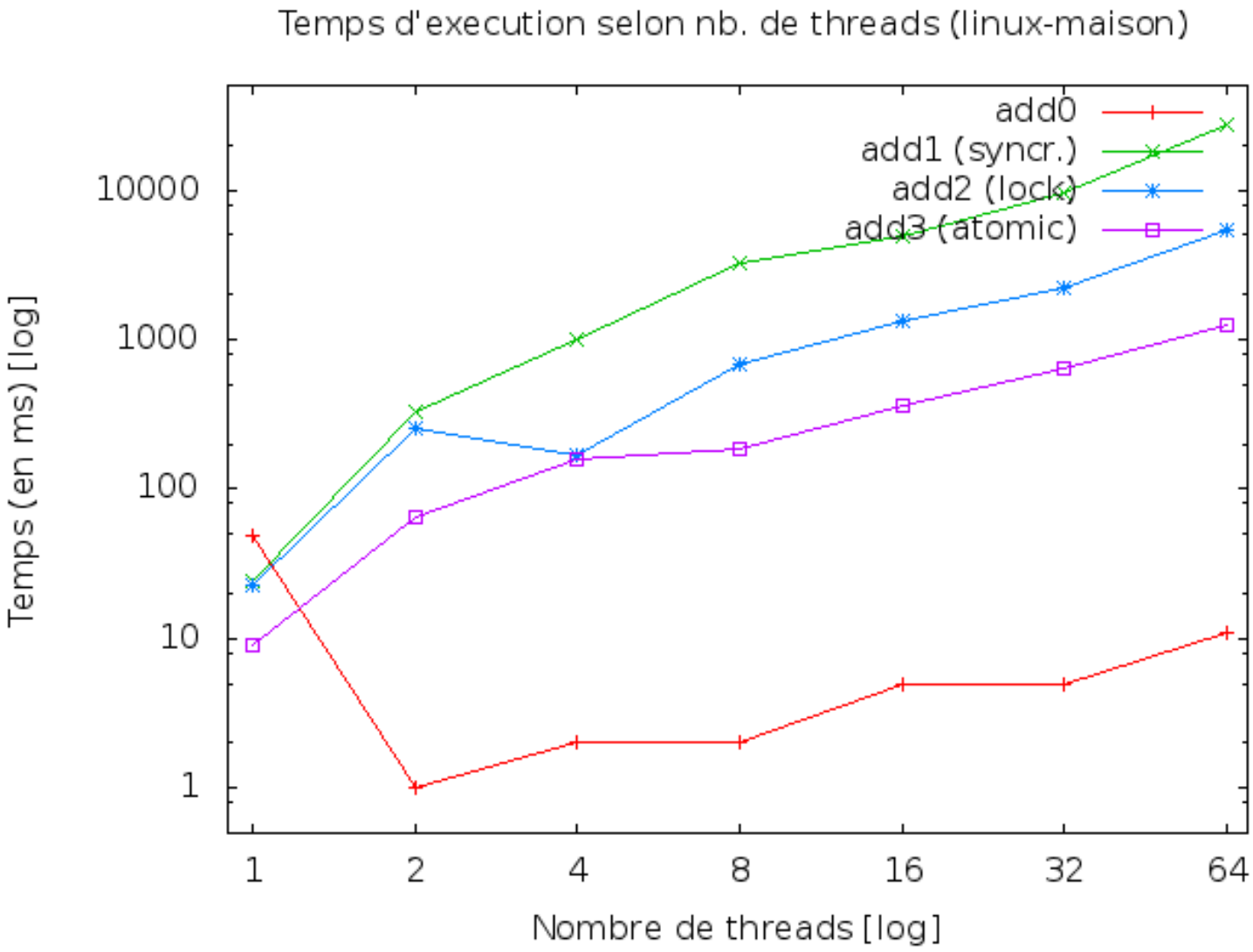
```
// Modelise une variable contenant un entier.
class VarInteger {
  private int val = 0;
  private final ReentrantLock lock; // Pour add2.
  private AtomicInteger atomicVal; // Pour add3.

  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; }
}
```

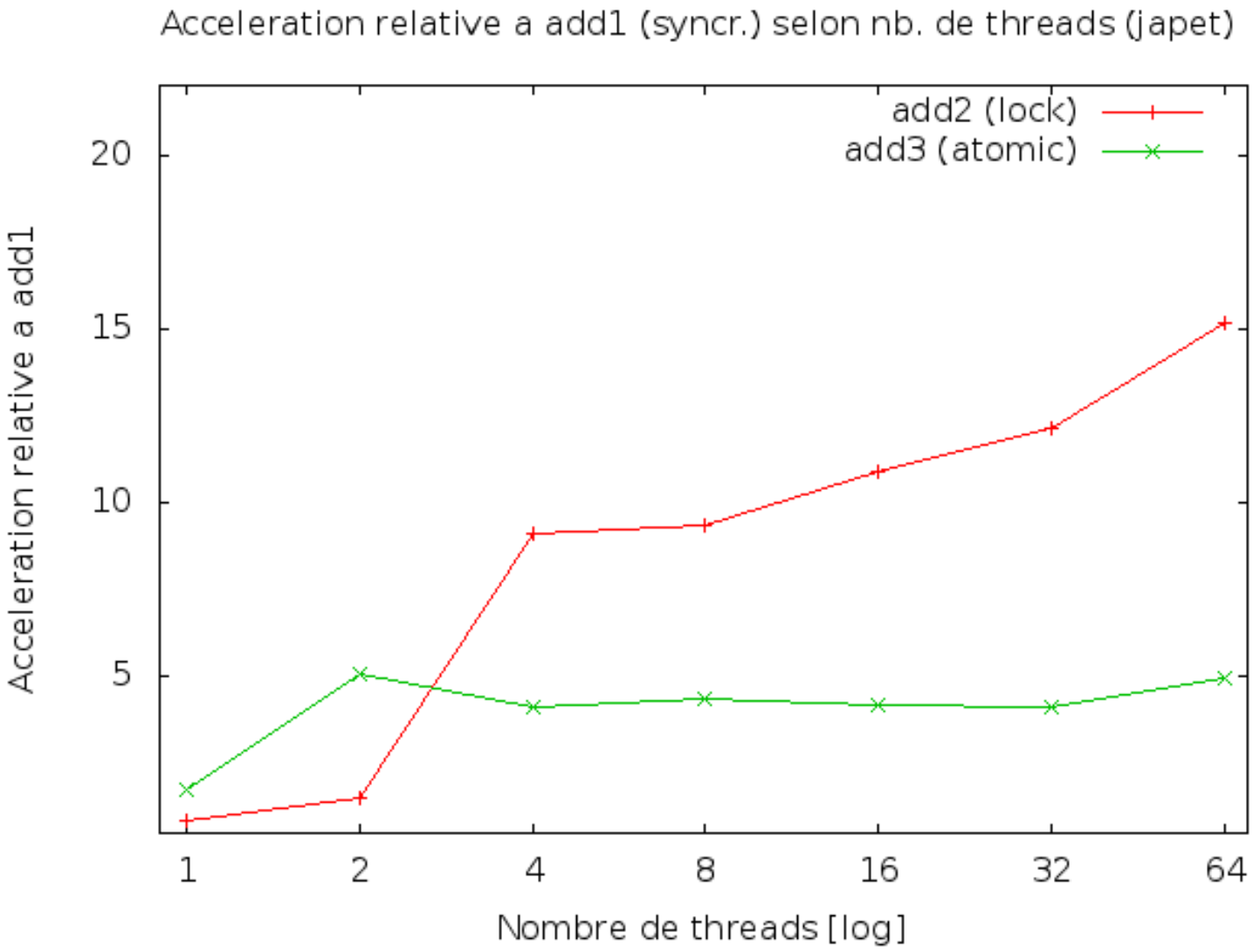
Programme Java 10.14 Un programme de *benchmark* utilisant `VarInteger`.

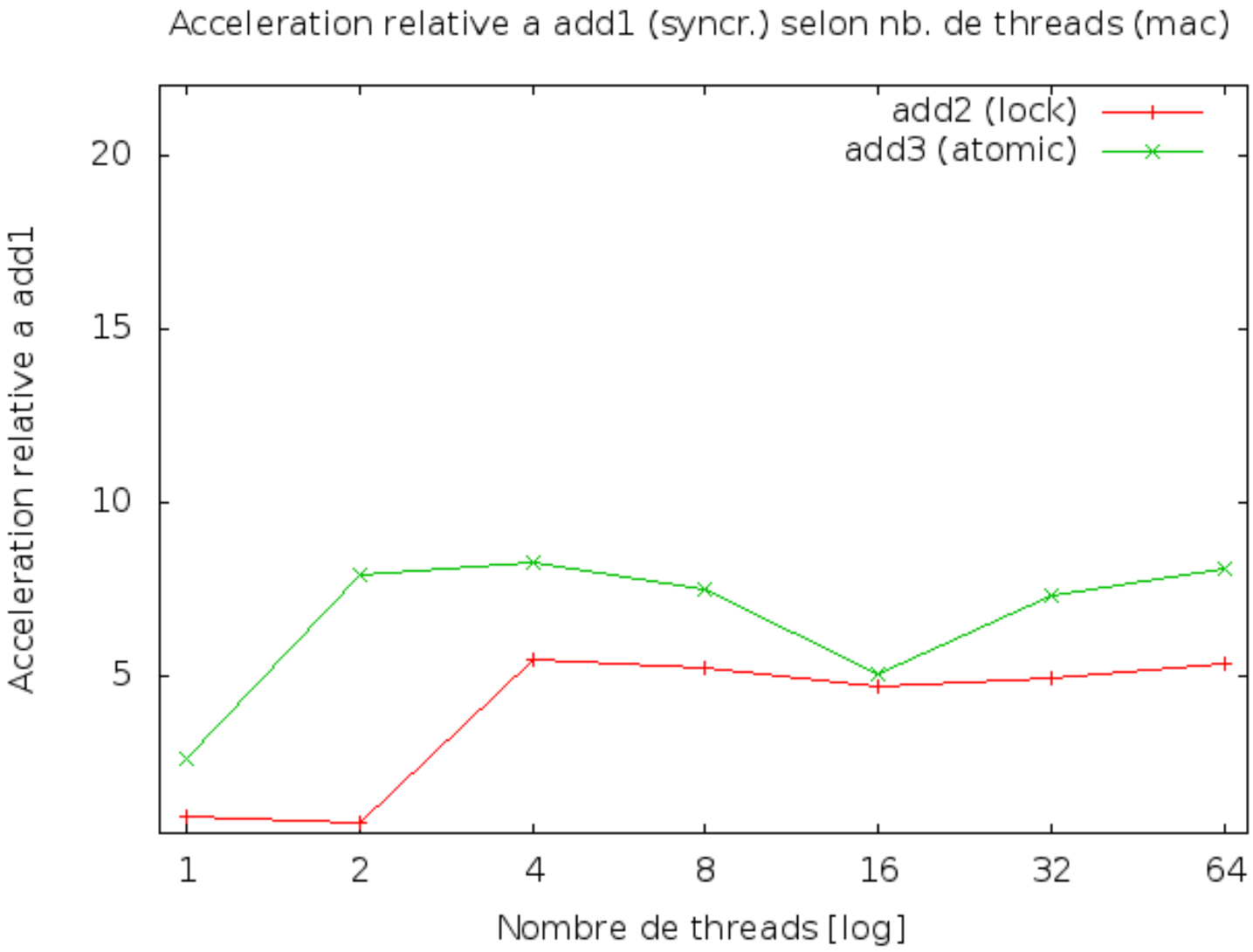
```
// On alloue un tableau.
int a[] = new int[nbElements];
for( int i = 0; i < nbElements; i++ ) { a[i] = 1; }

// 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" );
}
```

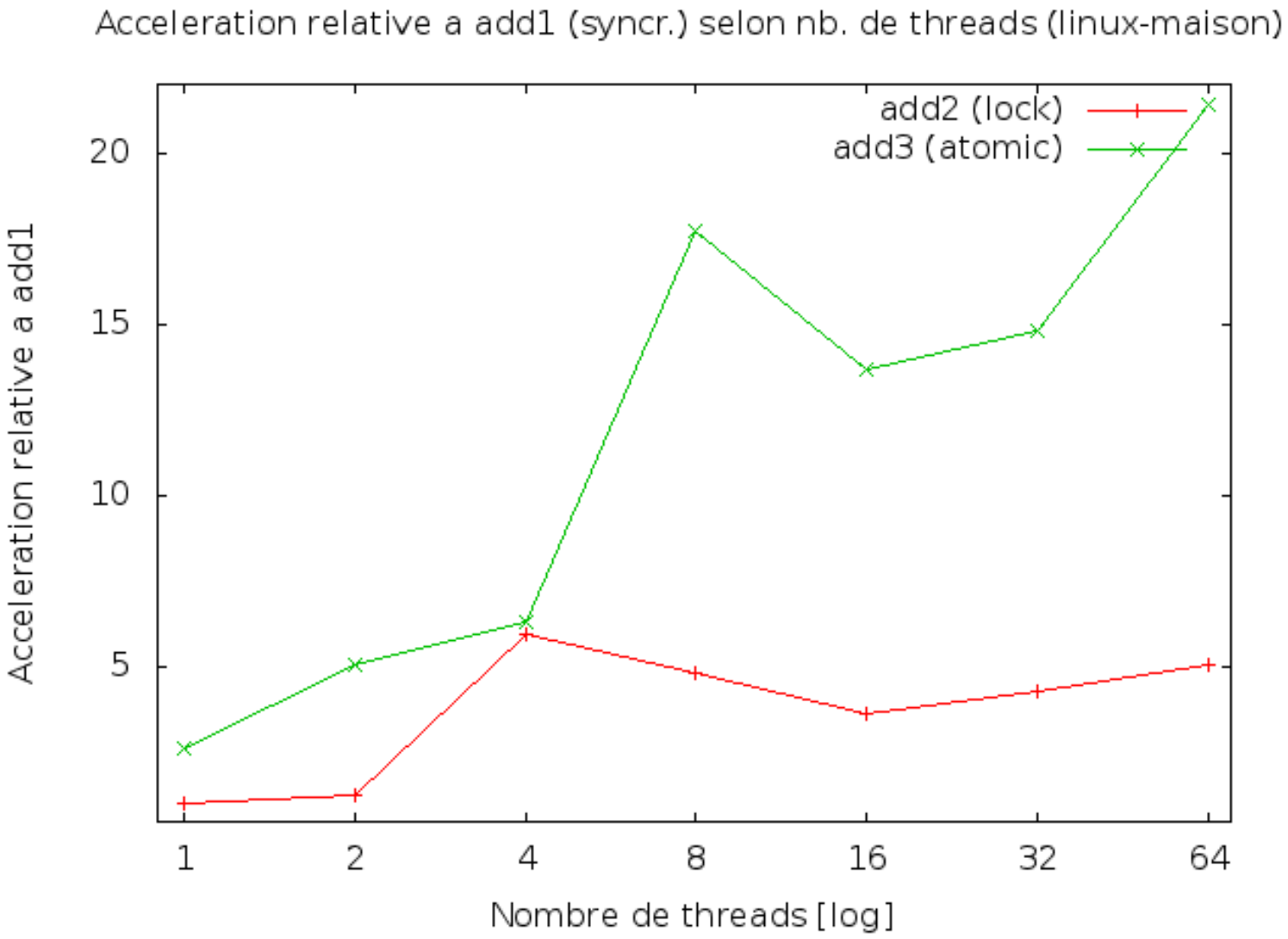

Temps d'execution selon nb. de threads (japet)
add0
add1 (syncr.)
add2 (lock)
add3 (atomic)
10000
1000
100
10
1
Temps (en ms) [log]
1
2
4
8
16
32
64
Nombre de threads [log]


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`)

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 10.15 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 certains 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>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Generiques.java:32: error: illegal start of type
    Pair<String,String>[] a = (Pair<>[]) new Pair<String,Str
```

```
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 Pa
                                                      ^
  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 :

```
@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*

Programme Java 10.16 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 = 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`.

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).

Programme Java 10.17 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, seui
    SommationRT droiteRT = new SommationRT(a, mid+1, sup, se

    // 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);
```

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`.

```
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 complet
  // results in RuntimeException or Error, not ExecutionExce
  // and that interrupts of the calling thread do not cause
  // the method to abruptly return by throwing InterruptedEx
}
```

Figure 10.2: Quelques méthodes des classe `RecursiveTask` et `ForkJoinTask`.

Remarques concernant `ForkJoinPool`

Quelques remarques additionnelles concernant `ForkJoinPool`)

- 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* où chaque cellule assure une **synchronisation** entre producteur (unique) et consommateurs de la cellule.

Les cellules ont le comportement suivant :

- Initialement, toutes les cellules sont **vides**.

- Une cellule peut être lue avec `get`.
  Lorsque vide, le *thread* appelant **est mis en attente**.

- Une cellule d'une `IStructure` ne peut être écrite, avec `put`, **qu'une seule et unique fois**.

Programme Java 10.18 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-Struct
   */
  void put( int i, T v )
       throws AlreadyFullException;

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

class AlreadyFullException
      extends RuntimeException{}
```

Programme Java 10.19 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 10.20 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 ){}
}
```

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

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

Programme Java 10.21 La mise en oeuvre de `put` : À compléter!

```
public void put( int i, T v ) {









}
```

Programme Java 10.22 La mise en oeuvre de `get` : À compléter!

```
public T get( int i ) {









}
```