Outils pratiques pour le multitâche

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

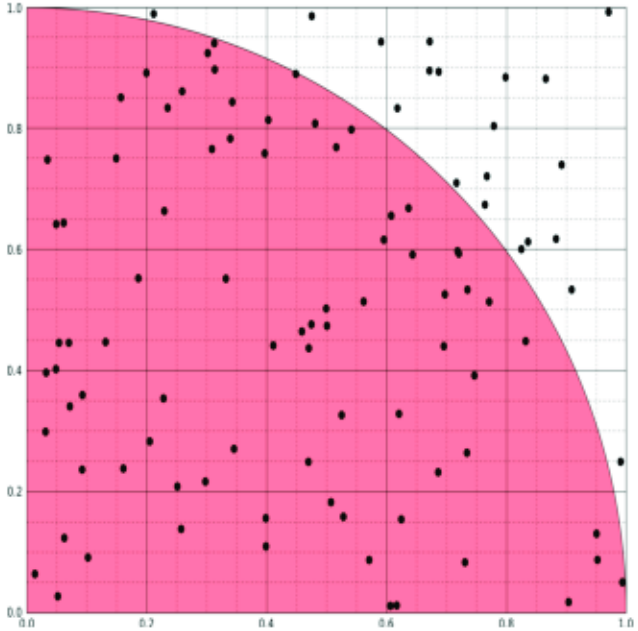
Version du 14 octobre 2018

Exemple : estimation de $\pi/4$ par Monte-Carlo (1/2)

```
class Tirages implements Runnable {
  final long nbTirages;                 // à réaliser par chaque tâche
  long tiragesDansLeDisque = 0 ;        // obtenus par chaque tâche

  Tirages(long nbTirages){
    this.nbTirages = nbTirages;
  }

  public void run(){
    double x, y;
    for (long i = 0; i < nbTirages; i++) {
      x = ThreadLocalRandom.current().nextDouble(1);
      y = ThreadLocalRandom.current().nextDouble(1);
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
  }
}
```



C'est un Runnable tout-à-fait naturel

Exemple de Callable

```
class Tirages implements Callable<Long>{
  final long nbTirages ;
  long tiragesDansLeDisque = 0 ;
  Tirages(long nbTirages){
    this.nbTirages = nbTirages;
  }

  public Long call(){
    for (long i = 0; i < nbTirages; i++) {
      double x = ThreadLocalRandom.current().nextDouble(1);
      double y = ThreadLocalRandom.current().nextDouble(1);
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
    return tiragesDansLeDisque;
  }
}
```

Notez le « return » dans la méthode call().

Utiliser un service de completion

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
CompletionService<Long> ecs =
                    new ExecutorCompletionService<Long>(executeur) ;

for (int j = 0; j < 10 ; j++) {
  ecs.submit( new Tirages( nbTirages/10 ) );
}

int tiragesDansLeDisque = 0;
for (int j = 0; j < 10; j++) {
  tiragesDansLeDisque += ecs.take().get(); // take() est bloquant
}

double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

Un problème d'atomicité

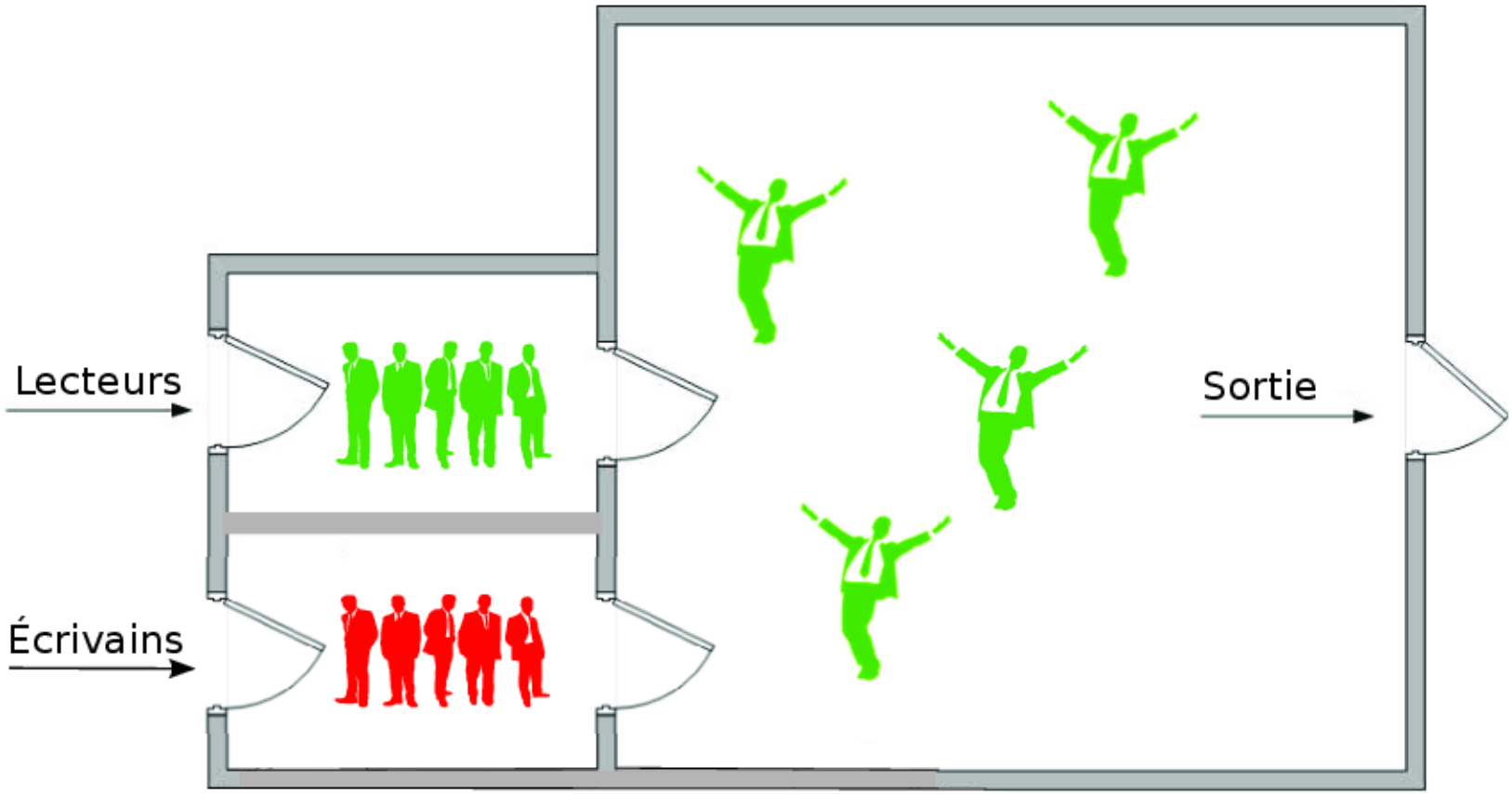
Utiliser une collection concurrente ou synchronisée ne garantit pas de produire un code correct, car **la composition d'instructions atomiques ne forme quasiment jamais une suite d'instructions atomique.**

```
while(! maListe.isEmpty() ) {
  Element e = maListe.remove(0);
  System.out.println(e.m);
}
```

n'est pas thread-safe !

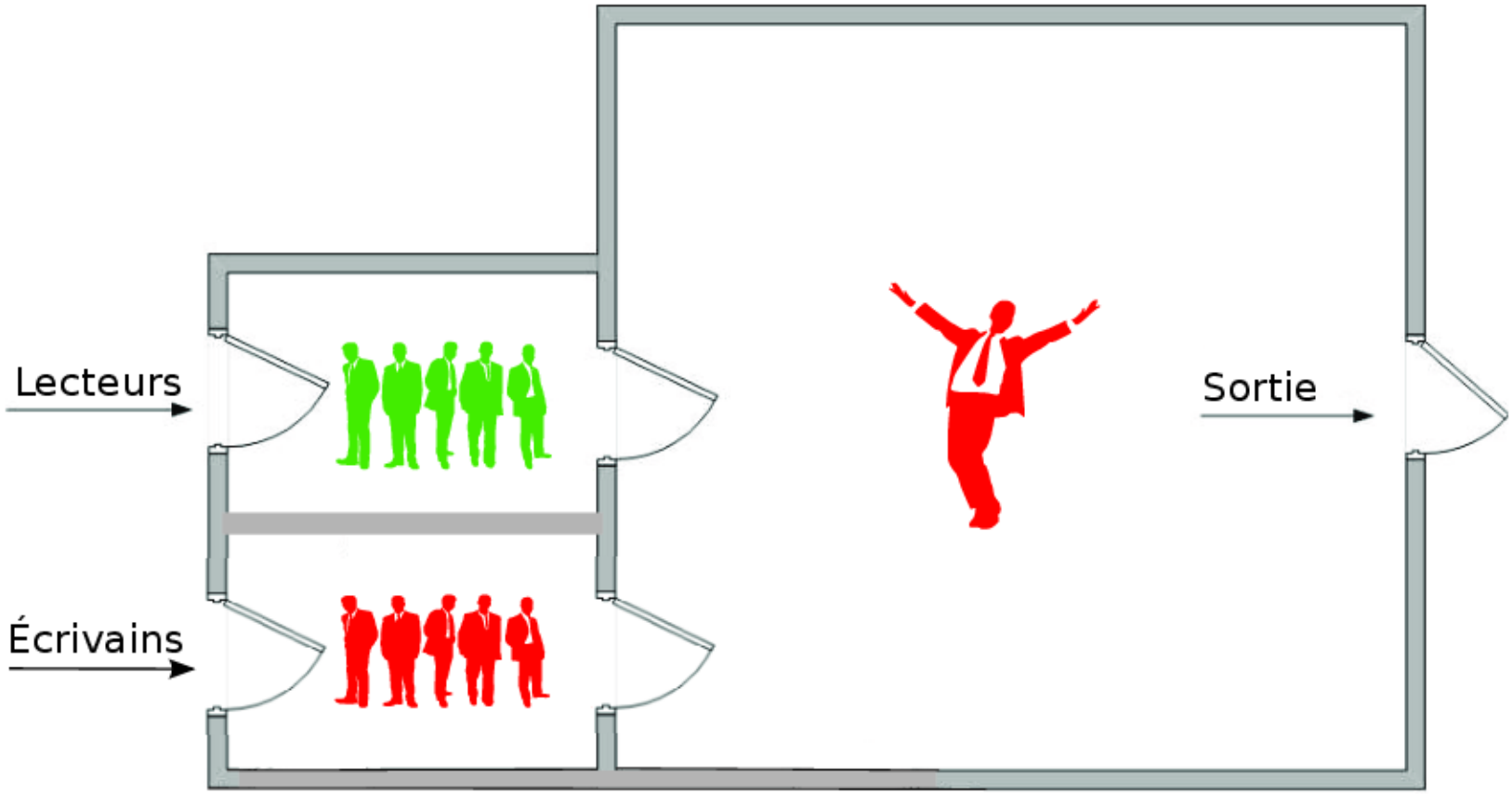
Même dans le cas où **`maListe`** est une instance d'une classe concurrente ou synchronisée, ce code pourra lever une exception *si la liste est vidée entre le moment du test dans la boucle* **`while`** *et la tentative de retrait de son premier élément.*

Fonctionnement d'un verrou de Lecture/Ecriture



Plusieurs threads peuvent posséder le verrou de lecture simultanément ! Ils sont alors implicitement autorisés à lire en parallèle les données protégées par ce verrou.

Fonctionnement d'un verrou de Lecture/Ecriture



Lorsqu'un thread possède le verrou d'écriture, aucun **autre** thread ne possède le verrou de lecture, ni le verrou d'écriture. Il pourra ainsi modifier de manière atomique les données protégées par ce verrou de lecture-écriture.

N.B. Ce thread pourra réclamer en sus le verrou de lecture : il possèdera alors les deux verrous !

Ce qu'il faut retenir

Java dispose de nombreux outils dédiés à la programmation multithread. Seuls quelques uns des plus importants ont été abordés ici.

L'emploi d'un « *threadpool* » permet de développer un code plus simple et plus élégant qui laisse à la JVM le soin de gérer les threads en charge d'exécuter la liste des tâches.

Il existe *trois catégories de collections* utilisables dans un contexte multithread ; chacune possède ses propres spécificités qui influent sur la manière de les employer. La documentation permet de connaître les usages corrects d'une collection donnée.

Outre les verrous intrinsèques des objets, il existe *d'autres verrous plus souples d'utilisation* et qui, parfois, offrent de meilleures performances. Nous verrons lors du prochain cours un *nouveau type de verrou*, le « stamped lock », qui produit assez souvent des résultats meilleurs que ceux observés avec les verrous de lecture-écriture.