Collection Concurrente

Rémi Forax
forax@univ-mlv.fr

java.util.concurrent

- Deux sortes de classes
  - Des classes utilitaires
    - Sémaphore
    - CountDownLatch
    - CyclicBarrier
    - Exchanger
  - Des collections concurrentes
    - BlockingQueue
    - ExecutorService
    - ScheduledExecutorService

Semaphore

- `java.util.concurrent.Semaphore`
- Permet de limiter le nombre d'accès à une ressource partagée entre différentes threads, en comptant le nombre d'autorisations acquises et rendues
- `Semaphore sema = new Semaphore(MAX, true)` crée un objet sémaphore équitable (`true`) disposant de `MAX` autorisations
- `sema.acquire()` pour "décompter" de 1, `sema.release()` pour "rendre" l'autorisation

Sémaphore (acquisition)

- `sema.acquire()`
  - la thread courante (t) tente d'acquérir une autorisation.
    - Si c'est possible, la méthode retourne et décrémente de un le nombre d'autorisations.
    - Sinon, t bloque jusqu'à ce que:
      - Soit la thread t soit interrompue (l'exception InterruptedException est levée, tout comme si le statut d'interruption était positionné lors de l'appel à acquire())
      - Soit une autre thread exécute un release() sur ce sémaphore. T peut alors être débloquée si elle est la première en attente d'autorisation (notion d'«équité» paramétrable dans le constructeur)

Sémaphores (acquisition)

- Tentative d'acquisition insensible au statut d'interruption
  - void acquireUninterruptibly() : si la thread est interrompue pendant l'attente, elle continue à attendre. Au retour, son statut d'interruption est positionné.
- Tentative d'aquistion non bloquante ou bornée
  - boolean tryAcquire() [ne respecte pas l'équité] et boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException

Sémaphore (relachement)

- `sema.release()`
  - incrémente le nombre d'autorisations. Si des threads sont en attente d'autorisation, l'une d'elles est débloquée
    - Une thread peut faire release() même si elle n'a pas fait de acquire() !
    - Différent des moniteurs

Sémaphores

- Sémaphore *binaire*: créé avec 1 seule autorisation
  - Ressource soit disponible soit occupée
    - Peut servir de verrou d'exclusion mutuelle
    - MAIS le verrou peut être «relâché» par une autre thread !
- Paramètre d'équité du constructeur: (*fair*)
    - Si false:
      - aucune garantie sur l'ordre d'acquistion des autorisations
      - Par ex: une thread peut acquérir une autorisation alors qu'une autre est bloquée dans un acquire() (*barging*)
    - Si true:
      - L'ordre d'attente FIFO est garanti (modulo l'exécution interne)

Sémaphores

- Surcharge des méthodes pour plusieurs autorisations
  - acquire(int permits), release(int permits), acquireUninterruptibly(int permits)...
- Manipulation sur le nombre d'autorisations
  - int availablePermits(), int drainPermits() et void reducePermits(int reduction)
- Connaissance des threads en attente d'autorisation
  - boolean hasQueuedThreads(), int getQueueLength(), Collection<Thread> getQueuedThreads()

Loquet avec compteur

- java.util.concurrent.CountDownLatch
- Initialisé (constructeur) avec un entier
- Décrémenté par la méthode countDown()
  (pas bloquant)
- Une ou plusieurs threads se mettent en attente sur ce loquet par await() jusqu'à ce que le compteur "tombe" à zéro.
- Le compteur ne peut pas être réinitialisé
  (sinon, CyclicBarrier)

Barrière cyclique

- java.util.concurrent.CyclicBarrier,même genre CountDownLatch, réinitialisable (reset())
- Constructeur accepte un Runnable spécifiant du code à exécuter par la dernière thread arrivée
- Par ex: si créé avec 3, chaque thread qui fait await() sur la barrière attend que 3 threads aient fait await()
- Chaque await() retourne l'ordre d'arrivée à la barrière
- Pour connaître l'état de la barrière
  - getParties() donne le nombre de threads concernées
  - getNumberWaiting() le nombre de threads en attente

Barrière cyclique

- ***Broken barrier*** (tout ou rien) dans le cas où
  - un thread quitte le point de barrière prématurément
    - Interruption, problème d'exécution, reset de la barrière
  - Dans ce cas, toutes les threads en attente sur await() lèvent BrokenBarrierException

Echangeur java.util.concurrent.Exchanger

- Exchanger<V> permet à 2 threads d'échanger deux valeurs d'un même type V
- `V exchange(V x) throws InterruptedException` est une méthode bloquante qui attend qu'une autre thread ait fait le même appel sur le même objet échangeur
  - À cet instant, chaque méthode retourne la valeur passée en argument par l'autre thread
  - Lève InterruptedException si la thread est interrompue
- `V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException` permet de borner l'attente

Collection et concurrence

- En java trois façons d'obtenir des collections concurrentes
  - Utiliser Hashtable et Vector (1.0)
    - Ts les appels sont synchronized
  - Utiliser Collections.synchronized*(...) (1.2)
    - Proxy dont ts les appels sont synchronized sur la collection prise en paramètre
  - Utiliser les collections concurrentes (1.5)
    - Pas de synchronized
    - Utilise les volatiles, atomique et les Locks

Synchronized et Collection

- Si un collection est synchronizée, son iterateur ne l'est pas (pas possible)

```
public static void main(String[] args) {
  final List<String> list=Collections.synchronizedList(
    new ArrayList<String>());
  new Thread(new Runnable() {
    public void run() {
      for(;;) {
        synchronized(list) {
          for(String s:list)          // utilise un iterateur
            System.out.println(s);
        }
        try {
          Thread.sleep(1000);
        } catch(...
      }
    }
  }).start();
}
for(int i=0;i<100000;i++)
  list.add(Integer.toString(i));
```

Collections concurrentes

- Ces collections possèdent des propriétés communes
  - Il est interdit de stocker null
  - Les bulks operations addAll(), removeAll(), retainAll() ne sont pas garantie d'être atomiques
  - Les itérateurs ne sont pas fail-fast,
    et donc renvoient jamais de ConcurrentModificationException

Collections concurrentes

- Marche sans synchronized
  - List
    - CopyOnWriteArrayList (snapshot iterator)
  - Set
    - CopyOnWriteArraySet (snapshot iterator)
    - ConcurrentSkipListSet (weakly consistent iterator)
  - Map
    - ConcurrentSkipListMap (weakly consistent iterator)
    - ConcurrentHashMap (weakly consistent iterator)

Les files d'attente (BlockingQueue)

- Interface qui hérite de java.util.Queue
- Opérations :
  - boolean offer(E o) ajoute si possible, sinon retourne false
    - boolean offer(E o,long timeout,TimeUnit unit) throws InterruptedException
  - E poll() retourne la tête et l'enlève, ou null si c'est vide
    - E poll(long timeout, TimeUnit unit) throws InterruptedException
  - E peek() retourne la tête, ou null si c'est vide, mais de l'enlève pas
  - int drainTo(Collection<? super E> c) retire tous les éléments dispo.
  - int drainTo(Collection<? super E> c, int maxElem) retire au plus maxElem éléments

Les files d'attente (BlockingQueue)

- Interface qui hérite de java.util.Queue
- Opération bloquantes
    - void put(E o) throws InterruptedException
    - E take() throws InterruptedException

ArrayBlockingQueue et LinkedBlockingQueue

- Implémentent BlockingQueue par tableau et liste chaînée
  - ArrayBlockingQueue peut être paramétré pour être "équitable" (*fair*) ou non dans l'accès des threads mises en attente
  - LinkedBlockingQueue peut être optionnellement bornée à la création par une capacité maximale
  - Implémentent Iterable: l'itération est faite dans l'ordre
    - "***weakly consistent***" itérateur qui ne lève jamais ConcurrentModificationException. Les modifications ultérieures à la création peuvent (ou non) être reflétées par l'itérateur.
  - int remainingCapacity() retourne la capacité restante ou Integer.MAX_VALUE si la file n'est pas bornée
  - int size() donne le nombre d'éléments dans la file

Autres files d'attente

- DelayQueue<E extends Delayed>
  - File d'attente non bornée dans laquelle ne peuvent être retirés les éléments que lorsque leur délai (getDelay()) a expiré
- SynchronousQueue<E>
  - File de capacité zéro dans laquelle chaque put() doit attendre un take() et vice versa (sorte de canal de rendez-vous)
  - Par défaut, pas équitable (fairness), mais spécifiable à la construction
- PriorityBlockingQueue<E>
  - File d'attente non bornée qui trie ses éléments selon un ordre fixé à la construction (un Comparator<? super E> ou éventuellement l'ordre naturel).
  - Attention: l'itérateur ne respecte pas l'ordre

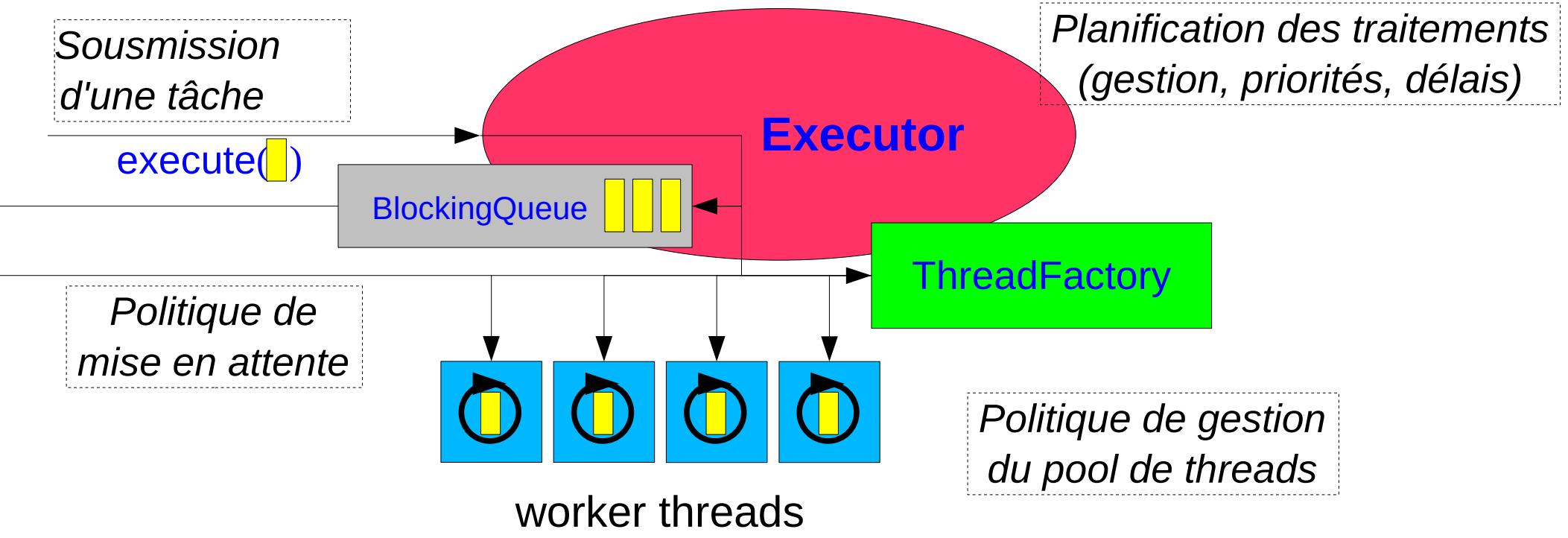
Les executors

- Interface Executor: permet de "gérer" des threads
  - Dissocier la soumission des traitements de leur exécution
  - Une seule méthode: void execute(Runnable command)
  - Plein d'implantations: Thread-pool, Scheduled pool, etc.
  - Différentes fabriques d'Executor dans la classe Executor**s**

  - Selon la classe concrète utilisée, le traitement planifiée est prise en charge
    - par un **nouveau** processus léger
    - ou par un processus léger "**réutilisé**"
    - ou encore par **celui qui appelle** la méthode execute()
    - et ce en **concurrence** ou de manière **séquentielle**...

Principe d'un Executor

- Un traitement (du code à exécuter) est "soumise"
- L'executor gère les traitements (exécution, mise en file d'attente, planification, création de threads...)
- Il est prise en charge par un "worker thread"



Interface ExecutorService

- Sous-classe d'Executor qui permet de gérer plus finement la planification des traitements

- Gère des traitements spécifiques :
  - l'objet représentant la cible du code à exécuter:
    Runnable devient une nouvelle interface Callable<T>
  - l'exécuteur gére en plus de la planification des différents traitements, leur terminaison, leur valeur de retour, la terminaison de l'exécuteur, etc.

Interface Callable<T>

- Callable<T> étend la notion de Runnable, en spécifiant un type de retour (de type T) et des exceptions :

```
package java.util.concurrent;
public interface Callable<T> {
    public T call() throws Exception;
}
```

- Executors.**callable**(Runnable task) permet de transformer un runnable en callable.

Executor & ExecutorService

- L'interface Executor
  - Demande l'exécution d'un runnable
    - void execute(Runnable r)
- L'interface ExecutorService hérite de Executor, ajoute les méthodes:
  - Demande l'exécution d'un callable
    - <T> Future<T> **submit**(Callable<T> task)
  - Demande l'exécution d'un runnable
    - Future<?> **submit**(Runnable task)
    - <T> Future<T> **submit**(Runnable task, T result)
      renvoie result si le traitement est effectué

Interface Future<T>

- Future<T> correspond à une valeur qui va être (ou est déjà) calculée par un traitement (calcul asynchrone)
  - Obtenir la valeur
    - T **get**() throws InterruptedException, ExecutionException, CancellationException
      - Méthode bloquante jusqu'à: a) terminaison normale b) thread interrompue c) levée d'exception à l'exécution d) tâche annulée
    - T **get**(long timeout, TimeUnit unit) throws InterruptedException,ExecutionException,TimeoutException CancellationException
      - Si **timeout** est écoulé, lève TimeoutException

InterruptedException est une exception dont la **cause** a été levées par la méthode call() du Callable.

TimeUnit

- Type énuméré représentant 1 durée dans une unité donnée
  - MICROSECONDS, MILLISECONDS, NANOSECONDS, SECONDS, MINUTES, HOURS, DAYS

  - Offre une méthode conversion
    `public long convert(long sourceDuration, TimeUnit sourceUnit)`
    par ex: `TimeUnit.MILLISECONDS.convert(10L, TimeUnit.MINUTES)`
    convertit 10 minutes en milisecondes

  - Méthodes raccourcis: `public long toNanos(long duration)`
    équivalent à: `NANOSECONDS.convert(duration, this)`

Arrêter un ExecutorService

- void **shutdown**() continue à exécuter les tâches déjà planifiées, mais plus aucune tâche ne peut être soumise
- List<Runnable> **shutdownNow**() tente d'arrêter activement (typiquement Thread.interrupt()) et/ou n'attend plus les tâches en cours, et retourne la liste de celles qui n'ont pas commencé
- boolean **isShutdown**() vrai si **shutdown**() a été appelé
- boolean **isTerminated**() vrai si toutes les tâches sont terminées après un shutdown() ou shutdownNow()
- boolean **awaitTermination**(long timeout, TimeUnit unit) bloque jusqu'à ce que:
  - soit toutes les tâches soient terminées (après un shutdown(),
  - soit le timeout ait expiré,
  - soit la thread courante soit interrompue

Un exemple

- Temps d'accès à www.google.fr

```
final InetAddress host=InetAddress.getByName("www.google.fr");

ExecutorService executor = Executors.newCachedThreadPool();
Future<Long> future = executor.submit(new Callable<Long>() {
  public Long call() throws Exception {
    long time=System.nanoTime();
    host.isReachable(2000);
    return System.nanoTime()-time;
  }
});

executor.shutdown();

try {
  System.out.println("reach "+host+" in "+future.get()+" ns");
} catch (ExecutionException e) {
  throw e.getCause();
}
```

Soumettre plusieurs traitements

- Il est possible de soumettre plusieurs traitements en attendant l'exécution de **tous** ou d'**au moins un** traitement
  - En demandant l'attente de tous les traitements
    - <T> List<Future<T>> **invokeAll**(Collection<? extends Callable<T>> tasks)
    - <T> List<Future<T>> **invokeAll**(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)
  - En demandant la fin d'au moins un traitement avec succes
    - <T> T **invokeAny**(Collection<? extends Callable<T>> tasks)
    - <T> T **invokeAny**(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

Un autre exemple

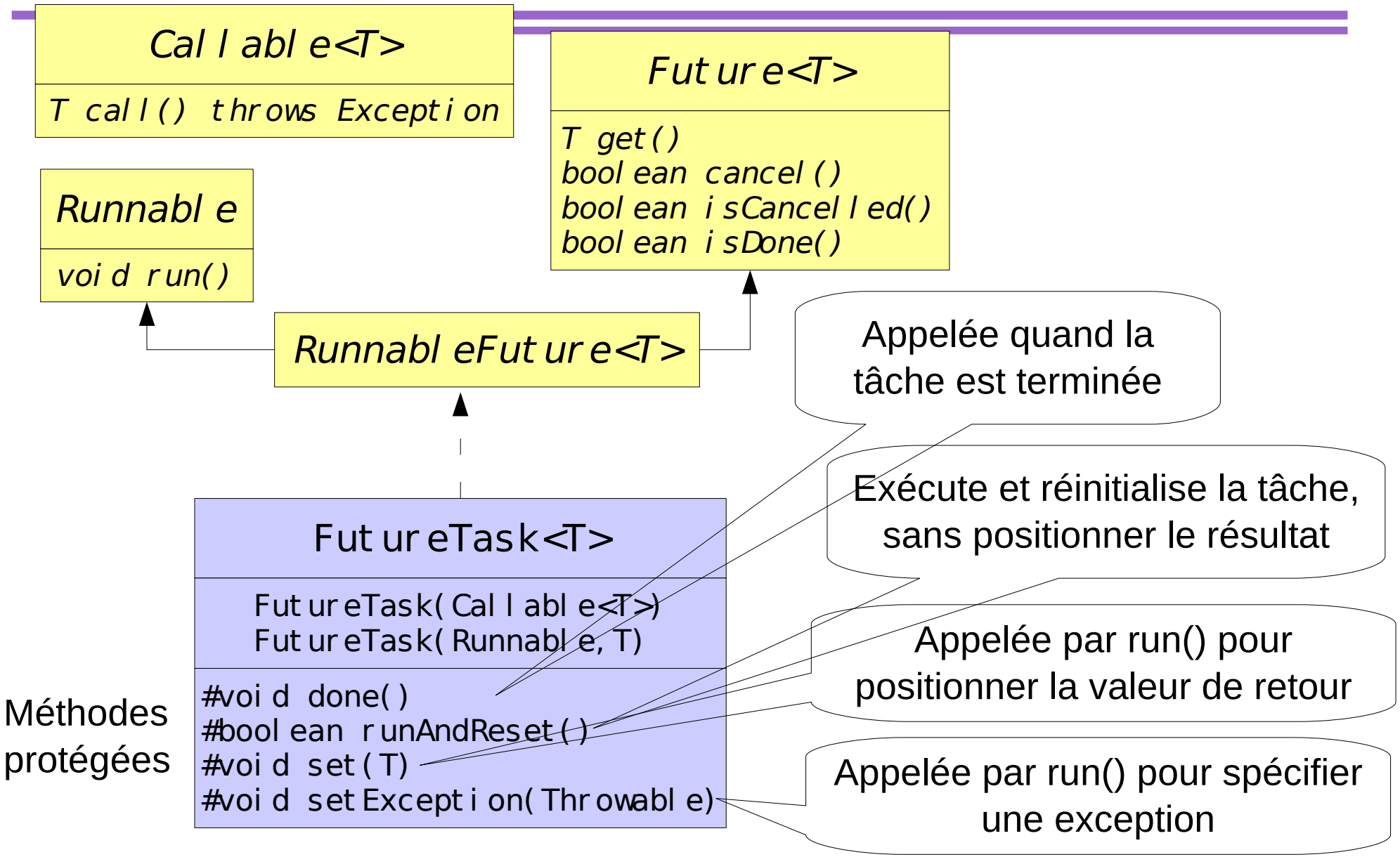
- Temps d'accès à plusieurs sites

```
InetAddress[] hosts=...

ArrayList<Callable<Long>> callables=new ArrayList<Callable<Long>>();
for(int i=0;i<hosts.length;i++) {
  final InetAddress host=hosts[i];
  callables.add(new Callable<Long>() {
    public Long call() throws Exception {
      long time=System.nanoTime();
      host.isReachable(2000);
      return System.nanoTime()-time;
    }
  });
}

ExecutorService executor = Executors.newFixedThreadPool(2);
List<Future<Long>> futures = executor.invokeAll(callables);
executor.shutdown();

for(int i=0;i<hosts.length;i++)
  System.out.println("reach "+hosts[i]+" in "+futures.get(i).get()+" ns");
```

Future et traitement

- Un Future sert aussi à *monitorer* le traitement
  - Savoir si le calcul est terminée (quelle que soit la manière)
    - boolean **isDone**()
  - la tâche a été annulée avant d'avoir terminée
    - boolean **isCancelled**()
  - Arrêter un traitement
    - boolean **cancel**(boolean mayInterruptIfRunning)
      - Retourne false si pas annulable (déjà terminée)
    - Si le traitement n'est pas encore exécuté, il est sortie de la queue des traitements
    - Si il est en cours d'exécution, il peut ou non être interrompu
    - Si il est déjà fini, rien à faire

FutureTask


Callable<T>
T call() throws Exception
Future<T>
T get()
boolean cancel()
boolean isCancelled()
boolean isDone()
Runnable
void run()
RunnableFuture<T>
FutureTask<T>
FutureTask(Callable<T>)
FutureTask(Runnable, T)
#void done()
#boolean runAndReset()
#void set(T)
#void setException(Throwable)
Méthodes protégées
Appelée quand la tâche est terminée
Exécute et réinitialise la tâche, sans positionner le résultat
Appelée par run() pour positionner la valeur de retour
Appelée par run() pour spécifier une exception

CompletionService

- Permet de découpler la planification d'un ensemble de tâches de la manière dont on gère leurs résultats
  - interface CompletionService<T>
  - implementation ExecutorCompletionService<T>
    - Le constructeur accepte un Executor en argument
  - Pour soumettre une tâche:
    `Future<T> submit(Callable<T> task)` ou
    `Future<T> submit(Runnable task, T result)`
  - Pour récupérer (bloquant) et retirer les résultats (dans l'ordre):
    `Future<V> take()`
  - Pour récupérer/retirer (non bloquant; peut retourner null)
    `Future<V> poll()` ou
    `Future<V> poll(long timeout, TimeUnit unit)`

Un exemple de CompletionService

- On affiche les calculs le plus rapidement possible

```
ExecutorService executor = Executors.newFixedThreadPool(3);
CompletionService<Long> completion=
  new ExecutorCompletionService<Long>(executor);
HashMap<Future<?>,InetAddress> map=new HashMap<Future<?>,InetAddress>();
for(int i=0;i<hosts.length;i++) {
  final InetAddress host=hosts[i];
  Future<Long> future = completion.submit(new Callable<Long>() {
    public Long call() throws Exception {
      long time=System.nanoTime();
      host.isReachable(5000);
      return System.nanoTime()-time;
    }
  });
  map.put(future,host);
}
executor.shutdown();

for(int i=0;i<hosts.length;i++) {
  Future<Long> future=completion.take();
  System.out.println("reach "+map.get(future)+" in "+future.get()+" ns");
}
```

ScheduledExecutorService

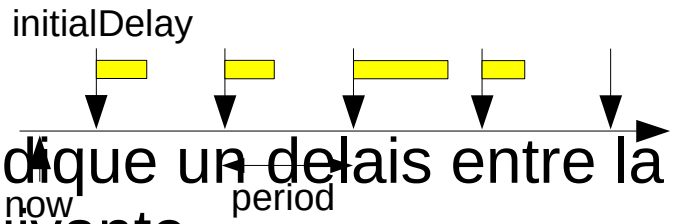
- Ajoute des contraintes temporelles au ExecutorService, marche comme java.util.Timer

- Exécute un traitement après un temps d'attente
  - ScheduledFuture<?> **schedule**(Runnable command,long delay, TimeUnit unit)
  - <V> ScheduledFuture<V> **schedule**(Callable<V> callable, long delay, TimeUnit unit)

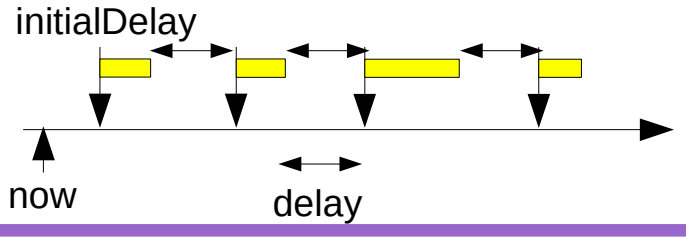
- Permet aussi d'exécuter des traitement de façon periodique

Periode ou Delais

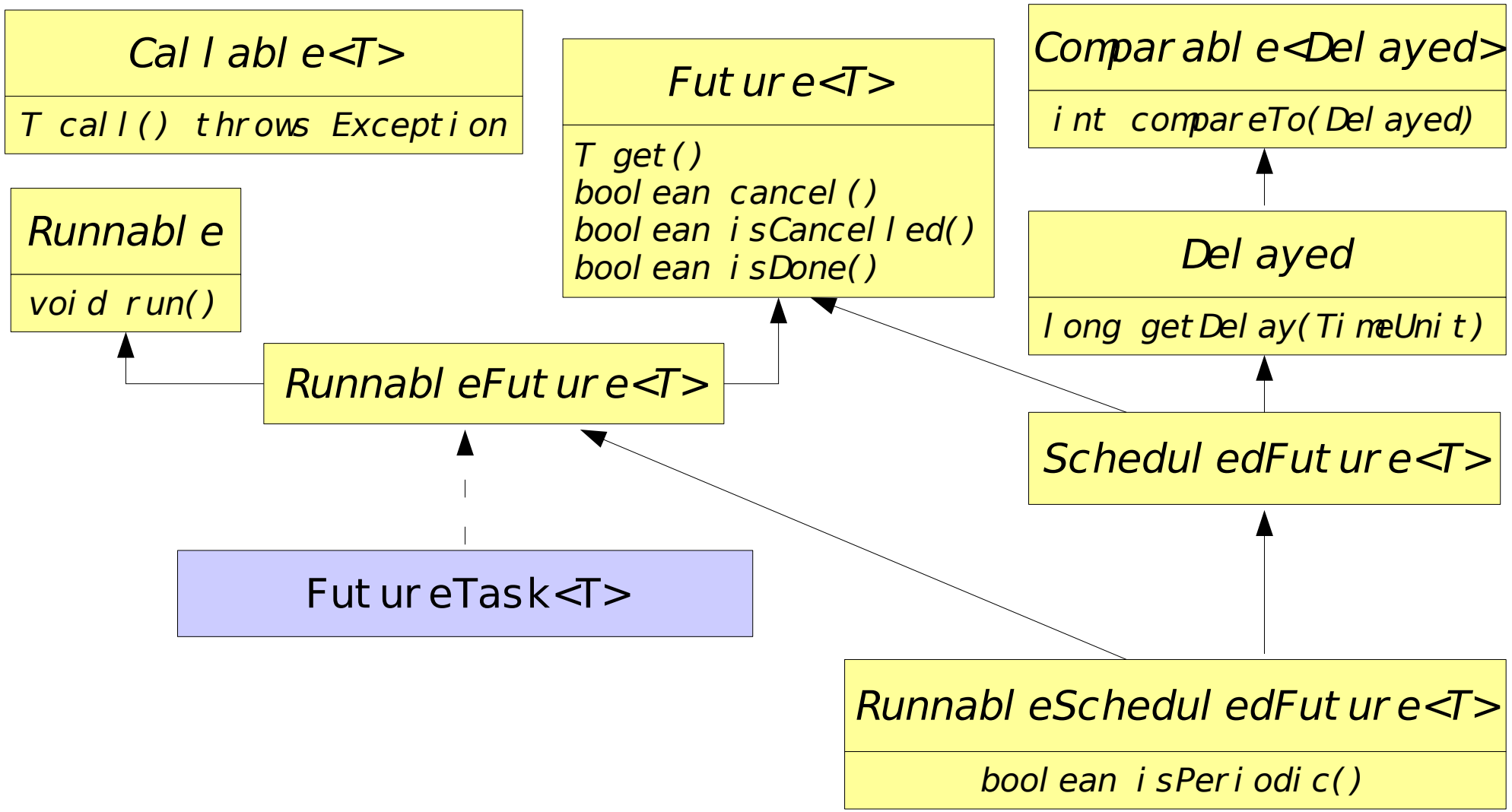
- Deux façons d'indiquer les répétitions
  - Indique la période entre deux exécutions
    - ScheduledFuture<?> **scheduleAtFixedRate**(Runnable command, long initialDelay, long period, TimeUnit unit)



  - Indique un delais entre la fin d'une exécution et l'exécution suivante
    - ScheduledFuture<?> **scheduleWithFixedDelay**(Runnable command,long initialDelay, long delay, TimeUnit unit)



ScheduledFuture



Executors

- ThreadPoolExecutor
  - Normalement configuré par les fabriques de Executors
    - Executors.newCachedThreadPool() pool de thread non borné, avec réclamation automatique des threads inactives
    - Executors.newFixedThreadPool(int) pool de threads de taille fixe, avec file d'attente non bornée
    - Executors.newSingleThreadExecutor() une seule thread qui exécute séquentiellement les tâches soumises
  - Si une thread d'un exécuteur meurt, elle est remplacée
- ScheduledThreadPoolExecutor
  - Rendre non configurable un Executor/ScheduledExecutor:

```
ExecutorService
unconfigurableExecutorService(ExecutorService)
```

Paramètres des pools de threads

- **Nombre de threads** qui sont créées, actives ou attendant une tâche à executer
  - getPoolSize() existantes
  - [get/set]corePoolSize() gardées dans le pool, même si inactives
    - Par défaut, elles ne sont créées qu'à la demande. On peut forcer par prestartCoreThread() ou prestartAllCoreThreads()
  - [get/set]MaximumPoolSize() maximum à ne pas dépasser
- Temps d'inactivité avant terminaison des threads du pool
  - [get/set]KeepAliveTime() si **poolSize>corePoolSize**.
    - Par défaut, Executors fournit des pool avec keepAlive de 60s
- ThreadFactory
  - Spécifiable par le constructeur. Executors.defaultThreadFactory() crée des threads de même groupe, de même priorité normale et de statut non démon

Gestion des tâches

- Lorsque une requête de planification de tâche arrive (execute()) à un Executor, il peut utiliser une file d'attente
  - Si **poolSize < corePoolSize**
    - **Créer une thread** plutôt que de mettre en file d'attente
  - Si **poolSize >= corePoolSize**,
    - **Mettre en file d'attente** plutôt que de créer une thread
  - Si la **requête ne peut pas être mise en file d'attente**
    - Une **nouvelle thread est créée** à moins que **maximumPoolSize** soit atteint
      - dans ce cas, la **requête est rejetée**

Tâches rejetées

- Si on tente de planifier une tâche par execute() dans un Executor
  - qui a déjà été arrêté par shutdown()
  - ou qui utilise une taille maximale finie pour son pool de thread et pour sa file d'attente et qui est "saturé"
- Alors il invoque la méthode
  - rejectedExecution(Runnable tache,ThreadPoolExecutor this) de son RejectedExecutionHandler qui peut être une des quatre classes internes de ThreadPoolExecutor:
    - AbortPolicy: lève l'exception RejectedExecutionException (Runtime)
    - CallerRunsPolicy: la thread appelant execute() se charge de la tâche rejetée
    - DiscardPolicy: la tâche est simplement jetée, oubliée
    - DiscardOldestPolicy: si l'Executor n'est pas arrêté, la tâche en tête de file d'attente de l'Executor est jetée, et une nouvelle tentative de planification est tentée (ce qui peut provoquer à nouveau un rejet...)

Politique de mise en attente

- Passages direct (*direct handoffs*)
  - L'attente de mise dans la file échoue si une thread n'est pas immédiatement disponible: création d'une thread
  - Ex: SynchronousQueue
- Files d'attente non bornées
  - Si les corePoolSize thread de base sont déjà créées et occupées, alors la requête est mise en attente (bien si threads indépendantes)
  - Ex: LinkedBlockingQueue
- Files d'attente bornées
  - Équilibre à trouver entre la taille de la file d'attente et la taille maximale du pool de threads
  - Ex: ArrayBlockingQueue