

Threads

Université de Nice - Sophia Antipolis

Version 3.4 – 23/12/11

Richard Grin

Plan de cette partie

- ❑ Présentation des threads
- ❑ Classe Thread
- ❑ Synchronisation entre threads
- ❑ wait et notify
- ❑ Difficultés du multitâche
- ❑ Divers compléments
- ❑ Nouvelle API pour la concurrence

R. Grin

Java : threads

page 2

- ❑ Ce support expose les concepts de base sur les threads en Java
- ❑ Quelques points de la nouvelle API pour la concurrence sont étudiés à la fin du support

R. Grin

Java : threads

page 3

Définitions

- ❑ Un programme est multitâche quand il lance (ou peut lancer) l'exécution de plusieurs parties de son code en même temps
- ❑ A un moment donné, il comporte plusieurs points d'exécution liés aux différentes parties qui s'exécutent en parallèle

R. Grin

Java : threads

page 4

Systèmes d'exploitation

- ❑ Tous les systèmes d'exploitation modernes sont multitâches et ils permettent l'exécution de programmes multitâches
- ❑ Sur une machine monoprocesseur cette exécution en parallèle est simulée
- ❑ Si le système est préemptif, il peut à tout moment prendre la main à un programme pour la donner à un autre
- ❑ Sinon, un programme garde la main jusqu'à ce qu'il la cède à un autre

R. Grin

Java : threads

page 5

Threads et processus

- ❑ Le multitâche s'appuie sur les processus ou les *threads* (fil d'exécution en français)
- ❑ Chaque processus a son propre espace mémoire (espace où sont rangées les valeurs des variables utilisées par le processus)
- ❑ Un processus peut lancer plusieurs *threads* qui se partagent le même espace mémoire et peuvent donc se partager des variables
- ❑ Un thread prend moins de ressources système qu'un processus

R. Grin

Java : threads

page 6

Exemples de *thread*

- ❑ L'interface graphique avec l'utilisateur lance un *thread* pour charger une image pour continuer à traiter les événements générés par les actions de l'utilisateur
- ❑ Le serveur réseau qui attend les demandes de connexions venant des autres machines lance un *thread* pour traiter chacune des demandes
- ❑ La multiplication de 2 matrices (m, p) et (p, n) peut être effectuée en parallèle par $m \times n$ threads

R. Grin

Java : threads

page 7

Utilité du multitâche

- ❑ Sur une machine multiprocesseurs il permet d'améliorer les performances en répartissant les différentes tâches sur différents processeurs
- ❑ Par exemple, le calcul du produit de 2 matrices peut être réparti en n tâches parallèles (ou k tâches si le nombre k de processeurs est inférieur à n)
- ❑ La répartition des tâches sur les processeurs est le plus souvent faite automatiquement par le système qui offre le multitâche

R. Grin

Java : threads

page 8

Utilité du multitâche (2)

- ❑ Sur une machine monoprocesseur, il peut aussi être intéressant d'utiliser le multitâche pour
 - modéliser plus simplement (simulation par exemple)
 - profiter des temps de pose d'une tâche (attente d'entrées-sorties ou d'une action de l'utilisateur) pour exécuter d'autres tâches
 - réagir plus vite aux actions de l'utilisateur en rejetant une tâche longue et non-interactive dans un autre *thread* (par exemple, chargement d'une image ou lecture de données qui proviennent d'un réseau)

R. Grin

Java : threads

page 9

Problèmes du multitâche

- ❑ Il est souvent plus difficile d'écrire un programme multitâche
- ❑ Et surtout, il est difficile de déboguer un programme qui utilise le multitâche

R. Grin

Java : threads

page 10

Java et le multitâche

- ❑ Java supporte l'utilisation des *threads*
- ❑ A l'inverse de la plupart des autres langages, le programmeur n'a pas à utiliser des bibliothèques natives du système pour écrire des programmes multitâches

R. Grin

Java : threads

page 11

Threads en Java

- ❑ A tout thread Java sont associés
 - un objet qui détermine le code qui est exécuté par le thread
 - un objet qui « contrôle » le thread et le représente auprès des objets de l'application ; on l'appellera le « contrôleur de thread »

R. Grin

Java : threads

page 12

Interface **Runnable**

- La classe de l'objet qui détermine le code à exécuter doit implémenter l'interface **Runnable**

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

méthode qui détermine le code à exécuter par le thread

R. Grin

Java : threads

page 13

Un *thread* n'est pas un objet !

- La méthode **run()** « saute » d'un objet à l'autre en exécutant les méthodes des classes de ces objets :

```
o1.m1();
```

```
o2.m2();
```

```
...
```

- Un *thread* est une unité d'exécution qui, à un moment donné, exécute une méthode
- A un autre moment, ce même *thread* pourra exécuter une autre méthode d'une autre classe

R. Grin

Java : threads

page 14

Contrôleur de thread

- Le contrôleur d'un thread est une instance de la classe **Thread** (ou d'une classe fille) qui
 - est l'intercesseur entre le thread et les objets de l'application
 - permet de contrôler l'exécution du thread (pour le lancer en particulier)
 - a des informations sur l'état du thread (son nom, sa priorité, s'il est en vie ou non,...)

R. Grin

Java : threads

page 15

Classe **Thread**

R. Grin

Java : threads

page 16

Classe **Thread**

- Elle implémente l'interface **Runnable** (mais la méthode **run()** ne fait rien)
- Une instance d'une classe fille de **Thread** peut donc être à la fois un contrôleur de thread et définir le code à exécuter
- Lorsqu'une instance de **Thread** est créée, il faut indiquer le code du thread qui sera contrôlé par cette instance

R. Grin

Java : threads

page 17

2 façons de créer un contrôleur de *thread*

- 1^{ère} façon : utiliser le constructeur **Thread(Runnable)** de la classe **Thread** :
 1. créer un **Runnable** (le code qui sera exécuté par le *thread*)
 2. le passer au constructeur de **Thread**
- 2^{ème} façon : créer une instance d'une classe fille de la classe **Thread** qui redéfinit la méthode **run()**

R. Grin

Java : threads

page 18

Créer un contrôleur de *thread* avec l'interface **Runnable**

```
class Tache implements Runnable {
    . . .
    public void run() {
        // Code qui sera exécuté par le thread
        . . .
    }
}
```

```
Tache tache = new Tache(...);
Thread t = new Thread(tache) ;
```

R. Grin

Java : threads

page 19

Créer un contrôleur de *thread* avec une classe fille de la classe **Thread**

```
class ThreadTache extends Thread {
    . . .
    public void run() {
        // Code qui sera exécuté par le thread
        . . .
    }
}
```

```
ThreadTache threadTache = new ThreadTache(...);
```

R. Grin

Java : threads

page 20

Quelle façon utiliser ?

- ❑ Si on veut hériter d'une autre classe pour la classe qui contient la méthode `run()`, on est obligé de choisir la 1^{ère} façon (`Thread(Runnable)`)
- ❑ Il est aussi plus simple d'utiliser la 1^{ère} façon pour partager des données entre plusieurs threads
- ❑ Sinon, l'écriture du code est (légèrement) plus simple en utilisant la 2^{ème} façon

R. Grin

Java : threads

page 21

Nom d'un thread

- ❑ Des constructeurs de `Thread` permettent de donner un nom au thread en le créant
- ❑ Le nom va faciliter le repérage des threads durant la mise au point

R. Grin

Java : threads

page 22

Lancer l'exécution d'un *thread*

- ❑ On appelle la méthode `start()` du contrôleur de thread :
`t.start();`
- ❑ Le code du `Runnable` s'exécute en parallèle au code qui a lancé le thread
- ❑ Attention, une erreur serait d'appeler directement la méthode `run()` : la méthode `run()` serait exécutée par le thread qui l'a appelée et pas par un nouveau thread

R. Grin

Java : threads

page 23

Relancer l'exécution d'un *thread*

- ❑ On ne peut relancer un thread qui a déjà été lancé
- ❑ Si l'exécution de la méthode `run` du thread n'est pas encore terminée, on obtient une `java.lang.IllegalThreadStateException`
- ❑ Si elle est terminée, aucune exception n'est lancée mais rien n'est exécuté

R. Grin

Java : threads

page 24

Vie du contrôleur de *thread*

- Le contrôleur de thread existe indépendamment du *thread*,
 - avant le démarrage du *thread*, par exemple, pour initialiser des variables d'instances du contrôleur
 - après la fin de l'exécution de ce *thread*, par exemple, pour récupérer des valeurs calculées pendant l'exécution du thread et rangées dans des variables d'instances du contrôleur

R. Grin

Java : threads

page 25

Utilisation d'une classe interne

- La méthode **run** est **public**
- Si on ne souhaite pas qu'elle soit appelée directement, on peut utiliser une classe interne à une classe fille de **Thread** pour implémenter **Runnable**

R. Grin

Java : threads

page 26

Utilisation d'une classe interne anonyme

- Si le code d'une tâche comporte peu de lignes (sinon ça devient vite illisible), on peut lancer son exécution en parallèle en utilisant une classe anonyme :

```
Thread t = new Thread() {
    . . .
    public void run() {
        . . .
    }
};
t.start();
```

Ou encore :

```
new Thread(
    new Runnable() {
        . . .
        public void run() {
            . . .
        }
    }
);
```

R. Grin

Java : threads

page 27

Méthodes publiques principales de la classe **Thread**

```
void start()
static void sleep(long)
    throws InterruptedException
void join() throws InterruptedException
void interrupt()
static boolean interrupted()
int getPriority()
void setPriority(int)
static Thread currentThread()
static void yield()
```

R. Grin

Java : threads

page 28

Thread courant

- La méthode **currentThread** montre bien qu'un thread n'est pas un objet
- Placée dans une méthode de n'importe quelle classe, elle retourne l'objet **Thread** qui contrôle le thread qui exécute cette méthode au moment où **currentThread** est appelé
- On peut ainsi faire un traitement spécial dans le cas où la méthode est exécuté par un certain thread (par exemple le thread de répartition des événements dans un GUI)

R. Grin

Java : threads

page 29

Attente de la fin d'un thread

- Soit un thread **t**

```
t.join();
```

 attend la fin de l'exécution du thread contrôlé par **t**
- On remarquera qu'après la fin de l'exécution du *thread* **t** on peut encore envoyer de messages à l'*objet* contrôleur de thread **t**
- On peut aussi interroger la tâche exécutée par le thread pour récupérer le résultat d'un calcul effectué par le thread

R. Grin

Java : threads

page 30

Passer la main

- La méthode static de la classe **Thread**
`public static void yield()`
 permet de passer la main à un autre thread de priorité égale ou supérieure
- Elle permet d'écrire des programmes plus portables qui s'adaptent mieux aux systèmes multitâches non préemptifs

R. Grin

Java : threads

page 31

Dormir

- La méthode static de la classe **Thread**
`public static void sleep(long millis)`
`throws InterruptedException`
 fait dormir le thread qui l'appelle
- Si elle est exécutée dans du code synchronisé, le thread ne perd pas le moniteur (au contraire de `wait()`)

R. Grin

Java : threads

page 32

Méthode `interrupt()`

- `t.interrupt()`
 demande au thread contrôlé par `t` d'interrompre son exécution
- Cette méthode n'interrompt pas brutalement le thread mais positionne son état « *interrupted* » (l'indicateur d'interruption)

R. Grin

Java : threads

page 33

Méthode `interrupted()`

- La méthode `static interrupted()`
 renvoie la valeur de l'état « *interrupted* » du thread courant
- Attention, après l'exécution de cette méthode, l'état « *interrupted* » du thread est mis à `false`

R. Grin

Java : threads

page 34

Interrompre un thread

- Pour qu'un thread interrompe vraiment son exécution, il doit participer activement à sa propre interruption
- 2 cas :
 - le thread est en attente, par les méthodes `sleep`, `wait`, `join`, ou en attente d'une entrée-sortie
 - toutes les autres situations...

R. Grin

Java : threads

page 35

Interrompre un thread en attente (1)

- Si le thread est en attente avec la méthode `sleep`, ou par `wait` ou `join`, la méthode `interrupt` provoque la levée d'une `java.lang.InterruptedException`
- Le thread « interrompu » gère cette interruption dans un bloc `catch` qui traite cette exception
- Malheureusement, une exception n'est pas levée si le thread est en attente d'une entrée-sortie (sauf avec `InterruptibleChannel` de NIO) ; seul l'indicateur d'interruption est positionné

R. Grin

Java : threads

page 36

Interrompre un thread en attente (2)

- Souvent le thread va repositionner lui-même l'état « *interrupted* » (pour être repéré plus tard par un `Thread.interrupted()` car la levée d'une `InterruptedException` enlève l'état « *interrupted* » du thread :

```
try {
    . . .
    catch(InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

- Voir l'API `java.nio` pour les interruptions des attentes des entrées-sorties

R. Grin

Java : threads

page 37

Interrompre un thread qui n'est pas en attente (1)

- En ce cas le thread doit vérifier périodiquement s'il a été interrompu par la méthode `static Thread.interrupted()`
- Par exemple avec une boucle du type

```
while (! Thread.interrupted()) {
    . . . // faire son travail
}
```

R. Grin

Java : threads

page 38

Interrompre un thread qui n'est pas en attente (2)

- Attention, l'appel de la méthode `interrupted()` annule l'interruption : si cette méthode renvoie `true` pour signaler une interruption et que le thread décide de l'ignorer, un nouvel appel à cette méthode renverra `false` (sauf si le thread est à nouveau interrompu)

R. Grin

Java : threads

page 39

Réaction à une interruption

- Le thread interrompu peut réagir différemment à une interruption
- Il peut interrompre son exécution avec un `return` ou en lançant une exception
- Il peut effectuer un traitement particulier en réaction à l'interruption mais ne pas s'interrompre
- Il peut ignorer l'interruption

R. Grin

Java : threads

page 40

Exemple 1

```
for (int i = 0; i < 100; i++) {
    try {
        Thread.sleep(1000);
    }
    catch (InterruptedException e) {
        // S'interrompt après avoir fait
        // le ménage
        . . .
        return;
    }
    // Effectue un traitement
    traitement();
}
```

R. Grin

Java : threads

page 41

Exemple 2

```
for (int i = 0; i < 100; i++) {
    // Effectue un traitement lourd
    traitement();
    if (Thread.interrupted()) {
        // S'interrompt après avoir fait
        // le ménage
        . . .
        return;
    }
}
```

R. Grin

Java : threads

page 42

Threads et exceptions

- ❑ Si une exception n'est pas traitée (par un bloc `try-catch`), elle interrompt l'exécution du thread courant mais pas des autres threads
- ❑ La méthode `run` ne peut déclarer lancer une exception contrôlée car elle redéfinit une méthode sans clause « `throws` »
- ❑ Une exception non saisie peut être saisie par le groupe du thread (étudié plus loin)

R. Grin

Java : threads

page 43

Synchronisation entre *threads*

R. Grin

Java : threads

page 44

Section critique

- ❑ L'utilisation de *threads* peut entraîner des besoins de synchronisation pour éviter les problèmes liés aux accès simultanés aux variables
- ❑ En programmation, on appelle section critique une partie du code qui ne peut être exécutée en même temps par plusieurs threads sans risquer de provoquer des anomalies de fonctionnement

R. Grin

Java : threads

page 45

Exemple de problème

- ❑ Si $x = 2$, le code `x = x + 1;` exécuté par 2 threads, peut donner en fin d'exécution 3 ou 4 suivant l'ordre d'exécution :
 1. T1 : lit la valeur de x (2)
 2. T2 : lit la valeur de x (2)
 3. T1 : calcul de x + 1 (3)
 4. T2 : calcul de x + 1 (3)
 5. T1 : range la valeur calculée dans x (3)
 6. T2 : range la valeur calculée dans x (3)
- ❑ x contient 3 au lieu de 4 !

R. Grin

Java : threads

page 46

Nécessaire synchronisation

- ❑ Il faut donc éviter l'exécution simultanée de sections de code critiques par plusieurs threads
- ❑ En Java le code synchronisé sur un objet est utilisé pour synchroniser les threads et les empêcher d'exécuter en même temps des portions de code
- ❑ Plusieurs threads ne peuvent exécuter en même temps du code synchronisé sur un même objet

R. Grin

Java : threads

page 47

2 possibilités pour synchroniser du code sur un objet `o`

- ❑ Méthode synchronisée `m` (avec un message envoyé à l'objet `o` : `o.m(...)`) :

```
public synchronized int m(...) { . . . }
```

- ❑ Bloc synchronisé sur l'objet `o` :

```
synchronized(o) {
    // le code synchronisé
    . . .
}
```

R. Grin

Java : threads

page 48

Moniteur d'un objet

- ❑ La protection du code synchronisé contre les accès multiples repose sur les moniteurs des objets
- ❑ Chaque objet Java possède un moniteur
- ❑ À un moment donné, un seul thread peut posséder le moniteur d'un objet
- ❑ Si d'autres threads veulent acquérir le même moniteur, ils sont mis en attente, en attendant que le premier thread rende le moniteur

R. Grin

Java : threads

page 49

Acquisition et restitution d'un moniteur

- ❑ Un *thread* t acquiert automatiquement le moniteur d'un objet o en exécutant du code synchronisé sur cet objet o
- ❑ t rend le moniteur en quittant le code synchronisé (ou se met en attente en appelant `o.wait()` dans le code synchronisé)
- ❑ Il peut quitter le code synchronisé normalement, ou si une exception est lancée et non saisie dans le code synchronisé

R. Grin

Java : threads

page 50

Résumé

- ❑ Tant que t exécute du code synchronisé sur un objet o , les autres threads ne peuvent exécuter du code synchronisé sur ce même objet o (le même code, ou n'importe quel autre code synchronisé sur o) ; ils sont mis en attente
- ❑ Lorsque t quitte le code synchronisé ou se met en attente par `o.wait()`, un des threads en attente peut commencer à exécuter le code synchronisé
- ❑ Les autres threads en attente auront la main à tour de rôle (si tout se passe bien...)

R. Grin

Java : threads

page 51

Exemple

```
public class Compte {
    private double solde;

    public void deposer(double somme) {
        solde = solde + somme;
    }

    public double getSolde() {
        return solde;
    }
}
```

R. Grin

Java : threads

page 52

Exemple (suite)

- ❑ On lance 3 threads du type suivant :

```
Thread t1 = new Thread() {
    public void run() {
        for (int i = 0; i < 100; i++) {
            compte.deposer(1000);
        }
    }
};
```

- ❑ A la fin de l'exécution, on n'obtient pas nécessairement 300.000

R. Grin

Java : threads

page 53

Exemple (fin)

- ❑ Pour éviter ce problème il faut rendre `deposer` synchronisée :

```
public synchronized
void deposer(double somme)
```

- ❑ En fait, pour sécuriser la classe contre les accès multiples, il faudra aussi rendre `getSolde` synchronisé car, en Java, un `double` n'est pas lu en une opération atomique. Il faut donc éviter que `getSolde` ne soit exécuté en même temps que `deposer`

R. Grin

Java : threads

page 54

Favoriser la détection des problèmes

- ❑ En fait, si on exécute le code précédent sans rendre `deposer` synchronized, on obtiendra bien souvent (mais pas toujours...) le bon résultat
- ❑ Ça dépend du fonctionnement du multitâche du système d'exploitation sous-jacent, et de la JVM
- ❑ Pour repérer plus facilement les problèmes de multitâche, on peut ajouter des appels aux méthodes `yield` ou `sleep` qui forcent le thread à rendre la main, et permettre ainsi à un autre thread de pouvoir s'exécuter

R. Grin

Java : threads

page 55

Provoquer le problème

```
public class Compte {
    private double solde;

    public void deposer(double somme) {
        double soldeTemp = solde;
        Thread.yield();
        solde = soldeTemp + somme;
    }

    public double getSolde() {
        return solde;
    }
}
```

Avec tous les SE et JVM, la main pourra être rendue pendant l'exécution de cette méthode

R. Grin

Java : threads

page 56

Synchronisation et performances

- ❑ L'exécution de code synchronisé peut nuire aux performances
- ❑ Il peut provoquer des blocages entre threads ; peu de perte de performance s'il n'y a pas de blocage
- ❑ Du code synchronisé peut empêcher des optimisations (*inlining*) au moment de la compilation

R. Grin

Java : threads

page 57

Limiter la portée du code synchronisé

- ❑ Il faut synchroniser le moins de code possible pour faciliter les accès multiples (les performances seront meilleures s'il y a moins de threads en attente d'un moniteur)
- ❑ Attention, éviter d'appeler à l'intérieur d'une portion synchronisée des méthodes qu'un client peut redéfinir (dans une classe fille) ; en effet, une redéfinition non appropriée peut provoquer des pertes de performance ou même des inter-blocages

R. Grin

Java : threads

page 58

Méthodes de classe synchronisées

- ❑ Si on synchronise une méthode `static`, on bloque le moniteur de la classe
- ❑ On bloque ainsi tous les appels à des méthodes synchronisées de la classe (mais pas les appels synchronisés sur une instance de la classe)

R. Grin

Java : threads

page 59

Méthode synchronisée et héritage

- ❑ La redéfinition d'une méthode synchronisée dans une classe fille peut ne pas être synchronisée
- ❑ De même, la redéfinition d'une méthode non synchronisée peut être synchronisée

R. Grin

Java : threads

page 60

Synchronisation et visibilité de modifications

- La synchronisation permet d'assurer la visibilité par les autres threads de la modification d'une variable par un thread (voir « happens-before » dans la section « difficultés du multi-tâche » dans la suite de ce cours

R. Grin

Java : threads

page 61

wait et notify

R. Grin

Java : threads

page 62

Exécution conditionnelle

- Lorsqu'un programme est multi-tâche, la situation suivante peut se rencontrer :
 - Un thread t1 ne peut continuer son exécution que si une condition est remplie
 - Le fait que la condition soit remplie ou non dépend d'un autre thread t2
- Par exemple, t1 a besoin du résultat d'un calcul effectué par t2

R. Grin

Java : threads

page 63

Exécution conditionnelle (2)

- Une solution coûteuse serait que t1 teste la condition à intervalles réguliers
- Les méthodes `wait()` et `notify()` de la classe `Object` permettent de programmer plus efficacement ce genre de situation

R. Grin

Java : threads

page 64

Schéma d'utilisation de wait-notify

- Cette utilisation demande un travail coopératif entre les threads t1 et t2 :
 1. Ils se mettent d'accord sur un objet commun `objet`
 2. Arrivé à l'endroit où il ne peut continuer que si la condition est remplie, t1 se met en attente : `objet.wait();`
 3. Quand t2 a effectué le travail pour que la condition soit remplie, il le notifie : `objet.notify();` ce qui débloque t1

R. Grin

Java : threads

page 65

Besoin de synchronisation

- Le mécanisme d'attente-notification lié à un objet met en jeu l'état interne de l'objet ; pour éviter des accès concurrent à cet état interne, une synchronisation est nécessaire
- Les appels `objet.wait()` et `objet.notify()` (et `objet.notifyAll()`) ne peuvent donc être effectués que dans du code synchronisé sur `objet`
- Autrement dit, `wait` et `notify` ne peuvent être lancés que dans une section critique

R. Grin

Java : threads

page 66

Méthode `wait()`

- `public final void wait()`
`throws InterruptedException`
- `objet.wait()`
 - nécessite que le *thread* en cours possède le moniteur de *objet*
 - bloque le *thread* qui l'appelle, jusqu'à ce qu'un autre *thread* appelle la méthode `objet.notify()` ou `objet.notifyAll()`
 - libère le moniteur de l'objet (l'opération « blocage du thread – libération du moniteur » est atomique)

R. Grin

Java : threads

page 67

Utilisation de `wait`

- Mauvaise utilisation :
~~`if (!condition) objet.wait();`~~
 - si on quitte l'attente avec le `wait()`, cela signifie qu'un autre thread a notifié que la condition était remplie
 - mais, après la notification, et avant le redémarrage de ce thread, un autre thread a pu prendre la main et modifier la condition
- Le bon code (dans du code synchronisé sur le même objet que le code qui modifie la condition) :

```
while (!condition) {
    objet.wait();
}
```

R. Grin

Java : threads

page 68

Méthode `notifyAll()`

- `public final void notifyAll()`
- `objet.notifyAll()`
 - nécessite que le *thread* en cours possède le moniteur de *objet*
 - débloquent *tous* les threads qui s'étaient bloqués sur l'objet avec `objet.wait()`

R. Grin

Java : threads

page 69

Méthode `notifyAll()`

- Un seul des threads débloqués va récupérer le moniteur ; on ne peut prévoir lequel
- Les autres devront attendre qu'il relâche le moniteur pour être débloqués à tour de rôle, mais ils ne sont plus bloqués par un `wait`
- En fait, souvent ils se bloqueront à nouveau eux-mêmes (s'ils sont dans une boucle `while` avec `wait`)

R. Grin

Java : threads

page 70

Méthode `notify()`

- `objet.notify()`
 - idem `notifyAll()` mais
 - ne débloquent qu'un seul thread
- On ne peut prévoir quel sera le thread débloqué et, le plus souvent, il vaut donc mieux utiliser `notifyAll()`

R. Grin

Java : threads

page 71

Déblocage des threads

- Le thread débloqué (et élu) ne pourra reprendre son exécution que lorsque le thread qui l'a notifié rendra le moniteur de l'objet en quittant sa portion de code synchronisé
- Le redémarrage et l'acquisition se fait dans une opération atomique

R. Grin

Java : threads

page 72

notify perdus

- Si un `notifyAll()` (ou `notify()`) est exécuté alors qu'aucun thread n'est en attente, il est perdu : il ne débloquera pas les `wait()` exécutés ensuite

R. Grin

Java : threads

page 73

Exemple avec wait-notify

- Les instances d'une classe `Depot` contiennent des jetons
- Ces jetons sont
 - déposés par un producteur
 - consommés par des consommateurs
- Les producteurs et consommateurs sont des *threads*

R. Grin

Java : threads

page 74

Exemple avec wait-notify

```
public class Depot {
    private int nbJetons = 0;

    public synchronized void donneJeton() {
        try {
            while (nbJetons == 0) {
                wait();
            }
            nbJetons--;
        }
        catch (InterruptedException e) {}
    }
}
```

R. Grin

Java : threads

page 75

Exemple avec wait-notify

```
public synchronized void recois(int n) {
    nbJetons += n;
    notifyAll();
}
```

R. Grin

Java : threads

page 76

Variante de wait

- Si on ne veut pas attendre éternellement une notification, on peut utiliser une des variantes suivantes de `wait` :


```
public void wait(long timeout)
public void wait(long timeout,
                 int nanos)
```
- Dans ce cas, le thread doit gérer lui-même le fait de connaître la cause de son déblocage (`notify` ou temps écoulé)

R. Grin

Java : threads

page 77

Moniteurs réentrants

- Un *thread* qui a acquis le moniteur d'un objet peut exécuter les autres méthodes synchronisées de cet objet ; il n'est pas bloqué en demandant à nouveau le moniteur

R. Grin

Java : threads

page 78

Affectations atomiques

- ❑ Il est inutile de synchroniser une partie de code qui ne fait qu'affecter (ou lire) une valeur à une variable de type primitif de longueur 32 bits ou moins (`int`, `short`, ...)
- ❑ En effet, la spécification du langage Java spécifie qu'une telle affectation ne peut être interrompue pour donner la main à un autre *thread*
- ❑ Mais, cette spécification n'assure rien pour les affectations de `double` et de `long` !

R. Grin

Java : threads

page 79

Stopper un thread

- ❑ Si on stoppe un thread il arrête de s'exécuter
- ❑ Il ne peut reprendre son exécution ; si on veut pouvoir suspendre et reprendre l'exécution du thread, voir `suspend` et `resume`
- ❑ `stop()` est *deprecated* car le thread stoppé peut laisser des objets dans un état inconsistant car il relâche leur moniteur quand il arrête son exécution
- ❑ On peut simuler `stop` en utilisant une variable

R. Grin

Java : threads

page 80

Simuler `stop()`

- ❑ Pour rendre possible l'arrêt d'un *thread* T, on peut utiliser une variable `arretThread` visible depuis T et les threads qui peuvent stopper T :
 - T initialise `arretThread` à `false` lorsqu'il démarre
 - pour stopper T, un autre thread met `arretThread` à `true`
 - T inspecte à intervalles réguliers la valeur de `arretThread` et s'arrête quand `arretThread` a la valeur `true`

R. Grin

Java : threads

page 81

Simuler `stop()`

- ❑ `arretThread` doit être déclarée `volatile` si elle n'est pas accédée dans du code synchronisé (`volatile` est étudié plus loin dans ce cours)

R. Grin

Java : threads

page 82

Interrompre un thread

- ❑ Le mécanisme décrit précédemment pour stopper un thread ne fonctionne pas si le thread est en attente (`wait`, `sleep`, attente d'une entrée-sortie avec le paquetage `java.nio` depuis le SDK 1.4)
- ❑ Dans ce cas, on utilise `interrupt()` qui interrompt les attentes et met l'état du thread à « *interrupted* » (voir la section sur la classe `Thread` du début de ce cours)

R. Grin

Java : threads

page 83

Suspendre et relancer un thread

- ❑ `suspend()` et `resume()` sont *deprecated* car ils favorisent les interblocages (un thread suspendu ne relâche pas les moniteurs qu'il possède)
- ❑ On peut les remplacer en utilisant une variable `suspendreThread` comme pour la méthode `stop()`
- ❑ Comme il faut pouvoir reprendre l'exécution on doit en plus utiliser `wait()` et `notify()`

R. Grin

Java : threads

page 84

Simuler **suspend** et **resume**

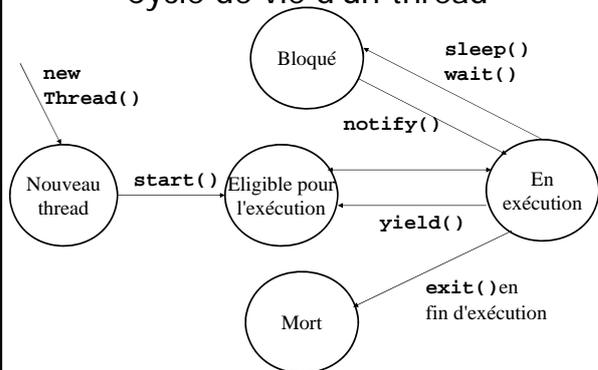
- ❑ Pendant son exécution le thread scrute la valeur de `suspendreThread`
- ❑ Si la valeur est `true`, le thread se met en attente avec `wait` sur un objet `o`
- ❑ Quand un autre thread veut relancer l'exécution du thread, il met la variable `suspendreThread` à `false` et il appelle `notify` sur l'objet `o`
- ❑ Rappel : les appels de `wait` et `notify` doivent se faire dans des sections synchronisées sur `o`

R. Grin

Java : threads

page 85

Cycle de vie d'un thread



R. Grin

Java : threads

page 86

Threads démons

- ❑ 2 types de threads
 - les threads utilisateur
 - les démons
- ❑ La différence :
 - la JVM fonctionne tant qu'il reste des threads utilisateurs en exécution
 - la JVM s'arrête s'il ne reste plus que des démons
- ❑ Les démons sont là seulement pour rendre service aux threads utilisateur. Exemple : ramasse-miettes

R. Grin

Java : threads

page 87

Threads démons

- ❑ La méthode `void setDaemon(boolean on)` de la classe `Thread` permet d'indiquer que le thread sera un démon (thread utilisateur par défaut)
- ❑ Elle doit être appelée avant le démarrage du thread par l'appel de `start()`

R. Grin

Java : threads

page 88

Difficultés liées au multitâche

R. Grin

Java : threads

page 89

Difficultés du multitâche

- ❑ Si un thread `t` doit attendre 2 notify de 2 autres threads `t1` et `t2`, ce serait une faute de coder

```
wait();
wait();
```

R. Grin

Java : threads

page 90

Difficultés du multitâche

- En effet, les 2 `notify()` peuvent arriver « presque en même temps » :
 - le thread `t1` envoie un 1^{er} `notify()` qui débloque le 1^{er} `wait()` ; il relâche ensuite le moniteur et permet ainsi à l'autre thread `t2`...
 - ... d'envoyer le 2^{ème} `notify()` avant que le 2^{ème} `wait()` ne soit lancé
 - le thread `t` reste bloqué éternellement sur le 2^{ème} `wait()` (qui ne recevra jamais de `notify()`)

R. Grin

Java : threads

page 91

Comment coder cette situation ?

- On compte le nombre de `notify()` avec une variable qui est incrémentée dans une partie critique (synchronisée) qui contient le `notify()` (pour être certain que la variable représente vraiment le nombre de `notify()`) :

```
nbNotify++;
notifyAll();
```

R. Grin

Java : threads

page 92

Comment coder cette situation (suite)

- Et on se met en attente dans une boucle (dans une portion synchronisée) :

```
while (nbNotify < 2) {
    wait();
}
```

- Si on reçoit 1 `notify()` entre les 2 `wait()`, `nbNotify` sera égal à 2 et on sortira de la boucle sans faire le 2^{ème} `wait()`

R. Grin

Java : threads

page 93

Éviter la synchronisation

- Comme la synchronisation a un coût non négligeable, il faut essayer de l'éviter quand c'est possible
- Par exemple, si un seul thread écrit une valeur de type `int` qui est lue par plusieurs autres threads, on peut se passer de synchronisation car les opérations de lecture-écriture de `int` sont atomiques
- Mais attention, il y a de nombreux pièges !

R. Grin

Java : threads

page 94

Partage de variables par les threads

- Soit `v` une variable partagée par plusieurs threads
- Si le thread `T` modifie la valeur de `v`, cette modification peut ne pas être connue immédiatement par les autres threads
- Par exemple, le compilateur a pu utiliser un registre pour conserver la valeur de `v` pour `T`
- La spécification de Java n'impose la connaissance de cette modification par les autres threads que lors de l'acquisition ou le relâchement du moniteur d'un objet (`synchronized`)

R. Grin

Java : threads

page 95

Volatile

- Pour éviter ce problème, on peut déclarer la variable `v` `volatile`
- On est ainsi certain que tous les threads partageront une zone mémoire commune pour ranger la valeur de la variable `v`
- De plus, si une variable de type `long` et `double` est déclarée `volatile`, sa lecture et son écriture est garantie atomique

R. Grin

Java : threads

page 96

Attention !

- ❑ Dans les anciennes versions de Java, une variable volatile `init` ne peut servir à indiquer l'initialisation de variables non volatiles
- ❑ Par exemple, dans le code suivant :


```

v1 = 8;
v2 = 3;
init = true;

```

 même si `init` est lue comme vraie par un autre thread, `v1` et `v2` n'ont peut-être pas été initialisées car le compilateur pouvait intervertir les instructions s'il le juge utile pour l'optimisation

R. Grin

Java : threads

page 97

« *Happens-before* »

- ❑ Les nouvelles versions de Java (à partir de la version 5) formalisent les cas où une modification effectuée par un thread est nécessairement connue par un autre thread (ce qui signifie que, dans tous les autres cas, cette modification n'est *peut-être* par visible par l'autre thread) avec la notion de « *happens-before* » (arrive-avant) définie dans le chapitre 17 de la spécification du langage

R. Grin

Java : threads

page 98

Définition de « *happens-before* »

- ❑ Une modification effectuée par un thread T1 est assurée d'être visible par un thread T2 si et seulement si cette modification *happens-before* la lecture par le thread T2
- ❑ Cette notion de *happens-before* n'est pas nécessairement liée au temps ; il faut des conditions bien précises pour qu'elle ait lieu
- ❑ Le transparent suivant résume les cas de « bas niveau » ; l'API `java.util.concurrent` ajoute d'autres cas de plus haut niveau

R. Grin

Java : threads

page 99

Cas de « *happens-before* »

- ❑ Dans *un seul thread*, une action *happens-before* toute action qui arrive après dans le code
- ❑ La libération d'un moniteur *happens-before* l'acquisition future de ce moniteur
- ❑ L'écriture dans un champ `volatile` *happens-before* la lecture future de la valeur de ce champ
- ❑ `t1.start()` *happens-before* toutes les actions effectuées par le thread t1
- ❑ Les actions d'un thread t1 *happens-before* les actions d'un thread t2 qui suivent l'instruction `t1.join()`

R. Grin

Java : threads

page 100

Transitivité de « *Happens-before* »

- ❑ Si une action A *happens-before* une action B et si B *happens-before* une action C, alors A *happens-before* C
- ❑ Ce qui signifie, par exemple, que toute action effectuée par un thread t1 *happens-before* toute action effectuée par un thread t2, qui suit l'exécution de `t1.join()`
- ❑ Ce qui signifie aussi que le transparent précédent « Attention ! » sur l'utilisation erronée d'une variable volatile n'est plus d'actualité

R. Grin

Java : threads

page 101

Priorités

R. Grin

Java : threads

page 102

Principe de base

- ❑ Si plusieurs threads de même priorité sont en exécution, on ne peut pas prévoir quel thread va prendre la main
- ❑ S'ils sont en attente d'exécution, un thread de plus grande priorité prendra toujours la main avant un autre thread de priorité plus basse
- ❑ Cependant, il peut arriver exceptionnellement qu'un thread continue son exécution alors que des threads de priorité supérieure sont en attente d'exécution

R. Grin

Java : threads

page 103

Niveaux de priorité

- ❑ Un nouveau thread a la même priorité que le thread qui l'a créé
- ❑ En général, tous les threads ont la même priorité (**NORM_PRIORITY**)
- ❑ Il faut faire appel à la méthode **setPriority** si on veut modifier cette priorité par défaut
- ❑ Le paramètre de **setPriority** doit être inclus entre **MIN_PRIORITY** et **MAX_PRIORITY**

R. Grin

Java : threads

page 104

Autres classes liées aux threads : **ThreadGroup** et **ThreadLocal**

R. Grin

Java : threads

page 105

ThreadGroup

- ❑ **ThreadGroup** représente un ensemble de *threads*, qui peut lui-même comprendre un *threadGroup* ; on a ainsi une arborescence de *threadGroup*
- ❑ On peut ainsi jouer en une seule instruction sur la priorité des *threads* du groupe ou sur le fait que les *thread* soient des démons ou non
- ❑ Cette classe n'est pas très utile et il est conseillé de ne pas l'utiliser ; on se limite ici à l'essentiel

R. Grin

Java : threads

page 106

Lancer un thread d'un groupe

- ❑ Pour créer un thread d'un groupe, on doit utiliser le constructeur de la classe **Thread** qui prend un groupe en paramètre ; par exemple :

```
ThreadGroup tg =
    new MonGroupe("monGroupe");
Thread t = new MonThread(tg, "monThread");
t.start();
```

R. Grin

Java : threads

page 107

ThreadGroup et exceptions

- ❑ La classe **ThreadGroup** contient la méthode **uncaughtException(Thread t, Throwable e)** qui est exécutée quand un des threads du groupe est stoppé par une exception non saisie
- ❑ On peut redéfinir cette méthode pour faire un traitement spécial sur les autres threads
- ❑ Depuis le JDK 5, il est plus simple d'utiliser la méthode **static setDefaultUncaughtException** ou d'instance **setUncaughtException** de la classe **Thread** pour traiter d'une façon particulière les exceptions non attrapées

R. Grin

Java : threads

page 108

ThreadGroup et exceptions

```
class MyThreadGroup extends ThreadGroup {
    public MyThreadGroup(String s) {
        super(s);
    }

    public void uncaughtException(Thread t, Throwable e) {
        // On met ici le traitement qui doit être exécuté
        // si un des threads du groupe reçoit une exception
        // non attrapée
        System.err.println("uncaught exception: " + e);
    }
}
```

R. Grin

Java : threads

page 109

ThreadLocal

- ❑ Cette classe de `java.lang` permet d'associer un état (typiquement une variable `static private`) à chaque *thread* sans créer d'instances différentes
- ❑ Puisque la valeur n'est connue que d'un seul thread on évite ainsi les problèmes liés aux accès concurrents
- ❑ C'est aussi un moyen commode d'utiliser une valeur tout au long du déroulement d'un thread sans utiliser une variable connue par toutes les lignes de code concernées

R. Grin

Java : threads

page 110

Classe ThreadLocal<T> Méthodes de base

- ❑ `public void set(T valeur)`
met une valeur de type `T` dans l'objet de type `ThreadLocal`
- ❑ `public T get()`
récupère la valeur rangé dans l'objet de type `ThreadLocal`
- ❑ `public void remove()`
enlève la valeur de l'objet

R. Grin

Java : threads

page 111

Classe ThreadLocal<T> Initialisation de la valeur

- ❑ `protected T initialValue()`
renvoie la valeur renvoyée par `get` si aucun `set` n'a été appelé avant (renvoie `null` dans la classe `ThreadLocal`)
- ❑ Pour donner une valeur initiale à la variable locale, il suffit de redéfinir cette méthode dans une sous-classe de `ThreadLocal`

R. Grin

Java : threads

page 112

Exemple

```
public class C {
    private static final ThreadLocal
        tlSession = new ThreadLocal<Session>();
    . . .
    public static getSession() {
        Session s = tlSession.get();
        if (s == null) {
            s = getFabrique().newSession();
            tlSession.set(s);
        }
    }
}
```

En appelant `C.getSession()`, tout le code parcouru par un thread pourra travailler avec la même session

R. Grin

Java : threads

page 113

InheritableThreadLocal

- ❑ Classe fille de `ThreadLocal`
- ❑ La différence avec `ThreadLocal` est que la valeur de ces variables (si elle a une valeur) est passée aux threads créés par le thread courant
- ❑ La méthode `protected T childValue(T valeurDuPère)` peut être redéfinie dans une classe fille pour donner au thread fils une valeur calculée à partir de la valeur du thread père (par défaut, cette méthode renvoie la valeur passée en paramètre)

R. Grin

Java : threads

page 114

Timers

R. Grin

Java : threads

page 115

Classes `Timer` et `TimerTask`

- Ces 2 classes du paquetage `java.util` permettent de lancer l'exécution de tâches à des intervalles donnés
- `TimerTask` a une méthode `run()` qui détermine la tâche à accomplir
- `Timer` détermine quand seront exécutées les tâches qu'on lui associe
- Dans les 2 classes des méthodes `cancel()` permettent d'interrompre une tâche ou toutes les tâches associées à un timer

R. Grin

Java : threads

page 116

Configuration du timer

- Un timer peut déclencher une seule exécution, ou pour déclencher des exécutions à des intervalles réguliers
- Pour l'exécution à des intervalles réguliers, le timer peut être configuré pour qu'il se cale le mieux par rapport au début des exécutions (méthode `scheduleAtFixedRate`), ou par rapport à la dernière exécution (méthode `schedule`)

R. Grin

Java : threads

page 117

Exemple d'utilisation de timer

```
final long debut =
    System.currentTimeMillis();
TimerTask afficheTemps =
    new TimerTask() {
        public void run() {
            System.out.println(
                System.currentTimeMillis()- debut);
        }
    };
Timer timer = new Timer();
Timer.schedule(afficheTemps, 0, 2000);
```

Démarrer
tout de suiteIntervalles de 2 s
entre 2 affichages

R. Grin

Java : threads

page 118

Timers et swing

- Pour utiliser un timer qui modifie l'affichage en Swing, il faut utiliser la classe `javax.swing.Timer`
- Cette classe utilise le thread de distribution des événements pour faire exécuter les tâches

R. Grin

Java : threads

page 119

Utiliser des classes non sûres
vis-à-vis des threads
(pas « *thread-safe* »)

R. Grin

Java : threads

page 120

Utiliser des classes non sûres

- ❑ Si c'est possible, synchroniser explicitement les accès aux objets partagés en construisant des classes qui enveloppent les classes non sûres, avec des méthodes synchronisées (illustré par les collections)
- ❑ Sinon, synchroniser les accès au niveau des clients de ces classes ; c'est plus difficile et moins pratique
- ❑ On peut aussi s'arranger pour que les méthodes non sûres ne soient appelées que par un seul thread (illustré par swing et le thread de distribution des événements)

R. Grin

Java : threads

page 121

Exemple des collections

- ❑ Les nouvelles collections (Java 2) ne sont pas sûres vis-à-vis des threads
- ❑ Ça permet
 - d'améliorer les performances en environnement mono-thread
 - davantage de souplesse en environnement multi-threads : par exemple, pour ajouter plusieurs objets, on peut n'acquérir qu'une seule fois un moniteur

R. Grin

Java : threads

page 122

Collections synchronisées

- ❑ L'API des collections permet d'obtenir une collection synchronisée à partir d'une collection non synchronisée, par exemple avec la méthode `static Collections.synchronizedList`

R. Grin

Java : threads

page 123

Protection des collections non synchronisées

- ❑ Il faut synchroniser explicitement les modifications des collections :

```
private ArrayList al;
. . .
Synchronized(al) {
    al.add(...);
    al.add(...);
}
```

Avantage sur **Vector** : une seule acquisition de moniteur pour plusieurs modifications

R. Grin

Java : threads

page 124

Protection des collections non synchronisées

- ❑ Cette technique n'est pas toujours possible si les classes non synchronisées font appel elles-mêmes à des objets non-protégés auxquelles on ne peut accéder

R. Grin

Java : threads

page 125

Collections *thread-safe*

- ❑ Le paquetage `java.util.concurrent` du JDK 5 (voir section suivante) contient des collections « *thread-safe* » qui offrent sécurité et performance en environnement multi-tâche : **ConcurrentHashMap**, **CopyOnWriteArrayList**, **CopyOnWriteArraySet**
- ❑ Il ne faut les utiliser que lorsqu'il y a des risques dus à des modifications par plusieurs threads (moins performantes que les collections « normales »)

R. Grin

Java : threads

page 126

Paquetage `java.util.concurrent`

R. Grin

Java : threads

page 127

Nouvelle API

- ❑ Le JDK 1.5 a ajouté un nouveau paquetage qui offre de nombreuses possibilités, avec de bonnes performances
- ❑ Le programmeur n'aura ainsi pas à réinventer la roue pour des fonctionnalités standards telles que les exécutions asynchrones, les gestions de collections accédées par plusieurs threads (telles que les files d'attente), les blocages en lectures/écritures, etc.

R. Grin

Java : threads

page 128

- ❑ Ce cours ne fait que survoler quelques possibilités offertes par cette API
- ❑ Pour plus de précisions se reporter à la javadoc de l'API et à ses tutoriels

R. Grin

Java : threads

page 129

Problèmes avec **Runnable**

- ❑ La méthode `run` ne peut renvoyer une valeur et elle ne peut lancer d'exceptions contrôlées par le compilateur
- ❑ Les valeurs calculées par la méthode `run` doivent donc être récupérées par une méthode de type `getValeur()` et les exceptions contrôlées doivent être attrapées et signalées d'une façon détournée au code qui a lancé `run`

R. Grin

Java : threads

page 130

Nouveau cadre

- ❑ La nouvelle API fournit un nouveau cadre pour faciliter et enrichir les possibilités lors du lancement de tâches en parallèle
- ❑ L'interface `Callable` améliore `Runnable`
- ❑ `Future` facilite la récupération des valeurs calculées en parallèle
- ❑ `Executor` et `Executors` découplent la soumission de tâches et leur exécution, et offrent une gestion de pools de threads

R. Grin

Java : threads

page 131

Considérations techniques

- ❑ De nouvelles instructions ont été ajoutées aux processeurs pour faciliter leur utilisation en environnement multi-cœurs
- ❑ Par exemple, dans les processeurs Intel, l'instruction « *compare-and-swap* » (CAS), **en une seule opération atomique**, (la main ne peut être donnée à un autre thread pendant son exécution) compare la valeur d'un emplacement mémoire à une valeur donnée et, selon le résultat de la comparaison, modifie un emplacement mémoire

R. Grin

Java : threads

page 132

Considérations techniques

- ❑ La JVM a été adaptée pour bénéficier des ces nouvelles instructions
- ❑ La nouvelle API de `java.util.concurrent` s'appuie sur ces nouveautés pour améliorer la gestion du multitâche, en particulier pour améliorer les performances, par rapport à l'utilisation de `synchronize`

R. Grin

Java : threads

page 133

Interface `Callable<V>`

- ❑ Elle représente une tâche à exécuter (par `ExecutorService`), qui renvoie une valeur de type `V`
- ❑ Une seule méthode qui exécute la tâche : `V call() throws Exception`
- ❑ La classe `Executors` contient des méthodes pour envelopper les anciennes interfaces `Runnable`, `PrivilegedAction` et `PrivilegedExceptionAction`, et les transformer en `Callable`

R. Grin

Java : threads

page 134

Interface `Executor`

- ❑ Représente un objet qui exécute des tâches qu'on lui a soumises
- ❑ La soumission d'une tâche est effectuée par la seule méthode de l'interface :
`void execute(Runnable tâche)`

R. Grin

Java : threads

page 135

Exécution des tâches

- ❑ Les tâches soumises à un exécuteur ne sont pas nécessairement exécutées par des threads mais c'est le plus souvent le cas
- ❑ Au contraire de la classe `Thread` qui lance immédiatement un thread avec la méthode `start()`, un exécuteur peut choisir le moment et la manière d'exécuter les tâches qu'on lui a soumises

R. Grin

Java : threads

page 136

Interface `Future<V>`

- ❑ Représente le résultat d'une tâche exécutée en parallèle
- ❑ Les méthodes de l'interface permettent de retrouver le résultat du travail (`get`), d'annuler la tâche (`cancel`) ou de savoir si la tâche a terminé son travail (`isDone`) ou a été annulée (`isCancelled`)

R. Grin

Java : threads

page 137

Méthodes `get`

- ❑ `V get()`
récupère le résultat du travail ; bloque si le travail n'est pas encore terminée
- ❑ `V get(long délai, TimeUnit unité)`
idem `get()` mais ne bloque qu'au plus le temps du délai passé en paramètre
- ❑ Exemple :
`get(50L, TimeUnit.SECONDS)`

R. Grin

Java : threads

page 138

Interface **ExecutorService**

- ❑ **ExecutorService** est une sous-interface de **Executor**
- ❑ Elle ajoute des méthodes pour gérer la fin de l'exécution des tâches et récupérer un **Future** comme résultat de la soumission d'une tâche

R. Grin

Java : threads

page 139

Méthodes de **ExecutorService**

- ❑ Méthodes **submit** pour soumettre une tâche (**Callable** ou **Runnable**) et récupérer un **Future** qui représente l'exécution de la tâche
- ❑ `<T> Future<T> submit(Callable<T> task)`
- ❑ Les méthodes avec **Runnable** en paramètre renvoient un **Future** dont la méthode **get** renvoie la valeur **null** ou une certaine valeur passée en paramètre après la fin de l'exécution (car un **Runnable** ne renvoie pas de valeur)

R. Grin

Java : threads

page 140

Méthodes de **ExecutorService**

- ❑ Méthodes **invokeAll** et **invokeAny** pour exécuter une collection de tâches (**Callable**) et renvoyer le résultat (**Future**) de toutes les tâches (**InvokeAll**) ou d'une seule qui s'est exécutée correctement (**invokeAny**) ; un temps maximum d'exécution peut être passé en paramètre et les tâches non terminées dans le temps imparti sont arrêtées
- ❑ Par exemple,


```
<T> List<Future<T>> invokeAll(
    Collection<? extends Callable<T>> tasks)
    throws InterruptedException
```

R. Grin

Java : threads

page 141

Méthodes de **ExecutorService**

- ❑ Méthodes **shutdown** et **shutdownNow** pour arrêter l'activité de l'exécuteur, d'une façon « civilisée » (avec **shutdown** les tâches en cours se terminent normalement ; seules les tâches en attente d'exécution ne seront pas exécutées) ou brutales (avec **shutdownNow** les tâches en cours d'exécution sont stoppées)
- ❑ `List<Runnable> shutdownNow()` renvoie la liste des tâches qui étaient en attente d'exécution

R. Grin

Java : threads

page 142

Méthodes de **ExecutorService**

- ❑ La méthode **isShutdown** indique si l'exécuteur a été arrêté
- ❑ La méthode **isTerminated** indique si toutes les tâches en cours d'exécution au moment de l'arrêt ont terminé leur exécution
- ❑ La méthode **awaitTermination** bloque en attendant la fin de l'exécution des tâches après un **shutdown** (un délai maximum d'attente est passé en paramètre)

R. Grin

Java : threads

page 143

Interface **ScheduledExecutorService**

- ❑ Hérite de **ExecutorService**
- ❑ 4 méthodes « schedule » qui créent un exécuteur et lancent les exécutions
- ❑ Permet de donner un délai avant l'exécution ou de faire exécuter périodiquement une tâche (comme les timers)
- ❑ On peut indiquer une périodicité moyenne pour les exécutions ou un intervalle fixe entre la fin d'une exécution et le début de la prochaine exécution

R. Grin

Java : threads

page 144

Classe **Executors**

- Contient des méthodes **static** utilitaires pour d'autres interfaces du paquetage : **Executor**, **ExecutorService**, **Callable**,...

R. Grin

Java : threads

page 145

Méthodes **static** « callable »

- Les méthodes **callable** renvoient un **Callable** qui enveloppe un **Runnable**, **PrivilegedAction** ou **PrivilegedExceptionAction**
- **privilegedCallable(Callable)** est réservée à un appel dans la méthode **doPrivileged** de la classe **AccessController** ; elle renvoie un **Callable** dont les autorisations ne dépendent pas du contexte d'appel (voir cours sur la sécurité en Java)

R. Grin

Java : threads

page 146

Les fabriques (1)

- Noms des méthodes commencent par « new »
- Une fabrique de **Thread** (**java.util.ThreadFactory**) peut être passée en paramètre si on veut des types spéciaux de threads (par exemple avec une certaine priorité, ou d'une certaine sous-classe de **Thread**)

R. Grin

Java : threads

page 147

Les fabriques (2)

- **newCachedThreadPool** : crée un pool de threads réutilisables
 - Les threads non utilisés depuis 1 minute sont supprimés
 - Nouveaux threads créés si nécessaire
- **newFixedThreadPool** : crée un pool contenant un nombre fixe de threads
 - Nombre de threads passés en paramètre

R. Grin

Java : threads

page 148

Les fabriques (3)

- **newSingleThreadExecutor** : crée un exécuteur qui exécutera les tâches les unes après les autres
- **newScheduledThreadPool** : crée un pool de threads qui peuvent exécuter des commandes après un certain délai ou périodiquement
- **newSingleThreadScheduledExecutor** : crée un exécuteur qui peut exécuter des commandes après un certain délai ou périodiquement, une commande à la fois

R. Grin

Java : threads

page 149

Exemple d'utilisation des pools

```

ExecutorService pool =
    Executors.newFixedThreadPool(10);
Runnable r = new Runnable() {
    public void run() {
        ...
    }
};
pool.execute(r);

```

R. Grin

Java : threads

page 150

Exemple de pool avec **Callable**

```

ExecutorService pool =
    Executors.newFixedThreadPool(10);
Callable<Integer> c =
    new Callable<Integer>() {
        public Integer call() {
            ...
        }
    };
Future<Integer> f = pool.submit(c);
...
int v = f.get();

```

R. Grin

Java : threads

page 151

- Les quelques transparents suivants présentent les nouvelles collections *thread-safe*

R. Grin

Java : threads

page 152

Classe **ConcurrentHashMap**

- Au contraire de **HashTable**, les lectures ne bloquent ni les autres lectures ni même les mises à jour
- Une lecture reflète l'état de la map après la dernière mise à jour complètement terminée (celles en cours ne sont pas prises en compte)
- Les méthodes sont à peu près les mêmes que celles de **HashMap**

R. Grin

Java : threads

page 153

Classe

CopyOnWriteArrayList

- Variante « *thread-safe* » de **ArrayList** dans laquelle toutes les opérations de mise à jour sont effectuées en faisant une copie du tableau sous-jacent (le tableau n'est jamais modifié)
- Cette variante n'est intéressante que lorsque les parcours de la liste sont bien plus fréquents que les mises à jour
- Les itérateurs reflètent la liste au moment où ils ont été créés ; ils ne peuvent modifier la liste
- Idem pour **CopyOnWriteArraySet**

R. Grin

Java : threads

page 154

Interfaces pour files d'attente

- L'interface **java.util.Queue**, introduite par le JDK 1.5, est semblable à **List** mais correspond à une file d'attente : elle ne permet des ajouts qu'à la fin et des suppressions qu'au début
- Cette contrainte permet plus d'optimisation que pour les implémentations de **List**

R. Grin

Java : threads

page 155

Interfaces pour files d'attente

- L'interface **java.util.concurrent.BlockingQueue** hérite de **Queue** et ajoute une méthode qui se bloque lorsqu'elle veut supprimer un élément dans une file vide (**take**) et une méthode qui se bloque lorsqu'elle ajoute un élément dans une file pleine (**put**)
- Elle permet d'implémenter des files pour des producteurs et des consommateurs

R. Grin

Java : threads

page 156

Classes pour files d'attente

- ❑ **BlockingQueue** est implémenté par les classes
 - **LinkedBlockingQueue**
 - **PriorityBlockingQueue**
 - **ArrayBlockingQueue**
 - **SynchronousQueue**

R. Grin

Java : threads

page 157

Synchronisation des threads

- ❑ La nouvelle API de concurrence offre des objets pour faciliter le codage des programmes qui utilisent plusieurs traitement en parallèle
- ❑ La classe **Semaphore** implémente un sémaphore classique
- ❑ Les classes **CountDownLatch** et **CyclicBarrier** représentent des structures un peu plus complexes qui facilitent la synchronisation de plusieurs threads

R. Grin

Java : threads

page 158

Semaphore

- ❑ Un sémaphore gère des ressources limitées
- ❑ La méthode **public void acquire()** demande une ressource et bloque jusqu'à ce qu'il y en ait une disponible ; lorsque la méthode retourne, une ressource en moins est disponible via le sémaphore
- ❑ La méthode **public void release()** rend une ressource ; lorsque la méthode retourne, une ressource de plus est disponible via le sémaphore

R. Grin

Java : threads

page 159

Semaphore

- ❑ Plusieurs variantes des méthodes de base **acquire** et **release** sont disponibles (pour acquérir ou rendre plusieurs ressources en même temps ou pour ne pas se bloquer en attente de ressources)

R. Grin

Java : threads

page 160

CountDownLatch et CyclicBarrier

- ❑ Les classes **CountDownLatch** et **CyclicBarrier** facilitent la décomposition d'un traitement en plusieurs sous-traitements parallèles
- ❑ A certains points d'exécution le traitement ne peut se poursuivre que si les sous-traitements ont terminé une partie de leur travail
- ❑ La classe **Phaser**, introduite par le JDK 7, est plus souple que ces 2 classes (voir javadoc)

R. Grin

Java : threads

page 161

CyclicBarrier

- ❑ Représente une barrière derrière laquelle n threads (n est un paramètre du constructeur) qui représentent les sous-traitements, attendent par la méthode **barriere.await()**
- ❑ Quand les n threads y sont arrivés, la barrière « se lève » et les threads continuent leur exécution
- ❑ La barrière est « cyclic » car elle se rabaisse ensuite, et les threads y sont à nouveau bloqués

R. Grin

Java : threads

page 162

CyclicBarrier

- Un constructeur permet de passer en paramètre un **Runnable** qui sera exécuté juste avant que la barrière ne se relève

R. Grin

Java : threads

page 163

CountDownLatch

- Rôle similaire à une barrière mais il ne peut servir qu'une seule fois et le fonctionnement est différent : l'arrivée au point d'attente est dissociée de l'attente elle-même
- Le constructeur prend en paramètre un nombre *n*
- Les sous-traitements peuvent décrémenter ce nombre par `cdl.countDown()`
- Des threads peuvent aussi attendre que *n* soit égal à 0 par `cdl.await()`

R. Grin

Java : threads

page 164

CountDownLatch

- Peut être utilisé pour le même usage qu'une barrière : les sous-traitements appellent `countDown` et `await` juste après
- Mais aussi comme un starter : *n* est initialisé à 1 et tous les threads commencent par attendre par `await` ; ensuite un thread « starter » appelle `countDown` pour faire démarrer tous les threads

R. Grin

Java : threads

page 165

Phaser

- La classe **Phaser**, introduite par le JDK 7, étend les possibilités de **CountDownLatch** et **CyclicBarrier**, par exemple (voir javadoc pour plus d'information) :
- Des nouvelles tâches peuvent être enregistrées en cours de traitement
- Un traitement différent peut être exécuté à l'arrivée d'une tâche et lorsqu'elle attend
- On peut forcer une fin d'exécution
- Un phaser peut être surveillé de l'extérieur

R. Grin

Java : threads

page 166

D'autres cas de « *happens-before* »

- L'ajout d'un objet dans une collection « *thread-safe* » de cette API *happens-before* (voir section « Difficultés du multi-tâche) un accès ou une suppression de l'objet dans la collection
- Les actions effectuées pour le calcul d'un **Future** *happens-before* la récupération de la valeur du **Future**
- Une action exécutée par un thread avant une attente à une barrière cyclique (`await`) *happens-before* l'action (optionnelle) exécutée par la barrière au moment où elle se lève, qui elle-même, *happens-before* une action qui suit `await`

R. Grin

Java : threads

page 167

Fork - join

R. Grin

Java : threads

page 168

Présentation

- ❑ Il est souvent complexe de programmer une tâche pour profiter des nouveaux processeurs multi-cœurs
- ❑ Le JDK 7 fournit le framework « fork-join » qui facilite cette programmation dans le cas fréquent où une tâche peut se décomposer récursivement en sous-tâches de même nature
- ❑ Les sous-tâches sont lancées de manière asynchrone (*fork*) et on attend leur exécution (*join*) pour combiner leur résultat

R. Grin

Java : threads

page 169

Exemple

- ❑ Un tri fusion d'une liste ou d'un tableau : chaque processeur se charge de la moitié des valeurs à trier et ensuite les 2 parties triées sont fusionnées
- ❑ Chacune des 2 parties peut elle-même être récursivement partagées en 2 parties qui seront fusionnées, et ainsi de suite
- ❑ Lorsque les parties sont assez petites, on les trie directement, sans les décomposer en 2, ce qui arrête la descente récursive

R. Grin

Java : threads

page 170

Le plus du framework « fork-join »

- ❑ Le nombre de processeurs est limité ; le framework fork-join attribue par défaut autant de threads/exécuteurs que de processeurs pour accomplir la tâche principal (possible de donner un autre nombre dans le constructeur)
- ❑ Les sous-tâches sont attribuées aux exécuteurs au fur et à mesure qu'elles sont créées (ajoutées à une collection de type **Deque**) attachée à chaque exécuteur)

R. Grin

Java : threads

page 171

Le plus du framework « fork-join »

- ❑ Le framework gère les exécuteurs de telle sorte qu'ils chôment le moins souvent possible
- ❑ Quand un exécuteur rencontre un join, il exécute une autre tâche
- ❑ Quand un exécuteur n'a plus de tâche à exécuter, il peut aller « voler » une tâche d'un autre exécuteur pour l'exécuter
- ❑ S'il n'y a rien à voler, il se « repose » momentanément en attendant l'arrivée d'une nouvelle tâche à accomplir

R. Grin

Java : threads

page 172

Classes

- ❑ 2 classes du paquetage `java.util.concurrent` permettent de programmer ce genre de tâche :
 - **ForkJoinPool**, une extension de **AbstractExecutorService** qui implémente l'algorithme de division des tâches en 2 sous-tâches
 - **ForkJoinTask<V>** qui représente la tâche à exécuter

R. Grin

Java : threads

page 173

ForkJoinPool

- ❑ Constructeur **ForkJoinPool(int nbThreads)** ; si on ne passe pas de paramètre, prend par défaut le nombre de processeurs
- ❑ Méthode **invoke(tâche)** lance la tâche (qui sera décomposée en sous-tâches)

R. Grin

Java : threads

page 174

2 types de tâche

- Héritent de `ForkJoinTask<V>`
- `RecursiveTask<V>` dont la méthode `compute` retourne une valeur (de type `v`)
- `RecursiveAction` dont la méthode `compute` ne retourne aucune valeur
- Le développeur doit écrire le code à exécuter dans chaque sous-tâche dans une classe fille d'une de ces classes, dans la méthode `protected compute()`

R. Grin

Java : threads

page 175

Quelques méthodes de `ForkJoinTask`

- `fork()` : exécute la tâche de façon asynchrone
- `v join()` : retourne la valeur calculée par la tâche quand elle a fini son exécution
- `invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)` (méthode surchargée ; voir javadoc) : lance les tâches `t1` et `t2` de façon asynchrone et attend leur fin (le plus simple)

R. Grin

Java : threads

page 176

Exemple schématique

```
compute() {
    if (portion du travail est petite) {
        faire le travail directement
    }
    else {
        diviser le travail en sous-tâches
        invokeAll(sous-tâche1, sous-tâche2,...)
        attendre les sous-tâches
        composer les résultats des sous-tâches
    }
}
```

R. Grin

Java : threads

page 177

Exemple de `RecursiveAction`

- Incrémenter les valeurs d'un tableau (exemple de la javadoc de `RecursiveAction`) :

```
class IncrTask
    extends RecursiveAction {
    final long[] t; final int lo;
    final int hi;
    IncrTask(long[] t, int lo, int hi) {
        this.t = t; this.lo = lo;
        this.hi = hi; }
    // suite prochain transparent...
```

R. Grin

Java : threads

page 178

Exemple de `RecursiveAction`

```
protected void compute() {
    if (hi - lo < SEUIL) {
        for (int i = lo; i < hi; ++i)
            t[i]++;
    }
    else {
        int mid = (lo + hi) >>> 1;
        invokeAll(
            new IncrTask(t, lo, mid),
            new IncrTask(t, mid, hi));
    }
}
```

R. Grin

Java : threads

page 179

Exemple de `RecursiveTask`

- Calcul des nombres de Fibonacci (exemple de la javadoc de `RecursiveTask`) :

```
class Fibonacci
    extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }
    public Integer compute() {
        if (n <= 1) return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

R. Grin

Java : threads

page 180

Lancement de l'exécution

```
ForkJoinPool pool =  
    new ForkJoinPool();  
pool.invoke(new Fibonacci(30));
```