SÉANCE 16

CONCURRENCE JAVA

SUJETS

Encore plus sur les Principes de Base des Threads Java

- Sleep
- Join
- Interrupt

Verrous intrinsèques de Java

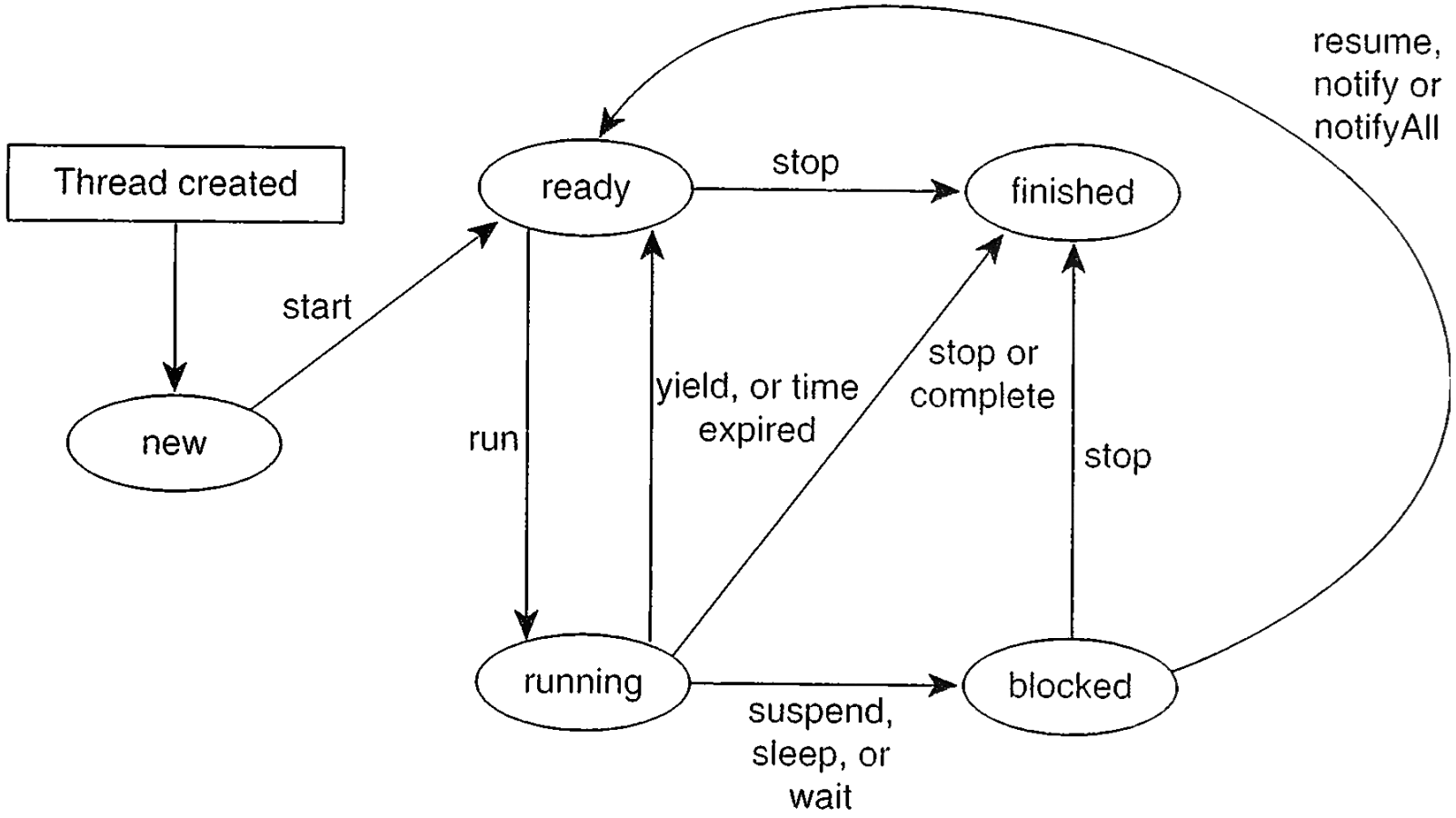
Moniteurs Implémentés avec des Verrous intrinsèques

Moniteurs Implémentés avec des Interfaces Lock et Condition

Sémaphores Java

Variables Atomiques

ÉTATS THREAD



CRÉER DES THREADS EN ÉTENDANT LA CLASSE THREAD

uOttawa
L'Université canadienne
Canada's university

```
// Custom thread class
public class CustomThread extends Thread
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Override the run method in Thread
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create a thread
    CustomThread thread = new CustomThread(...);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

CRÉER DES THREADS EN IMPLÉMENTANT L'INTERFACE EXÉCUTABLE

uOttawa
L'Université canadienne
Canada's university

```
// Custom thread class
public class CustomThread
  implements Runnable
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Implement the run method in Runnable
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }

  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create an instance of CustomThread
    CustomThread customThread
     = new CustomThread(...);

    // Create a thread
    Thread  thread = new  Thread(customThread);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

THREAD GROUPS

Un thread group est un ensemble de threads

Certains programmes continents plusieurs threads avec des fonctions similaires

- On peut les regrouper ensemble et effectuer des opérations sur le groupe entier

Ex., on peut suspendre ou résumer tous les threads dans un groupe en même temps.

UTILISATION DES GROUPES DE THREAD

uOttawa
L'Université canadienne
Canada's university

Construisez un thread group:

```
ThreadGroup g = new ThreadGroup("thread
 group");
```

Placez un thread dans le thread group:

```
Thread t = new Thread(g, new ThreadClass());
```

Trouvez combien de threads dans un groupe sont couramment en train d'exécuter:

```
System.out.println("the number of runnable
 threads in the group " + g.activeCount());
```

Trouvez à quel groupe le thread appartient-il:

```
theGroup = t.getThreadGroup();
```

PRIORITÉS

Les priorités des threads n'ont pas besoin d'être identiques

La priorité du thread par défaut est:

- NORM_PRIORITY(5)

La priorité est un nombre entier entre 0 et 10, où:

- MAX_PRIORITY(10)
- MIN_PRIORITY(0)

Vous pouvez utiliser:

- `setPriority(int)`: changer la priorité de ce thread
- `getPriority()`: retourner la priorité de ce thread

SLEEP

Faire un thread dormir (sleep) pour plusieurs millisecondes

- Très populaire avec les applications de jeux (animation 2D ou 3D)

  `Thread.sleep(1000);`

  `Thread.sleep(1000, 1000);` (la précision dépend du système)
- Notez que ces méthodes sont statiques
- Elles génèrent un `InterruptedException`

JOIN

Si vous créez plusieurs threads, chacun est responsable pour des calculs

Vous pouvez attendre que le thread meurt (die)

- Avant de rassembler les résultats de ces threads

Afin d'achever ceci, on utilise la méthode « join » définie dans la classe Thread

```
try {thread.join();}
catch (InterruptedException e){e.printStackTrace();}
```

L'Exception est générée si un autre thread a interrompu le thread courant

EXEMPLE JOIN

```
public class JoinThread {
    public static void main(String[] args) {
        Thread thread2 = new Thread(new WaitRunnable());
        Thread thread3 = new Thread(new WaitRunnable());

        thread2.start();
        try {thread2.join();} catch (InterruptedException e) {e.printStackTrace();}

        thread3.start();
        try {thread3.join(1000);} catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

INTERRUPTION DES THREADS

uOttawa
L'Université canadienne
Canada's university

Dans Java, on n'a aucune façon de forcer un Thread d'arrêter

- Si le Thread n'est pas implémenté correctement, il peut continuer son exécution indéfiniment

Mais on peut interrompre un Thread avec la méthode `interrupt()`

- Si le Thread dort (sleep) ou joint (join) un autre Thread, un `InterruptedException` est généré
- Dans ce cas, le statut interrompu du thread est remis a faux (false)

EXEMPLE INTERRUPT

```
public class InterruptThread {

  public static void main(String[] args) {

    Thread thread1 = new Thread(new WaitRunnable());
    thread1.start();

    try {Thread.sleep(1000);}
    catch (InterruptedException e){e.printStackTrace();}
    thread1.interrupt();
  }
```

EXEMPLE INTERRUPT

```
  private static class WaitRunnable implements Runnable {
    @Override
    public void run() {
      System.out.println("Current time millis: " + System.currentTimeMillis());
      try {Thread.sleep(5000);}
      catch (InterruptedException e) {
        System.out.println("The thread has been interrupted");
        System.out.println("The thread is interrupted: "+Thread.currentThread().isInterrupted());
      }
      System.out.println("Current time millis: " + System.currentTimeMillis());
    }
}
```

Sample Output:
Current time millis : 1274017633151
The thread has been interrupted
The thread is interrupted : false
Current time millis : 1274017634151

VERROUS INTRINSÈQUES

Tout le code qui peut être modifié simultanément par plusieurs threads doit être Thread Safe

Considérez le code simple suivant:

```
public int getNextCount(){
  return ++counter;
}
```

Un incrément comme ceci n'est pas une action atomique, il implique:

- Lire la valeur actuelle de `counter`
- Ajouter un (1) à sa valeur actuelle
- Stocker le résultat dans la mémoire

VERROUS INTRINSÈQUES

uOttawa
L'Université canadienne
Canada's university

Si on a deux threads qui invoquent `getNextCount()`, la séquence suivante d'événements peut se passer (parmi plusieurs scénarios possibles):

1	*Thread 1 :* reads counter, gets **0**, adds **1**, so **counter = 1**	2	*Thread 2 :* reads counter, gets **0**, adds **1**, so **counter = 1**
3	*Thread 1 :* writes **1** to the counter field and **returns 1**	4	*Thread 2 :* writes **1** to the counter field and **returns 1**

Conséquemment, on doit utiliser un verrou sur l'accès à "counter"

On peut ajouter un tel verrou à une méthode en utilisant simplement le mot-clé: `synchronized`

```
public synchronized int getNextCount(){
   return ++counter;
}
```

Ceci garantit que seulement un thread exécute la méthode

Si on a plusieurs méthodes avec le mot-clé synchronisé, seulement une méthode peut être exécutée à la fois

- Ceci s'appelle un **Verrou intrinsèque**

VERROUS INTRINSÈQUES

uOttawa
L'Université canadienne
Canada's university

Chaque objet Java possède un verrou intrinsèque associé avec lui (souvent appelé tout simplement moniteur)

Dans le dernier exemple, on a utilisé ce verrou pour synchroniser l'accès à une méthode

- Au lieu, on peut choisir de synchroniser l'accès à un bloc (ou segment) de code

```
public int getNextValue() {
        synchronized (this) {return value++;}
}
```

- Alternativement, on peut utiliser le verrou d'un autre objet

```
public int getNextValue() {
        synchronized (lock) {return value++;}
}
```

- Ce dernier est utile puisqu'il nous permet d'utiliser plusieurs verrous pour la sécurité du thread dans une classe simple

VERROUS INTRINSÈQUES

Alors, on a mentionné que chaque objet Java possède un verrou intrinsèque associé avec lui

Qu'en est-il des méthodes statiques qui ne sont pas associées avec un objet particulier?

- Il existe aussi un verrou intrinsèque associé avec la classe
- Utilisé seulement pour les méthodes à classe synchronisée (statique)

MONITEURS JAVA

Mauvaises nouvelles: Dans Java, il n'y a pas un mot-clé pour créer directement un moniteur

Bonnes nouvelles: il existe plusieurs mécanismes pour créer des moniteurs

- Le plus simple, utilise les connaissances qu'on a déjà accumulées concernant les verrous intrinsèques

MONITEURS JAVA

Les verrous intrinsèques peuvent être utilisés effectivement pour l'exclusion mutuelle (synchronisation de compétition)

On a besoin d'un mécanisme pour implémenter la synchronisation de coopération

- En particulier, on a besoin de permettre aux threads de se suspendre si une condition empêche leur exécution dans un moniteur
- Ceci est fait en utilisant les méthodes `wait()` et `notify()`

Ces deux méthodes sont très importantes qu'elles ont été définies dans la classe Object…

OPÉRATIONS "WAIT"

uOttawa
L'Université canadienne
Canada's university

`wait()`

Permet au thread actuel d'abandonner le moniteur et attendre (**wait**) jusqu'à ce qu'un autre thread entre dans le même moniteur et fait appel à **`notify()ou notifyAll()`**

`wait(long timeout)`

Permet au thread actuel d'attendre (**wait**) jusqu'à ce qu'un autre thread invoque la méthode **`notify() ou notifyAll()`**, ou bien que le temps spécifié est terminé

OPÉRATIONS "NOTIFY"

uOttawa
L'Université canadienne
Canada's university

`notify()`

Réveille un seule thread qui attend sur le moniteur de cet objet (verrou intrinsèque). Si plus qu'un seul thread attendent, le choix est arbitraire (est-ce que c'est juste?)

Le thread réveillé ne pourra pas procéder jusqu'à ce que le thread actuel quitte le verrou.

`notifyAll()`

Réveille tous les threads qui attendent le moniteur de cet objet.

Le thread réveillé ne pourra pas procéder jusqu'à ce que le thread actuel quitte le verrou.

Le thread suivant qui verrouille ce moniteur est aussi choisi au hasard.

EXEMPLE DE MONITEUR BASÉ SUR UN VERROU INTRINSÈQUE


```
public class BufferMonitor{
    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    public synchronized void deposit (int item ) throws InterruptedException{
        while (buffer.length == filled){
            wait(); // blocks thread
        }

        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;

        notify(); // free a task that has been waiting on a condition
    }
```

EXEMPLE DE MONITEUR BASÉ SUR UN VERROU INTRINSÈQUE


```
public synchronized int fetch() throws InterruptedException{
    while (filled == 0){
        wait(); // block thread
    }

    int item = buffer[next_out];
    next_out = (next_out + 1) % buffer.length;
    filled--;

    notify(); // free a task that has been waiting on a condition

    return item;
}
}
```

UN AUTRE MÉCANISME POUR CRÉER DES MONITEURS

uOttawa
L'Université canadienne
Canada's university

On peut aussi créer un moniteur en utilisant l'interface Java `Lock`

- **`ReentrantLock`** est l'implémentation la plus populaire de **`Lock`**

`ReentrantLock` définit deux constructeurs:

- Constructeur de Défaut
- Constructeur qui prend un Booléen (spécifiant si le verrou est juste)

Dans une situation de verrou juste, les threads vont avoir accès au verrou dans le même ordre qu'ils l'ont demandé (FIFO)

- Sinon, le verrou ne garantit aucun ordre particulier
- La justesse exige plus de computations (en termes de traitement), et donc, doit être seulement utilisé si requis

Afin d'acquérir le verrou, vous devez simplement utiliser la méthode `lock`, et pour le lâcher, faites appel à `unlock`

EXEMPLE “LOCK”

```
public class SimpleMonitor {
   private final Lock lock = new ReentrantLock();
   public void testA() {
      lock.lock();
      try {//Some code}
      finally {lock.unlock();}
   }
   public int testB() {
      lock.lock();
      try {return 1;}
      finally {lock.unlock();}
   }
}
```

QUESTION??

Pourquoi devons-nous avoir un bloc try-finally dans l'exemple précédent?

UN AUTRE MÉCANISME POUR CRÉER DES MONITEURS

uOttawa
L'Université canadienne
Canada's university

Comment implémenter les conditions?

- Sans pouvoir attendre une condition, les moniteurs seront inutiles…
  - *La Coopération n'est pas possible*

Il existe une classe spécifique qui a été développée juste pour cet effet: Classe `Condition`

- On crée un exemple de **`Condition`** en utilisant la méthode **`newCondition()`** définie dans l'interface **`Lock`**

EXEMPLE DE MONITEUR TAMPON (ENCORE)


```
public class BufferMonitor {

    int [] buffer = new int [100];
    int next_in = 0, next_out = 0, filled = 0;

    private final Lock lock = new ReentrantLock(true);
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
```

EXEMPLE DE MONITEUR TAMPON (ENCORE)


```
public void deposit (int item ) throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (buffer.length == filled){
            notFull.await(); // blocks thread (wait on condition)
        }
        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;
        notEmpty.signal(); // signal thread waiting on the empty condition
    }
    finally{
        lock.unlock();// Whenever you lock, you must unlock
    }
}
```

EXEMPLE DE MONITEUR TAMPON (ENCORE)


```
public void fetch () throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (filled == 0){
            notEmpty.await(); // blocks thread (wait on condition)
        }
        int item = buffer[next_out];
        next_out = (next_out + 1) % buffer.length;
        filled--;
        notFull.signal(); // signal thread waiting on the full condition
    }
    finally{
        lock.unlock();// Whenever you lock, you must unlock
    }
    return item;
}
}
```

“LOCK” VS “SYNCHRONIZED”

Les moniteurs implémentés avec des classes `Lock` et `Condition` ont quelques avantages comparés avec l’implémentation basée sur le *verrou intrinsèque*:

1. La capacité d’avoir plus qu’une variable de condition par moniteur (voir exemple précédent)
2. La capacité d’avoir un verrou juste (souvenez-vous que les blocs de code ou méthodes synchronisés ne garantissent pas la justesse)
3. La capacité de vérifier si le verrou est actuellement occupé (en faisant appel à la méthode `isLocked()`)
    - Alternativement, on peut faire appel à `tryLock()` qui acquiert le verrou seulement s’il n’est pas occupé par un autre thread
4. La capacité d’obtenir la liste de threads qui attendent le verrou (en faisant appel à la méthode `getQueuedThreads()`)

La liste ci-dessus n’est pas exhaustive…

“VERROU” VS “SYNCHRONISÉ”

Les désavantages de `Lock` et `Condition` :

1. Besoin d’ajouter le code d’acquisition et relâchement de lock

2. Besoin d’ajouter un bloc try-finally

SÉMAPHORES JAVA

Java définit une classe de sémaphores:
`java.util.concurrent.Semaphore`

Créer un sémaphore comptant:

```
Semaphore available = new Semaphore(100);
```

Créer un sémaphore binaire:

```
Semaphore available = new Semaphore(1);
```

On implémentera notre propre classe de sémaphores plus tard

EXEMPLE DE SÉMAPHORE

```
public class Example {
    private int counter= 0;

    private final Semaphore mutex = new Semaphore(1)

    public int getNextCount() throws InterruptedException {
        mutex.acquire();
        try {
            return ++counter;
        } finally {
            mutex.release();
        }
    }
}
```

EXERCICE DE SÉMAPHORE

Malgré qu'une classe de sémaphore soit incluse dans la librairie standard de Java, cependant, avec les connaissances que vous avez accumulé jusqu'à présent,

Pouvez-vous créer une classe de Sémaphores Comptants en utilisant des Verrous intrinsèques?

EXERCICE DE SÉMAPHORE

uOttawa
L'Université canadienne
Canada's university

```
public class Semaphore{
    private int count;
    public Semaphore () {
         count = 0;
    }
    public synchronized void acquire(){
        while (count <=0){
            try {
                wait();
            }
            catch (InterruptedException e){}
        }
         count--;
    }
```

```
public synchronized void release(){
        ++count;
        notify();
    }
}
```

VARIABLES ATOMIQUES

uOttawa
L'Université canadienne
Canada's university

Si on a besoin de la synchronisation pour une seule variable dans notre classe, on peut utiliser une classe atomique pour la rendre thread safe:

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference

Ces classes utilisent des mécanismes de "hardware" à bas niveau pour assurer la synchronisation

- Ceci résulte en une meilleure performance

EXEMPLES DE VARIABLES ATOMIQUES


```
public class AtomicCounter {
    private final AtomicInteger value = new AtomicInteger(0);
    public int getValue(){
        return value.get();
    }
    public int getNextValue(){
        return value.incrementAndGet();
    }
    public int getPreviousValue(){
        return value.decrementAndGet();
    }
}
```

Autres opérations possibles:
`getAndIncrement()`, `getAndAdd(int x)`, `addAndGet(int x)`…

MERCI!

QUESTIONS?