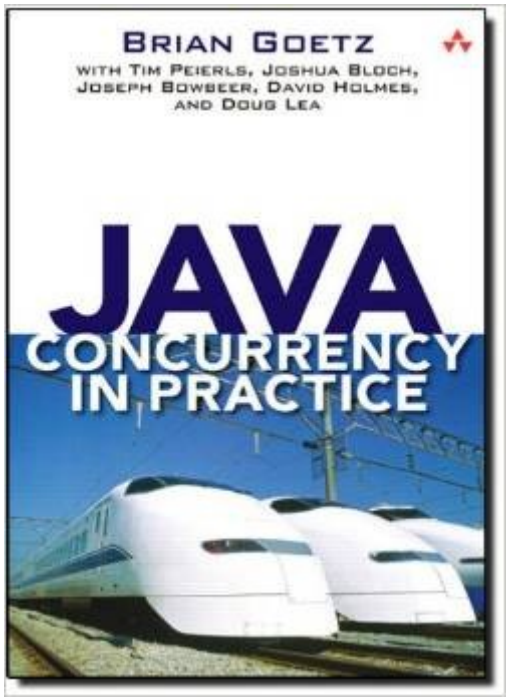
Concurrence
Threads

Rémi Forax

Avant propos

Ce cours est **pas évident** et **doit être appris**



Le livre du cours:
Java Concurrency in Practice
by Brian Goetz and al.

Le chapitre 17 dans la specification du langage Java
http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

Concurrence ?

Executer plusieurs codes en même temps sur une même machine

- Cela peut être le même code
  - Server Web
- Cela peut être des codes différents
  - Jeu, affichage / AI

Ne pas confondre avec distribué, qui implique plusiuers machines et la communication inter-machines

Pourquoi ?

Si le programme fait des entrées/sorties (IO bound)

- cela permet de traiter une autre requête pendant l'attente de donnée I/O

Si le programme fait des calculs (CPU bound)

- Cela permet d'utiliser tous les cores disponibles

Le modèle

Plusieurs fils d'exécution appelé thread

- Chaque thread à sa propre pile (*stack*)

  (là où on stocke les valeurs liés aux opérations courante)

Un ordonnanceur (*scheduler*) qui demande l'exécution des threads sur les cores physiques

- pendant un quantum de temps

Une seule mémoire partagée appelé le tas (*heap*)

- Unified Memory Access ou NUMA
- Utilisé pour l'échange d'informations entre threads

En Java (ou en C)

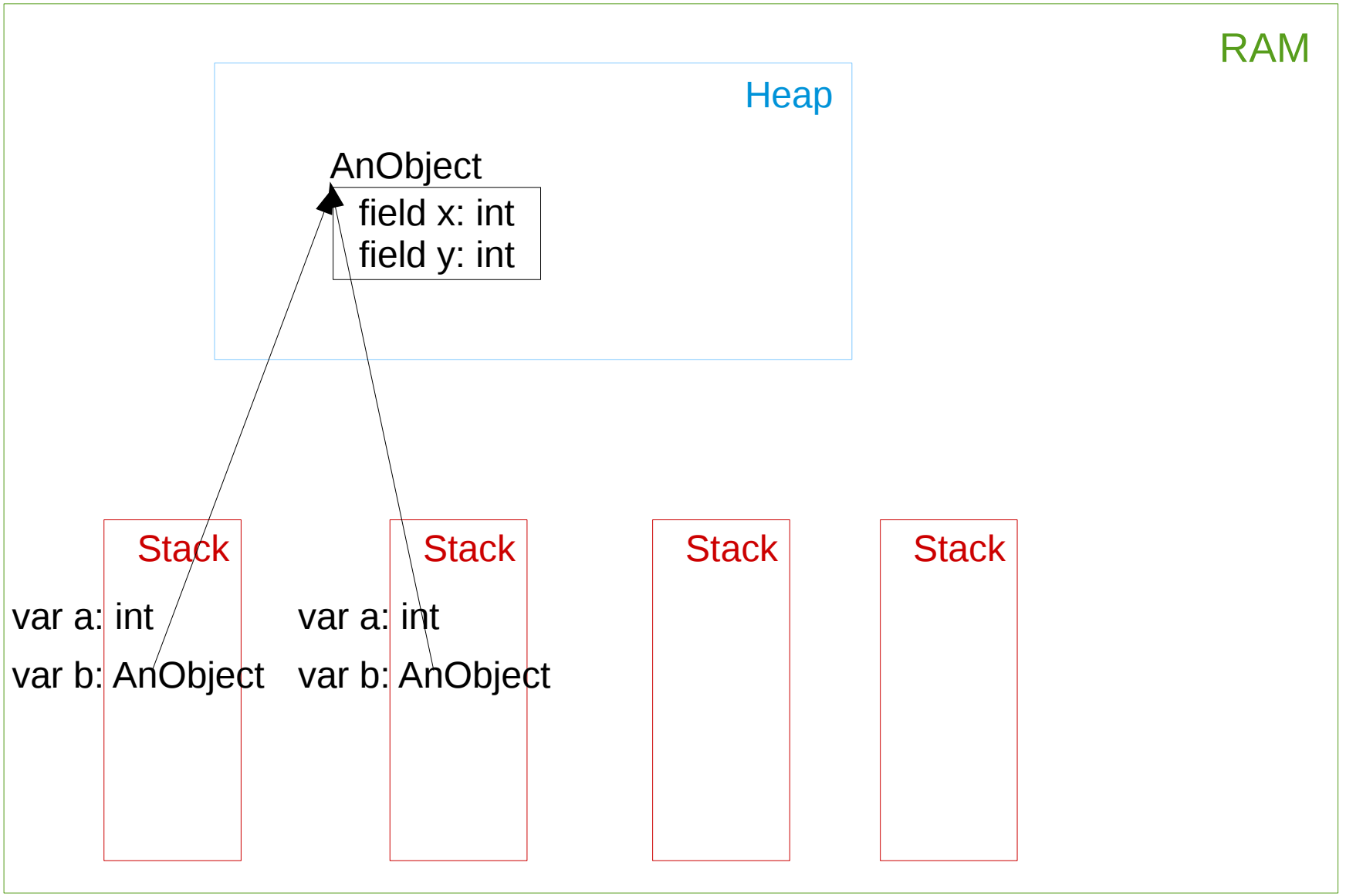
Un objet permet la création de thread système

- java.lang.Thread (pthread)
- puis démarrage thread.start() (pthread_start)

Les variables locales sont stockées sur la pile

Les champs des objets (des structs) alloué par new (malloc) sont stockés sur le tas

Modèle en mémoire



java.lang.Thread

Object qui contrôle une thread système

Il existe AVANT et APRES que la thread système existe

Créée avec un code a exécuter en paramètre sous forme d'un Runnable

Ce que fait start()

- Alloue une nouvelle pile
- Demande la création d'une thread système
- La thread système exécute runnable.run()
- Une fois l'exécution de run() fini, la thread système meurt

L'objet Thread Java ne meurt que si garbage collecté

Exemple

On a besoin d'un code à exécuter

```
public class Code implements Runnable {
  public void run() {
    System.out.println("Hello Thread");
  }
}
```

On peut maintenant créer une Thread et la démarrer

```
public class Hello {
  public static void main(String[] args) {
    Runnable code = new Code();
    Thread thread = new Thread(code);
    thread.start();
  }
}
```

Avec une Lambda

On utilise la syntaxe des lambdas pour simplifier l'écriture du code

```
public class Hello {
  public static void main(String[] args) {
    Runnable code = () -> {
      System.out.println("Hello Thread");
    };
    Thread thread = new Thread(code);
    thread.start();
  }
}
```

java.lang.Thread et héritage

En fait, java.lang.Thread implante Runnable ??

```
class Thread implements Runnable {    <---  WTF ?
  private final Runnable runnable;

  public void run() {                          <---  WTF ?
    // do nothing
  }

  public Thread(Runnable runnable) {
    this.runnable = this.runnable;
  }
  public Thread() {
    this.runnable = this;          <---  ahah !
  }
}
```

Ahhh

On peut hériter de java.lang.Thread et redéfinir run()

```
public class Code extends Thread {
  public void run() {
    System.out.println("Hello Thread");
  }
}
```

mais **on le fait pas** car

- Thread doit avoir 1 seul responsabilité
- Mauvais utilisation de l'héritage cf POO

Thread.currentThread()

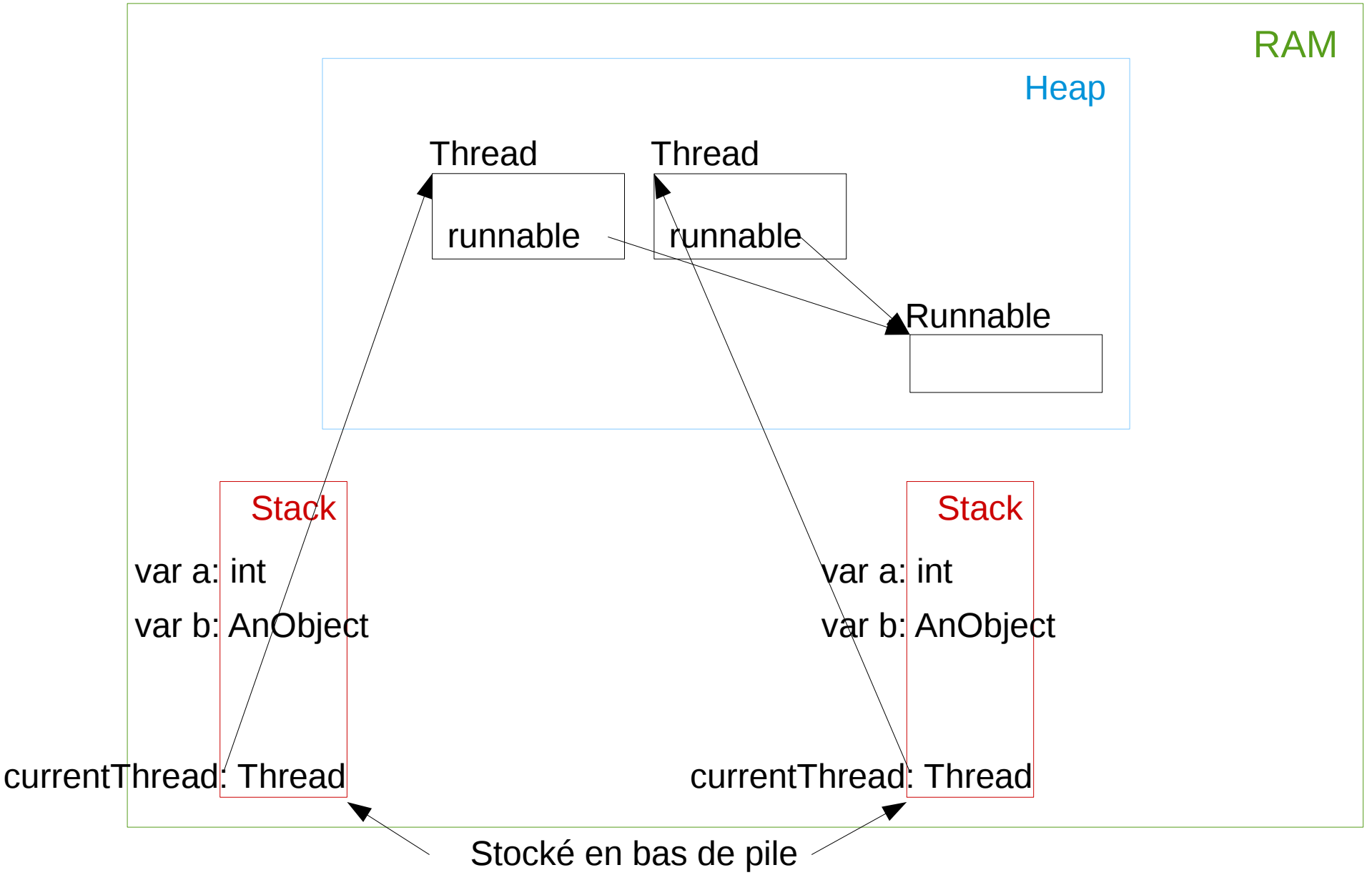
Un même code peut être exécuter par plusieurs threads

On peut demander quelle est la thread qui exécute le code

```
public static void main(String[] args) {
    Runnable code = () -> {
      System.out.println(Thread.currentThread());
    };
    Thread thread1 = new Thread(code);
    Thread thread2 = new Thread(code);
    thread1.start();
    thread2.start();
  }
```

Affiche ??

j.l.Thread en mémoire



Méthodes de j.l.Thread

Par convention,

- Une méthode statique de java.lang.Thread s'appliquera à la thread courante
  - Thread.currentThread()
  - Thread.sleep()
  - Thread.interrupted()
- Une méthode d'instance s'applique à la thread en premier paramètre
  - thread.setName()
  - thread.join()

Thread et ordonnancement

L'ordonnanceur ordonnance les threads systèmes comme il veut

- On ne contrôle par l'ordre dans lequel plusieurs threads s'exécute
- L'ordre change d'une exécution à l'autre :(
- L'ordonnanceur est équitable (*fair*)
  - garantie que au bout d'un certain temps d'exécution, toutes les threads auront eut le même temps d'accès aux cores

Ordre d'exécution

Le code suivant peut donc afficher
```
  Thread[Thread-0,5,main]
  Thread[Thread-1,5,main]
```
ou
```
  Thread[Thread-1,5,main]
  Thread[Thread-0,5,main]
```

```
public static void main(String[] args) {
  Runnable code = () -> {
    System.out.println(Thread.currentThread());
  };
  Thread thread1 = new Thread(code);
  Thread thread2 = new Thread(code);
  thread1.start();
  thread2.start();
}
```

La thread main

Le main() est aussi exécuter par une thread, dont le nom est “main”

```
public static void main(String[] args) {
  Thread t = Thread.currentThread();
  System.out.println(t.getName());   // main
}
```

Cycle de vie d'une thread

Il est possible de demander l'état de la thread système (avec Thread.getState())

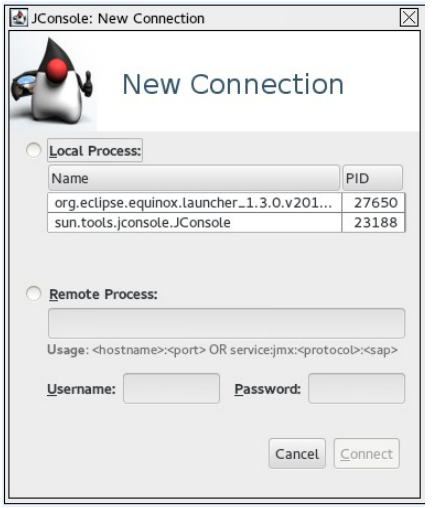
- NEW : pas encore démarré
- RUNNABLE: la thread peut s'exécuter sur un coeur (pourvu que le scheduler soit d'accord)
- BLOCKED : la thead est bloqué sur un verrou (cf cours 2)
- WAITING : la thread est bloqué en attente d'une autre thread (cf plus loin et cours 2)
- TIMED_WAITING : bloqué en attente mais l'attente est borné dans le temps (cf cours 2)
- TERMINATED : la thread système est morte

Ce qui est assez pratique pour le debug !

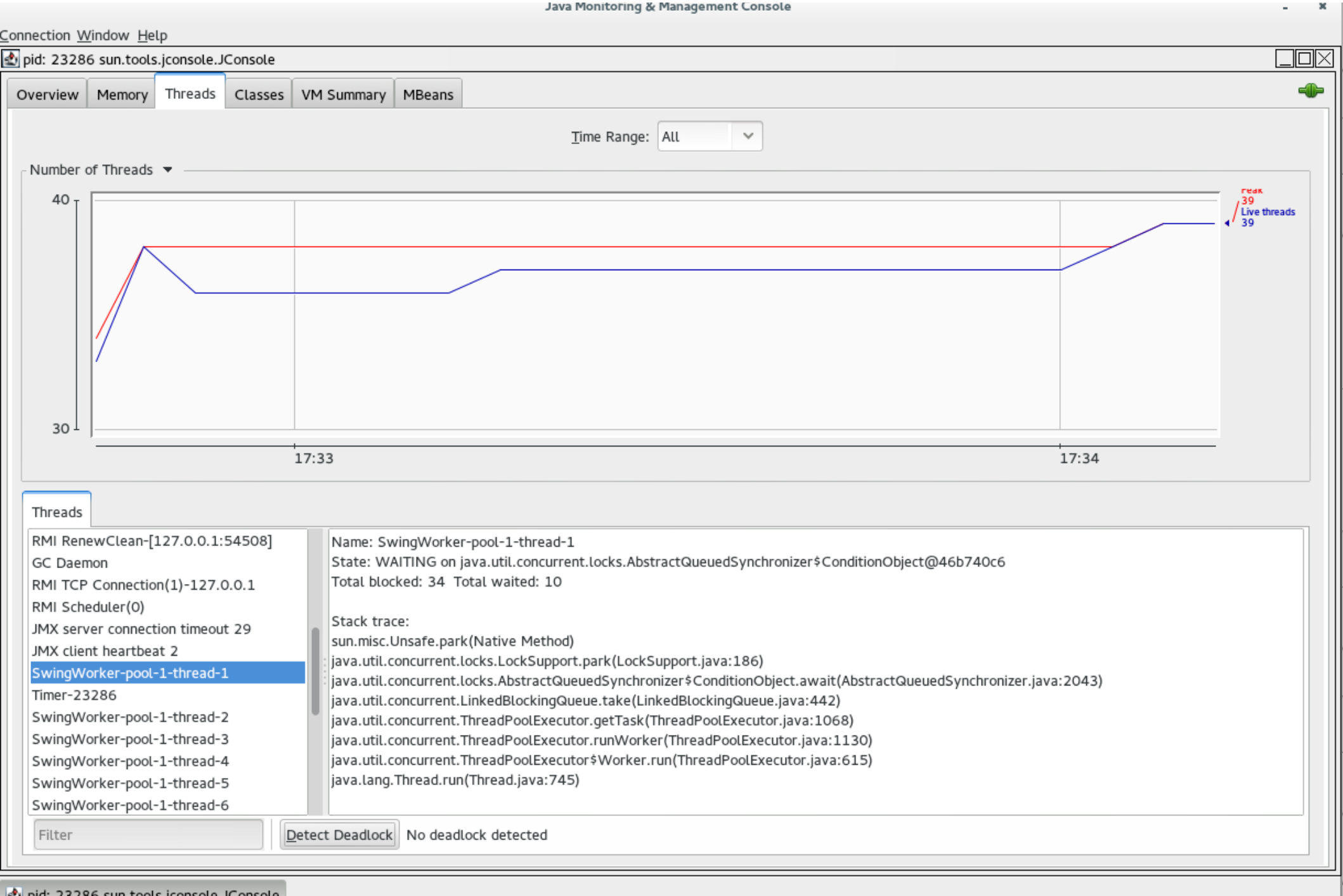
JConsole

Outil simple de monitoring de l'exécution d'un processus java

Possède un onglet Thread, qui décrit l'état de toutes les threads



Outil simple de monitoring de l'exécution d'un processus java



Les joies de la concurrence

Attention, Warning, Archtung !
le code

```
if (Thread.getState() == State.RUNNABLE) {
  ...
}
```

ne fait pas ce que vous croyez qu'il fait !

Le scheduler peut dé-scheduler la thread après l'appel à getState() et avant le ==, donc on test un état du passé et pas l'état actuel

C'est aussi vrai avec tous les appels sur les fichiers, par exemple, if (Files.exist(path)) { ... }

Envoyer des arguments à un Runnable

Il est possible d'accéder aux variables locales hors d'une lambda si la variable est **effectivement final** (assigné 1 seul fois)

```
public static void main(String[] args) {
  Runnable code = () -> {
    System.out.println(args[0]);
  };
  new Thread(code).start();
}
```

Cela garantie qu'il n'est pas possible qu'une Thread modifie la valeur d'une variable locale d'une autre thread.

Envoyer des arguments à un Runnable

Le code suivant ne marche pas

```
public static void main(String[] args) {
  for(int i = 0; i < 4; i++) {
    Runnable code = () -> {
      System.out.println(i);
    };
    new Thread(code).start();
  }
}
```

car 'i' n'est pas effectivement final

Envoyer des arguments à un Runnable

Deux façons de corriger le code, soit on introduit une variable temporaire effectivement final

```
for(int i = 0; i < 4; i++) {
  int j = i;
  new Thread(() -> {
    System.out.println(j);
  }).start();
}
```

soit on utilise IntStream.range().forEach()

```
IntStream.range(0, 4).forEach(i -> {
  new Thread(() -> {
    System.out.println(i);
  }).start();
});
```

Attendre la fin d'un thread

La thread courante peut attendre la fin (la mort) d'une autre thread

```
otherThread.join();
```

Cela permet d'introduire un ordre sur l'exécution sur le code

- Après l'appel à join(), on est sûr que le code du Runnable de otherThread est fini

Exemple de join()

Ici, on est sur que "end" sera affiché APRES "hello1" et "hello2", par contre, l'ordre entre "hello1" et "hello2" n'est pas définie !

```
public static void main(String[] args)
                                        throws InterruptedException {
  Runnable r = () -> {
    System.out.println("hello 1");
  };
  Thread t = new Thread(r);
  t.start();
  System.out.println("hello 2");
  t.join();
  System.out.println("end");
}
```


Cf plus tard

Comment arrêter une thread ?

Une thread s'arrête lorsque la méthode run() du Runnable est fini

Il n'est pas possible de tuer une thread

- Il y a bien une méthode thread.destroy() mais

```
/**
 * Throws {@link NoSuchMethodError}.
 *
 * @deprecated This method was originally designed to destroy this
 *     thread without any cleanup. Any monitors it held would have
 *     remained locked. However, the method was never implemented.
 *     Why are Thread.stop, Thread.suspend and Thread.resume Deprecated!</a>.
 * @throws NoSuchMethodError always
 */
@Deprecated
public void destroy() {
    throw new NoSuchMethodError();
}
```

Extrait de la javadoc de thread.destroy()

Comment arrêter une thread ?

Il existe seulement une façon coopérative

- Si le code d'une thread (le Runnable.run()) execute un appel bloquant
  - Thread.sleep(), Reader.read()/write(), thread.join(), etc.

- Il est possible d'interrompre celle-ci alors qu'elle est en attente sur l'appel bloquant

  ```
  thread.interrupt();
  ```

  dans ce cas, l'appel bloquant lève une exception InterruptedException ou InterruptedIOException

Et si on est pas dans un appel bloquant ?

Dans ce cas, thread.interrupt() positionne un flag indiquant que la thread a été interrompu

Si après on a un appel bloquant

- L'appel n'est pas effectué et lève InterruptedException à la place (le status est repositioné à faux)

Sinon on peut vérifier le status d'interruption

```
Thread.interrupted()
```

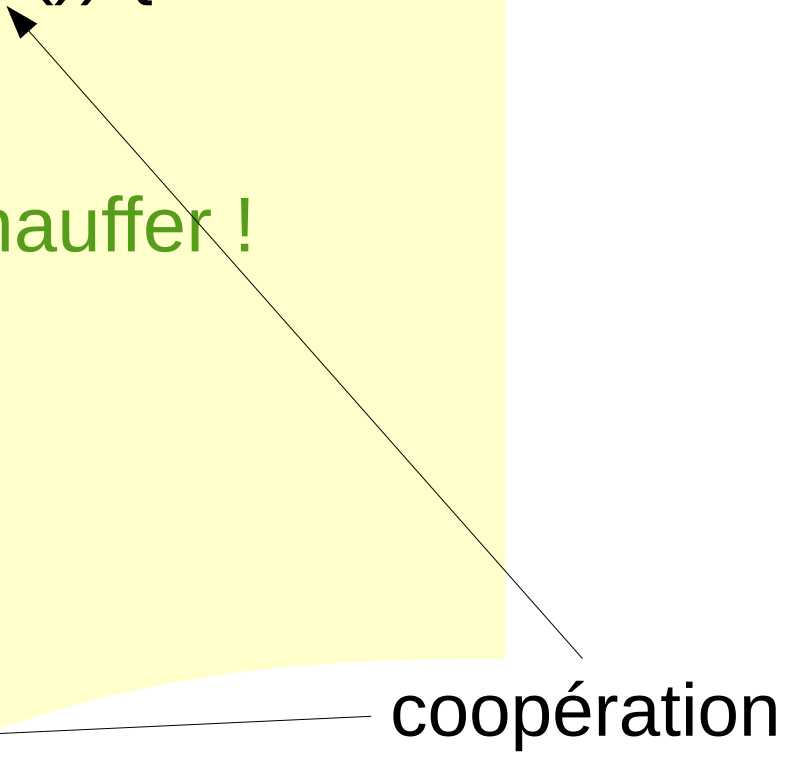
cette méthode devrait s'appeler **interruptedAndClear**() car elle repositionne le statut d'interruption à faux après l'appel

Exemple avec appel bloquant

```
public static void main(String[] args) throws IE {
  Thread t = new Thread(() -> {
    for(;;) {
      try {
        Thread.sleep(1_000);   // laissons le CPU souffler
      } catch(InterruptedException e) {
        return;
      }
    }
  });
  t.start();
  Thread.sleep(1_500);
  t.interrupt();
}
```

Exemple avec appel non bloquant

```
public static void main(String[] args) {
  Runnable r = () -> {
    while(Thread.interrupted()) {
      // pas d'appel bloquant !
      // attente active
      // attention le CPU va chauffer !
    }
  };
  Thread t = new Thread(r);
  t.start();
  Thread.sleep(1_500);
  t.interrupt();
}
```



En résumé

Si la thread fait des appels bloquants alors l'interrompre avec thread.interrupt() est suffisant

Si la thread ne fait pas d'appel bloquant, alors il faut tester avec Thread.interrupted()

Et pourquoi Thread.currentThread().interrupt() ?

Si on a du code après le while, il y a une astuce pour gérér les cas où il y a le status d'interruption posé et où l'exception levée de la même façon

```
public static void main(String[] args) {
  Thread t = new Thread(() -> {
    while(Thread.interrupted()) {
      try {
        // appel bloquant
      } catch(InterruptedException e) {
        Thread.currentThread().interrupt();      <----- WTF
      }
    }
    System.out.println("end");
  });
  t.start();
  Thread.sleep(1_500);
  t.interrupt();
}
```

On se ré-interrompt soit même

Thread daemon et arrêt de la machine virtuelle

La machine virtuelle s'arrête lorsqu'il n'y a plus de thread non-démon qui tourne

```
public static void main(String[] args) {
   Thread t = new Thread(() -> {
      try {
        Thread.sleep(10_000);  // sleep 10s
      } catch(InterruptedException e) {
        throw new AssertionError(e);
      }
   });
   t.setDaemon(true);  // avec cette ligne, la VM meurt rapidement
                       // sinon, on attend 10 s
   t.start();
}
```