Développons en Java

31. Les threads

Chapitre 31

Niveau : Supérieur

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

Sur une machine monoprocesseur, c'est le système d'exploitation qui alloue du temps d'utilisation du CPU pour accomplir les traitements de chaque threads, donnant ainsi l'impression que ces traitements sont réalisés en parallèle.

Sur une machine multiprocesseur, le système d'exploitation peut répartir l'exécution sur plusieurs coeurs, ce qui peut effectivement permettre de réaliser des traitements en parallèle.

Selon le système d'exploitation et l'implémentation de la JVM, les threads peuvent être gérés de deux manières :

- correspondre à un thread natif du système
- correspondre à un thread géré par la machine virtuelle

Dans les deux cas, cela n'a pas d'impact sur le code qui reste le même.

La JVM crée elle-même pour ses propres besoins plusieurs threads : le thread d'exécution de l'application, un ou plusieurs threads pour le ramasse-miettes, ...

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases pour le développement des threads en java.

Le système d'exploitation va devoir répartir du temps de traitement pour chaque thread sur le ou les CPU de la machine. Plus il y a de threads, plus le système va devoir switcher. De plus, un thread requiert des ressources pour s'exécuter notamment un espace mémoire nommé pile. Il est donc nécessaire de contrôler le nombre de threads qui sont lancés dans une même JVM.

Cependant, l'utilisation de plusieurs threads améliore généralement les performances, notamment si la machine possède plusieurs coeurs, car dans ce cas plusieurs threads peuvent vraiment s'exécuter en parallèle. Il est aussi fréquent que les traitements d'un thread soient en attente d'une ressource : le système peut alors plus rapidement allouer du temps CPU à d'autres threads qui ne le sont pas.

L'utilisation de la classe Thread est d'assez bas niveau. A partir de Java 5, le package java.util.concurrency propose des fonctionnalités de plus haut niveau pour faciliter la mise en oeuvre de traitements en parallèle et améliorer les performances de la gestion des accès concurrents.

Ce chapitre contient plusieurs sections :

31.1. L'interface Runnable

Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread.

Cette interface ne définit qu'une seule méthode : void run().

Dans les classes qui implémentent cette interface, la méthode run() doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread.

Exemple :

```
1  package com.jmdoudoux.test;
2
3  public class MonTraitement implements Runnable {
4    public void run() {
5      int i = 0;
6      for (i = 0; i > 10; i++) {
7        System.out.println("" + i);
8      }
9    }
10 }
```

31.2. La classe Thread

La classe Thread est définie dans le package java.lang. Elle implémente l'interface Runnable.

Elle possède plusieurs constructeurs : un constructeur par défaut et plusieurs autres qui peuvent avoir un ou plusieurs des paramètres suivants :

- le nom du thread
- l'objet qui implémente l'interface Runnable l'objet contenant les traitements du thread
- le groupe auquel sera rattaché le thread

Constructeur	Rôle
Thread()	Créer une nouvelle instance
Thread(Runnable target)	Créer une nouvelle instance en précisant les traitements à exécuter
Thread(Runnable target, String name)	Créer une nouvelle instance en précisant les traitements à exécuter et son nom
Thread(String name)	Créer une nouvelle instance en précisant son nom
Thread(ThreadGroup group, Runnable target)	Créer une nouvelle instance en précisant son groupe et les traitements à exécuter
Thread(ThreadGroup group, Runnable target, String name)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter et son nom
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter, son nom et la taille de sa pile
Thread(ThreadGroup group, String name)	Créer une nouvelle instance en précisant son groupe et son nom

Un thread possède une priorité et un nom. Si aucun nom particulier n'est donné dans le constructeur du thread, un nom par défaut composé du préfixe "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement lui est attribué.

La classe Thread possède plusieurs méthodes :

Méthode	Rôle
static int activeCount()	Renvoyer une estimation du nombre de threads actifs dans le groupe du thread courant et ses sous-groupes
void checkAccess()	Déterminer si le thread courant peut modifier le thread
void destroy()	Mettre fin brutalement au thread : ne pas utiliser car deprecated
int countStackFrames()	Deprecated
static Thread currentThread()	Renvoyer l'instance du thread courant
static void dumpStack()	Afficher la stacktrace du thread courant sur la sortie standard d'erreur
static int enumerate(Thread[] tarray)	Copier dans le tableau fourni en paramètre chaque thread actif du groupe et des sous-groupes du thread courant
static Map<Thread,StackTraceElement[]> getAllStackTraces()	Renvoyer une collection de type Map qui contient pour chaque thread actif les éléments de sa stacktrace
int getPriority()	Renvoyer la priorité du thread
ThreadGroup getThreadGroup()	Renvoyer un objet qui encapsule le groupe auquel appartient le thread
static boolean holdsLock(Object obj)	Renvoyer un booléen qui précise si le thread possède le verrou sur le monitor de l'objet passé en paramètre
void interrupt()	Demander l'interruption du thread
static boolean interrupted()	Renvoyer un booléen qui précise si une demande d'interruption du thread a été demandée
boolean isAlive()	Renvoyer un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	Renvoyer un booléen qui indique si le thread a été interrompu
void join()	Attendre la fin de l'exécution du thread
void join(long millis)	Attendre au plus le délai fourni en paramètre que le thread se termine
void join(long millis, int nanos)	Attendre au plus les délai fourni en paramètres (ms + ns) que le thread se termine
void resume()	Reprendre l'exécution du thread préalablement suspendue par suspend( ). Cette méthode est deprecated
void run()	Contenir les traitements à exécuter
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Définir le handler qui sera invoqué si une exception est levée durant l'exécution des traitements
static void sleep(long millis)	Endormir le thread pour le délai exprimé en millisecondes précisé en paramètre
static void sleep(long millis, int nanos)	Endormir le thread pour le délai précisés en paramètres
void start()	Lancer l'exécution des traitements : associer des ressources systèmes pour l'exécution et invoquer la méthode run()
void suspend()	Suspendre le thread jusqu'au moment où il sera relancé par la méthode resume( ). Cette méthode est deprecated
String toString()	Renvoyer une représentation textuelle du thread qui contient son nom, sa priorité et le nom du groupe auquel il appartient
void stop()	Arrêter le thread. Cette méthode est deprecated
static void yield()	Demander au scheduler de laisser la main aux autres threads

31.3. Le cycle de vie d'un thread

Un thread, encapsulé dans une instance de type classe Thread, suit un cycle de vie qui peut prendre différents états.

Le statut du thread est encapsulé dans l'énumération Thread.State

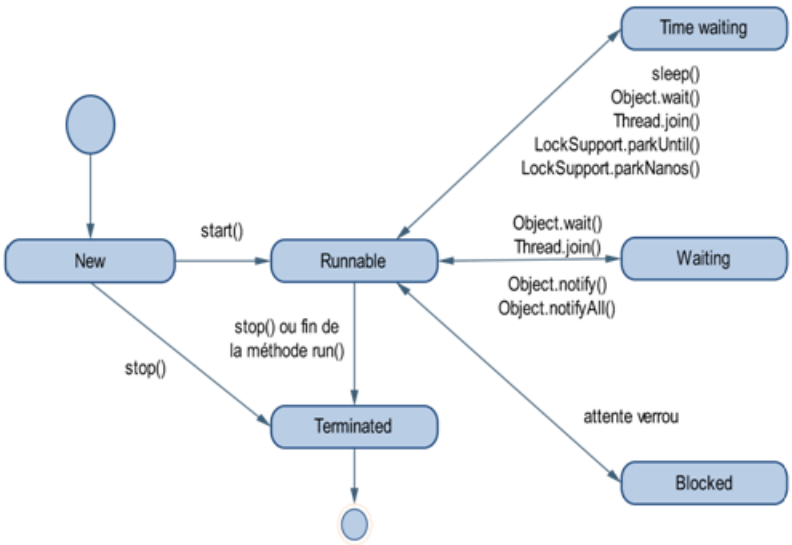
Valeur	Description
NEW	Le thread n'est pas encore démarré. Aucune ressource système ne lui est encore affectée. Seules les méthodes de changement de statut du thread start() et stop() peuvent être invoquées
RUNNABLE	Le thread est en cours d'exécution : sa méthode start() a été invoquée
BLOCKED	Le thread est en attente de l'obtention d'un moniteur qui est déjà détenu par un autre thread
WAITING	Le thread est en attente d'une action d'un autre thread ou que la durée précisée en paramètre de la méthode sleep() soit atteinte. Chaque situation d'attente ne possède qu'une seule condition pour retourner au statut Runnable : • si la méthode sleep() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque le délai précisé en paramètre de la méthode a été atteint • si la méthode suspend() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode resume sera invoquée • si la méthode wait() d'un objet a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode notify() ou notifyAll() de l'objet sera invoquée • si le thread est en attente à cause d'un accès I/O alors le thread ne retournera à l'état Runnable que lorsque cet accès sera terminé
TIMED_WAITING	Le thread est en attente pendent un certain temps d'une action d'un autre thread. Le thread retournera à l'état Runnable lorsque cette action survient ou lorsque le délai d'attente est atteint
TERMINATED	Le thread a terminé son exécution. La fin d'un thread peut survenir de deux manières : • la fin des traitements est atteinte • une exception est levée durant l'exécution de ses traitements

Le statut du thread correspond à celui géré par la JVM : il ne correspond pas au statut du thread sous-jacent dans le système d'exploitation.

Une fois lancé, plusieurs actions peuvent suspendre l'exécution d'un thread :

- invocation de la méthode sleep(), join() ou suspend()
- attente de la fin d'une opération de type I/O
- ...

Le diagramme ci-dessous illustre les différents états d'un thread et les actions qui permettent d'assurer une transition entre ces états.



L'invocation de certaines méthodes de la classe Thread peut lever une exception de type IllegalThreadStateException si cette invocation n'est pas permise à cause de l'état courant du thread.

31.3.1. La création d'un thread

Depuis Java 1.0, il existe plusieurs façons de créer un thread :

- créer une instance d'une classe anonyme de type Thread et implémenter sa méthode run(). Il suffit alors d'invoquer sa méthode start() pour démarrer le thread
- créer une classe fille qui hérite de la classe Thread. Il suffit alors de créer une instance de la classe fille et d'invoquer sa méthode start() pour démarrer le thread
- créer une classe qui implémente l'interface Runnable. Pour lancer l'exécution, il faut créer un nouveau Thread en lui passant en paramètre une instance de la classe et invoquer sa méthode start()
- à partir de Java 8, il est possible d'utiliser une expression lambda pour définir l'implémentation de l'interface Runnable

Il est possible de créer une instance de type Thread dont l'implémentation de la méthode run() va contenir les traitements à exécuter. La classe Thread implémente l'interface Runnable.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThread {
4
5    public static void main(String[] args) {
6      Thread t = new Thread() {
7        public void run() {
8          System.out.println("Mon traitement");
9        }
10     };
11     t.start();
12   }
13 }
```

Il est possible d'hériter de la classe Thread et de redéfinir la méthode run().

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonThread extends Thread {
4
5    @Override
6    public void run() {
7      System.out.println("Mon traitement");
8    }
9  }
```

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThread {
4
5    public static void main(String[] args) {
6      MonThread t = new MonThread();
7      t.start();
8   }
9  }
```

Enfin, il est possible d'implémenter l'interface Runnable. Celle-ci ne définit qu'une seule méthode run() dont l'implémentation doit contenir les traitements à exécuter.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonTraitement implements Runnable {
4
5    @Override
6    public void run(){
7      System.out.println("Mon traitement");
8    }
9  }
```

Pour exécuter les traitements dans un thread, il faut créer une instance de type Thread en invoquant son constructeur avec en paramètre une instance de la classe et invoquer sa méthode start().

Exemple :

```
1  public class TestThread {
2
3    public static void main(String[] args){
4      Thread thread = new Thread(new MonTraitement());
5      thread.start();
6    }
7  }
```

Il est préférable d'utiliser l'implémentation de Runnable car :

- elle permet à la classe d'hériter au besoin d'une classe mère
- elle permet une meilleure séparation des rôles
- elle évite des erreurs car il suffit simplement d'implémenter la méthode run()

Il est possible d'utiliser une instance de type Runnable pour plusieurs threads si l'implémentation est thread-safe.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThread {
4
5    public static void main(String[] args) {
6      Runnable runnable = new MonTraitement();
7
8      for (int i = 0; i < 10; i++) {
9        Thread thread = new Thread(runnable);
10       thread.start();
11     }
12   }
13 }
```

Il ne faut surtout pas invoquer la méthode run() d'un thread. Dans ce cas, les traitements seront exécutés dans le thread courant mais ne seront pas exécutés dans un thread dédié.

31.3.2. L'arrêt d'un thread

Par défaut, l'exécution d'un thread s'arrête pour deux raisons :

- la fin des traitements de la méthode run() est atteinte
- une exception est levée durant les traitements de la méthode run()

Historiquement la classe Thread possède une méthode stop() qui est déclarée deprecated depuis Java 1.1 et est conservée pour des raisons de compatibilité mais elle ne doit pas être utilisée car son comportement peut être aléatoire et inattendu.

La méthode stop() lève une exception de type ThreadDeath se qui interrompt brutalement les traitements du thread. C'est notamment le cas si un moniteur est posé : celui-ci sera libéré mais l'état des données pourrait être inconsistant.

Pour permettre une interruption des traitements d'un thread, il faut écrire du code qui utilise une boucle tant qu'une condition est remplie : le plus simple est d'utiliser un booléen.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonThread extends Thread {
4
5    private volatile boolean running = true;
6
7    public void arreter() {
8      this.running = false;
9    }
10
11   @Override
12   public void run() {
13     while (running) {
14       // traitement du thread
15       try {
16         Thread.sleep(500);
17       } catch (InterruptedException ex) {
18         ex.printStackTrace();
19       }
20     }
21   }
22 }
```

Le Java Memory Model permet à un thread de conserver une copie local de ses champs : pour une exécution correcte, il faut utiliser le mot clé volatile sur le booléen pour garantir que l'accès à la valeur se fera de et vers la mémoire.

Une fois un thread terminé, il passe à l'état terminated. Il ne peut plus être relancé sans lever une exception de type IllegalStateException. Pour le relancer, il faut créer une nouvelle instance.

31.4. Les démons (daemon threads)

Il existe deux catégories de threads :

- thread utilisateur (user thread)
- démon (daemon thread)

Un thread démon n'empêche pas la JVM de s'arrêter même s'il est encore en cours d'exécution. Une application dans laquelle les seuls threads actifs sont des démons est automatiquement fermée.

Généralement, les traitements d'un thread démon s'exécutent indéfiniment et ils ne sont pas interrompus : c'est l'arrêt de la JVM qui provoque leur fin. Lorsque la JVM s'arrête, elle termine tous les threads démons en cours d'exécution du moment qu'ils soient les seuls encore actifs.

Par exemple, les threads du ramasse-miettes sont généralement des démons.

Par défaut, un nouveau thread hérite de la propriété daemon du thread qui le lance.

Pour préciser qu'un thread est un démon, il faut invoquer sa méthode setDaemon() en lui passant la valeur true comme paramètre. Cette méthode doit être invoquée avant que le thread ne soit démarré : une fois le thread démarré, son invocation lève une exception de type IllegalThreadStateException.

La méthode isDaemon() renvoie un booléen qui précise si le thread est un démon.

Lorsque la JVM s'arrête, les threads démons sont arrêtés brutalement : leurs blocs finally ne sont pas exécutés. C'est la raison pour laquelle, les threads démons ne devraient pas être utilisés pour réaliser des opérations de type I/O ou des

traitements critiques.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreaddemon {
4
5    public static void main(String[] args) {
6      Thread daemonThread = new Thread(new Runnable() {
7        @Override
8        public void run() {
9          try {
10           while (true) {
11             System.out.println("Execution demon");
12           }
13         } finally {
14           System.out.println("Fin demon");
15         }
16       }
17     }, "Demon");
18
19     daemonThread.setDaemon(true);
20     daemonThread.start();
21   }
22 }
```

Le nombre de messages affichés varie de un à quelques uns avant l'arrêt de la JVM. Le message du bloc finally n'est jamais affiché.

31.5. Les groupes de threads

Un groupe de threads permet de regrouper des threads selon différents critères et de les manipuler en même temps ce qui évite d'avoir à effectuer la même opération individuellement sur tous les threads. Il permet aussi de définir des caractéristiques communes aux nouveaux threads qui lui sont ajoutés.

La notion de groupe permet aussi de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés.

La classe java.lang.ThreadGroup encapsule un groupe de threads : elle contient un ensemble de threads pour permettre de réaliser des opérations de gestion ou de contrôle sur tous ceux-ci. Elle peut aussi contenir d'autres ThreadGroups qui forment alors des sous-groupes. Cela permet de créer une hiérarchie dans les groupes.

Chaque groupe, à l'exception du groupe par défaut, possède un groupe parent. Chaque thread appartient à un groupe de threads (thread group) :

- soit explicitement dans un groupe de threads précisé en paramètre de l'une des surcharges du constructeur de la classe thread : Thread(ThreadGroup group, Runnable runnable), Thread(ThreadGroup group, String name), Thread(ThreadGroup group, Runnable runnable, String name)
- soit dans un groupe de threads par défaut si aucun n'est précisé. Par défaut, lors de la création d'un thread, si aucun groupe n'est précisé alors c'est le groupe du thread courant qui est utilisé.

Il existe un groupe de thread par défaut. Au lancement de la JVM, un ThreadGroup généralement nommé main est créé et sera utilisé comme groupe de threads par défaut.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
5    public static void main(String[] args) {
6      Runnable runnable = new MonTraitement();
7      Thread t = new Thread(runnable);
8      System.out.println("groupe:"+t.getThreadGroup().getName());
```

```
9       t.start();
10    }
11  }
```

Résultat :

```
1  groupe:main
```

La seule solution pour ajouter un Thread dans un groupe particulier est d'utiliser une des surcharges du constructeur de la classe Thread qui attend en paramètre un objet de type ThreadGroup :

- public Thread(ThreadGroup group, Runnable target)
- public Thread(ThreadGroup group, String name)
- public Thread(ThreadGroup group, Runnable target, String name)

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
5    public static void main(String[] args) {
6      Runnable runnable = new MonTraitement();
7      ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
8      Thread t = new Thread(monThreadGroup, runnable);
9      System.out.println("groupe:" + t.getThreadGroup().getName());
10     t.start();
11   }
12 }
```

Résultat :

```
1  groupe:Mon groupe de threads
```

Attention : une fois créé, un thread ne peut pas être déplacé vers un autre groupe.

La classe ThreadGroup possède plusieurs méthodes :

Méthode	Rôle
int activeCount()	Renvoyer une estimation du nombre de threads actifs dans le groupe et ses sous-groupes
int activeGroupCount()	Renvoyer une estimation du nombre de groupes actifs en incluant les sous-groupes
boolean allowThreadSuspension(boolean b)	Deprecated
void checkAccess()	Vérifier si le thread courant possède les permissions pour modifier son groupe
void destroy()	Détruire le groupe de threads et ses sous-groupes
int enumerate(Thread[] list) int enumerate(Thread[] list, boolean recurse)	Copier dans le tableau fourni en paramètre l'ensemble des threads actifs du groupe de threads et de ses sous-groupes
int enumerate(ThreadGroup[] list) int enumerate(ThreadGroup[] list, boolean recurse)	Copier dans le tableau fourni en paramètre l'ensemble des sous-groupes actifs
int getMaxPriority()	Renvoyer la priorité maximale du groupe
String getName()	Renvoyer le nom du groupe
ThreadGroup getParent()	Renvoyer le groupe parent
void interrupt()	Demander l'interruption de tous les threads du groupe
boolean isDaemon()	Renvoyer un booléen qui précise si le groupe est un démon. Lorsqu'un groupe est un démon, il sera détruit lorsque tous ses threads seront terminés
boolean isDestroyed()	Renvoyer un booléen qui précise si le groupe est détruit
void list()	Afficher des informations sur le groupe sur la sortie standard
boolean parentOf(ThreadGroup g)	Renvoyer un booléen qui précise si le groupe courant est le même que celui fourni en argument ou est d'un groupe parent
void resume()	Deprecated
void setDaemon(boolean daemon)	Préciser si le groupe est un démon ou non
void setMaxPriority(int pri)	Préciser la priorité maximale du groupe
void stop()	Deprecated
void suspend()	Deprecated
void uncaughtException(Thread t, Throwable e)	Cette méthode est invoquée par la JVM si un thread du groupe sans UncaughtExceptionHandler lève une exception durant son exécution

La classe ThreadGroup possède plusieurs propriétés :

- maxPriority définit la valeur maximale de la priorité des threads inclus dans le groupe
- name définit le nom du groupe. Il n'est possible de le modifier une fois le groupe créé
- daemon indique si le groupe est un démon. Si tel est le cas, le groupe sera détruit lorsque tous les threads qu'il contient seront terminés
- parent contient le groupe parent

La méthode setMaxPriority() permet de définir la priorité maximale des threads qui lui seront ajoutés et de ses sous-groupes.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
```

```
5     public static void main(String[] args) {
6       Runnable runnable = new MonTraitement();
7       ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
8       monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
9       Thread t = new Thread(monThreadGroup, runnable);
10      t.setPriority(Thread.MAX_PRIORITY);
11      monThreadGroup.list();
12      System.out.println("thread.priority=" + t.getPriority());
13      t.start();
14    }
15  }
```

Résultat :

```
1  java.lang.ThreadGroup[name=Mongroupe de threads,maxpri=5]
2  thread.priority=5
```

Une modification d'une de ces propriétés n'a pas d'impact sur les threads contenus dans le groupe.

Par exemple, lors de l'utilisation de la méthode setMaxPriority(), seule la propriété MaxPriority du groupe est modifiée. La priorité des threads déjà inclus dans le groupe n'est pas modifiée. La nouvelle valeur n'aura un impact que sur les prochains threads ajoutés au groupe.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
5    public static void main(String[] args) {
6      Runnable runnable = new MonTraitement();
7      ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
8      Thread t = new Thread(monThreadGroup, runnable);
9      t.setPriority(Thread.MAX_PRIORITY);
10     monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
11     monThreadGroup.list();
12     System.out.println("thread.proprity=" + t.getPriority());
13     t.start();
14   }
15 }
```

Résultat :

```
1  java.lang.ThreadGroup[name=Mongroupe de threads,maxpri=5]
2  thread.proprity=10
```

Il est donc possible qu'un thread appartenant à un groupe ait une priorité supérieure à la priorité maximale définit dans son groupe.

La méthode setDaemon() n'a aucune influence sur les threads contenus dans le groupe. Il est tout à fait possible d'ajouter des threads utilisateurs ou démon à un groupe dont la propriété daemon est true.

La méthode isDestroy() permet de savoir si le groupe est détruit.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
```

```
5     public static void main(String[] args) throws InterruptedException {
6       Runnable runnable = new MonTraitement();
7       ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
8       monThreadGroup.setDaemon(true);
9       System.out.println("groupe.isDaemon()=" + monThreadGroup.isDaemon());
10      Thread t = new Thread(monThreadGroup, runnable);
11      System.out.println("thread.isDaemon()=" + t.isDaemon());
12      monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
13      t.start();
14      t.join();
15      System.out.println("groupe.isDestroy()=" + monThreadGroup.isDestroyed());
16    }
17  }
```

Résultat :

```
1  groupe.isDaemon()=true
2  thread.isDaemon()=false
3  Mon traitement
4  Thread-0
5  groupe.isDestroy()=true
```

Une fois qu'un groupe est détruit, il n'est plus possible de lui ajouter un thread sinon une exception de type IllegalThreadStateException est levée.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
5    public static void main(String[] args) throws InterruptedException {
6      Runnable runnable = new MonTraitement();
7      ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
8      monThreadGroup.setDaemon(true);
9      System.out.println("groupe.isDaemon()=" + monThreadGroup.isDaemon());
10     Thread t = new Thread(monThreadGroup, runnable);
11     System.out.println("thread.isDaemon()=" + t.isDaemon());
12     monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
13     t.start();
14     t.join();
15     System.out.println("groupe.isDestroy()=" + monThreadGroup.isDestroyed());
16     t = new Thread(monThreadGroup, runnable);
17   }
18 }
```

Résultat :

```
1  groupe.isDaemon()=true
2  thread.isDaemon()=false
3  Montraitement Thread-0
4  groupe.isDestroy()=true
5  Exception in thread "main" java.lang.IllegalThreadStateException
6              at java.lang.ThreadGroup.addUnstarted(ThreadGroup.java:843)
7              at java.lang.Thread.init(Thread.java:348)
8              at java.lang.Thread.<init>(Thread.java:451)
9              at com.jmdoudoux.dej.thread.TestThreadGroup.main(TestThreadGroup.java:19)
```

La méthode parentOf() renvoie un booléen qui précise si le groupe est un parent du groupe passé en paramètre.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4    public static void main(String[] args) throws InterruptedException {
5      ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
6      ThreadGroup monSousThreadGroup = new ThreadGroup(monThreadGroup,
7          "Mon sous-groupe de threads");
8      System.out.println(monThreadGroup.parentOf(monSousThreadGroup));
9    }
10 }
```

Résultat :

```
1  true
```

Les méthodes resume(), stop() et suspend() qui permettaient d'interagir sur l'état des threads du groupe sont deprecated depuis Java 1.1.

Les méthodes activeCount() et enumerate() sont généralement utilisées ensemble pour obtenir la liste des threads actifs dans le groupe et ses sous-groupes.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadGroup {
4
5    public static void main(String[] args) throws InterruptedException {
6      int nbThreads;
7      Thread[] threads;
8      Runnable runnable = new MonTraitement();
9      ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
10     Thread t = new Thread(monThreadGroup, runnable, "thread groupe 1");
11     t.start();
12     t = new Thread(monThreadGroup, runnable, "thread groupe 2");
13     t.start();
14     ThreadGroup monSousThreadGroup = new ThreadGroup(monThreadGroup,
15         "Mon sous-groupe de threads");
16     t = new Thread(monSousThreadGroup, runnable, "thread sous groupe 1");
17     t.start();
18     nbThreads = monThreadGroup.activeCount();
19     System.out.println("groupe.activeCount()=" + nbThreads);
20     threads = new Thread[nbThreads];
21     monThreadGroup.enumerate(threads);
22     for (int i = 0; i < nbThreads; i++) {
23       System.out.println("Thread " + i + " = " + threads[i].getName());
24     }
25   }
26 }
```

Résultat :

```
1  Mon traitement thread groupe 1
2  Mon traitement thread groupe 2
3  groupe.activeCount()=3
4  Mon traitement thread sous groupe 1
5  Thread 0 = thread groupe 1
```

```
Thread 1 = thread groupe 2
Thread 2 = thread sous groupe 1
```

La classe Thread possède plusieurs méthodes relatives au groupe du thread :

- la méthode getThreadGroup() renvoie une instance de type ThreadGroup qui encapsule le groupe auquel appartient le thread
- la méthode static activeCount() renvoie un entier qui correspond à une estimation du nombre de threads actifs appartenant au même groupe et sous-groupes du thread courant

31.6. L'obtention d'informations sur un thread

Plusieurs méthodes de la classe Thread permettent d'obtenir des informations sur le thread.

Méthode	Rôle
boolean isDaemon()	Renvoyer un booléen qui précise si le thread est un démon
long getId()	Renvoyer un entier long dont la valeur est l'identifiant du thread
ClassLoader getContextClassLoader()	Renvoyer le context classloader du thread
StackTraceElement[] getStackTrace()	Renvoyer un tableau des éléments qui composent la stacktrace d'exécution du thread
int getPriority()	Renvoyer la priorité du thread

Chaque thread possède un nom. Par défaut, la JVM attribut un nom composé de Thread- suivi d'un numéro incrémenté. La méthode getName() permet d'obtenir le nom du thread.

Pour aider au débogage et dans les logs, il est intéressant de donner un nom plus explicite à chaque thread pour l'identifier facilement. Le nom peut être fourni en paramètre du constructeur de l'instance de type Thread ou en utilisant la méthode setName() qui permet de donner un nom explicit au thread.

31.6.1. L'état d'un thread

Plusieurs méthodes permettent d'obtenir des informations sur l'état d'un thread.

Méthode	Rôle
boolean isAlive()	Renvoyer un booléen qui précise si le thread est en cours d'exécution. Elle renvoie true tant que le thread a été démarré et qu'il n'est pas arrêté
Thread.State getState()	Renvoyer le statut du thread
boolean isInterrupted()	Renvoyer un booléen qui précise si le thread est interrompu

31.6.2. L'obtention du thread courant

La méthode statique currentThread() permet d'obtenir le thread dans lequel le code s'exécute.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonTraitement implements Runnable {
4    public void run() {
5      System.out.println("Mon traitement " + Thread.currentThread().getName());
```

```
  }
}
```

Exemple :

```
package com.jmdoudoux.dej.thread;

public class TestThread {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();

    for (int i = 0; i < 10; i++) {
      Thread thread = new Thread(runnable);
      thread.setName("monTraitement-" + i);
      thread.start();
    }
  }
}
```

31.7. La manipulation des threads

Il est possible de réaliser plusieurs opérations sur un thread :

- modifier la priorité d'exécution du thread
- mettre en sommeil le thread pour une certaine durée (en millisecondes)
- attendre la fin de l'exécution d'un autre thread
- mettre en pause le thread pour laisser aux autres threads plus de chance de s'exécuter
- interrompre le thread

31.7.1. La mise en sommeil d'un thread pour une certaine durée

La méthode static sleep() de la classe Thread permet de mettre en sommeil le thread courant pour le délai en millisecondes dont la valeur est fournie en paramètre.

Elle est bloquante, elle lève une exception de type InterruptedException au cours de son exécution si un autre thread demande l'interruption de l'exécution du thread.

Exemple :

```
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

La méthode sleep() est static : elle ne s'applique que sur le thread courant et il n'est pas possible de désigner le thread concerné.

Une surcharge de la méthode sleep() attend en paramètre la durée en millisecondes et une durée supplémentaire en nanosecondes qui peut varier entre 0 et 999999. La précision de cette attente supplémentaire est dépendante de la machine et du système d'exploitation.

Contrairement à la méthode wait() de la classe Object, la méthode sleep() ne libère pas les verrous qui sont posés par le thread.

31.7.2. L'attente de la fin de l'exécution d'un thread

La méthode join() de la classe Thread permet d'attendre la fin de l'exécution du thread. Elle peut lever une exception de type InterruptedException.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  import java.text.DateFormat;
4  import java.text.SimpleDateFormat;
5  import java.util.Date;
6
7  public class TestThreadJoin {
8
9    public static void main(String[] args) {
10     DateFormat df = new SimpleDateFormat("HH:mm:ss");
11     Thread thread1 = new Thread(new MonRunnable(10000));
12     Thread thread2 = new Thread(new MonRunnable(5000));
13
14     System.out.println(df.format(new Date()) + " debut");
15
16     thread1.start();
17     thread2.start();
18
19     try {
20       thread2.join();
21     } catch (InterruptedException e) {
22       e.printStackTrace();
23     }
24
25     System.out.println(df.format(new Date()) + " fin thread2");
26
27     try {
28       thread1.join();
29     } catch (InterruptedException e) {
30       e.printStackTrace();
31     }
32
33     System.out.println(df.format(new Date()) + " fin");
34
35   }
36
37   private static class MonRunnable implements Runnable {
38
39     private long delai;
40
41     public MonRunnable(long delai) {
42       this.delai = delai;
43     }
44
45     @Override
46     public void run() {
47       try {
48         Thread.sleep(delai);
49       } catch (InterruptedException e) {
50         e.printStackTrace();
51       }
52     }
53   }
54 }
```

Résultat :

```
19:57:04 debut
19:57:09 fin thread2
19:57:14 fin
```

Une surcharge de la méthode join() attend en paramètre un entier long qui définit la valeur en millisecondes d'un délai d'attente maximum.

Exemple :

```java
package com.jmdoudoux.dej.thread;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestThreadJoin {

  public static void main(String[] args) {
    DateFormat df = new SimpleDateFormat("HH:mm:ss");
    Thread thread1 = new Thread(new MonRunnable(10000));
    Thread thread2 = new Thread(new MonRunnable(5000));

    System.out.println(df.format(new Date()) + " debut");

    thread1.start();
    thread2.start();

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

    System.out.println(df.format(new Date()) + " fin thread2");

    try {
      thread1.join(1000);

      System.out.println("thread1 en cours d'execution : " + thread1.isAlive());

    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println(df.format(new Date()) + " fin");
  }

  private static class MonRunnable implements Runnable {

    private long delai;

    public MonRunnable(long delai) {
      this.delai = delai;
    }

    @Override
    public void run() {
      try {
        Thread.sleep(delai);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

```
54      }
55    }
56  }
```

Résultat :

```
20:01:04 debut
20:01:09 fin thread2
thread1 en cours d'execution : true
20:01:10 fin
```

31.7.3. La modification de la priorité d'un thread

Un thread possède une propriété qui précise sa priorité d'exécution. Pour déterminer ou modifier la priorité d'un thread, la classe Thread contient les méthodes suivantes :

Méthode	Rôle
int getPriority()	retourner la priorité d'exécution du thread
void setPriority(int)	modifier la priorité d'exécution du thread

Généralement, la priorité varie de 1 à 10 mais cela dépend de l'implémentation de la JVM. Plusieurs constantes permettent de connaître les valeurs de la plage de priorités utilisables et la valeur de la priorité par défaut :

- Thread.MIN_PRIORITY : la valeur de la priorité minimale
- Thread.MAX_PRIORITY : la valeur de la priorité maximale
- Thread.NORM_PRIORITY : la valeur de la priorité normale

Exemple :

```
package com.jmdoudoux.dej.thread;

public class TestThreadPriority {

  public static void main(String[] args) {
    System.out.println("MIN_PRIORITY : " + Thread.MIN_PRIORITY);
    System.out.println("MAX_PRIORITY : " + Thread.MAX_PRIORITY);
    System.out.println("NORM_PRIORITY : " + Thread.NORM_PRIORITY);
  }
}
```

Résultat :

```
MIN_PRIORITY : 1
MAX_PRIORITY : 10
NORM_PRIORITY : 5
```

La valeur par défaut de la priorité lors de la création d'un nouveau thread est celle du thread courant.

La méthode setPriority() lève une exception de type IllegalStateException si la valeur fournie en paramètre n'est pas incluse dans la plage Thread.MIN_PRIORITY et Thread.MAX_PRIORITY.

Exemple :

```
    Thread thread = new Thread();
    thread.setPriority(Thread.MAX_PRIORITY);
```

Attention : il n'y a aucune garantie sur le résultat du changement de la priorité d'un thread. La gestion des priorités est dépendante de l'implémentation de la JVM et/ou du système d'exploitation sous-jacent. Sur des machines de type Mac ou Unix, le thread qui a la plus grande priorité a systématiquement accès au processeur s'il ne se trouve pas en mode " en attente ". Sous Windows 95, le système ne gère pas correctement les priorités et il choisit lui-même le thread à exécuter : l'attribution d'une priorité supérieure permet simplement d'augmenter ses chances d'exécution.

31.7.4. Laisser aux autres threads plus de chance de s'exécuter

La méthode static yield() de la classe Thread tente de mettre en pause le thread courant pour laisser une chance aux autres threads de s'exécuter.

Attention : il n'y a aucune garantie sur le résultat de l'invocation de la méthode yield() car elle est dépendante de l'implémentation de la JVM.

31.7.5. L'interruption d'un thread

Si le thread n'est pas correctement codé, il n'est pas possible de forcer son arrêt.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestInterruptThread {
4
5    public static void main(String[] args) throws InterruptedException {
6      System.out.println("debut");
7      Thread thread = new Thread(new Runnable() {
8        boolean encore = true;
9
10       @Override
11       public void run() {
12         System.out.println("debut thread");
13         long i = 0;
14         while (encore) {
15           // très mauvais exemple qui simule une activité du thread
16           // Ne pas oublier d'interrompre l'exécution
17           i++;
18           i--;
19         }
20         System.out.println("i=" + i);
21         System.out.println("fin thread");
22       }
23     });
24
25     thread.start();
26
27     try {
28       Thread.sleep(100);
29     } catch (InterruptedException e) {
30       e.printStackTrace();
31     }
32
33     thread.interrupt();
34     thread.join();
35     System.out.println("fin");
36   }
37 }
```

Résultat :

```
debut
debut thread
```

Dans cette situation la seule solution pour arrêter le thread est d'arrêter la JVM elle-même.

Une solution pour remplacer l'invocation de la méthode stop(), qui est deprecated, est d'invoquer la méthode interrupt() pour demander l'interruption de l'exécution d'un thread. Il est nécessaire dans ce cas de s'assurer que les traitements du thread vont tenir compte du statut interrupted du thread pour s'interrompre.

Rien ne définit une sémantique claire à propos de l'utilisation de l'interruption d'un thread mais généralement lorsqu'elle est prise en compte cela se traduit par une fin de l'exécution des traitements effectués le plus proprement possible par le thread lui-même.

Le demande d'interruption n'a pas l'obligation a été prise en compte immédiatement. Généralement, si les traitements sont faits dans une boucle, celle-ci vérifie à chaque itération le statut interrupted. Selon le temps de traitements d'une itération, le délai entre deux vérifications peut être plus ou moins long.

Un thread possède une propriété booléenne qui indique si le statut du thread est interrompu (interrupted). Sa valeur par défaut est false. Lors de l'invocation de la méthode interrupt(), la valeur de la propriété passe à true.

La méthode isInterrupted() permet de renvoyer un booléen qui indique la valeur du statut interrupted du thread.

La méthode interrupted() permet de renvoyer un booléen qui indique la valeur du statut interrupted du thread et de réinitialiser sa valeur. Attention : l'invocation de la méthode interrupted() réinitialise le statut interrupted du thread. Une seconde invocation consécutive de cette méthode renverra toujours false.

L'interruption d'un thread en Java requiert une collaboration entre le thread qui demande l'interruption et le thread dont l'interruption est demandée. Une demande d'interruption ne doit pas nécessairement mettre fin immédiatement au traitement du thread : c'est une demande polie d'un autre thread qui lui demande de bien vouloir mettre fin à son exécution à sa convenance.

L'intérêt de l'interruption de manière coopérative est qu'elle permet de mettre en place un mécanisme souple pour annuler l'exécution de tâches.

Il est rare de vouloir qu'un traitement s'arrête de manière brutale et immédiate : il y a généralement dans ce cas un risque de laisser les données dans un état incohérent. La fin prématurée de l'exécution d'une tâche doit être réalisée par la tâche elle-même pour lui permettre de se terminer proprement par exemple en libérant des ressources.

Exemple :

```
package com.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        long i = 0;
        while (!Thread.currentThread().isInterrupted()) {
          // tres mauvais exemple qui simule une activité du thread
          // sur un temps heureusement très court
          i++;
          i--;
        }
        System.out.println("i=" + i);
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
```

```
29      }
30
31      thread.interrupt();
32      thread.join();
33      System.out.println("fin");
34    }
35  }
```

Résultat :

```
1  debut
2  debut thread
3  i=0
4  fin thread
5  fin
```

Lorsque le statut interrupted d'un thread est passé à true et qu'une méthode bloquante (Thread.sleep(), Thread.join(), Object.wait(), ...) est en cours d'exécution alors une exception de type InterruptedException est levée par cette méthode. Lorsque l'exception InterruptException est levée, le statut interrupted du thread est retiré : la méthode isInterrupted() renvoie false.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestInterruptThread {
4
5    public static void main(String[] args) throws InterruptedException {
6      System.out.println("debut");
7      Thread thread = new Thread(new Runnable() {
8        @Override
9        public void run() {
10         System.out.println("debut thread");
11         try {
12           Thread.sleep(5000);
13         } catch (InterruptedException e) {
14           System.out.println("Le thread est interrompu");
15           System.out.println("thread.isInterrupted()="
16               + Thread.currentThread().isInterrupted());
17         }
18         System.out.println("fin thread");
19       }
20     });
21
22     thread.start();
23
24     try {
25       Thread.sleep(1000);
26     } catch (InterruptedException e) {
27       e.printStackTrace();
28     }
29
30     thread.interrupt();
31     thread.join();
32     System.out.println("fin");
33   }
34 }
```

Résultat :

```
debut
debut thread
Le thread est interrompu
thread.isInterrupted()=false
fin thread
fin
```

Ainsi, il est possible qu'un thread ne s'arrête jamais.

Exemple :

```
package com.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        while (!Thread.currentThread().isInterrupted()) {
          try {
            Thread.sleep(500);
            System.out.println("traitement du thread");
          } catch (InterruptedException e) {
            System.out.println("InterruptedException capturee");
            System.out.println("thread.isInterrupted()="
                + Thread.currentThread().isInterrupted());
          }
        }
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
InterruptedException capturee
thread.isInterrupted()=false
traitement du thread
traitement du thread
traitement du thread
traitement du thread
```

Une bonne pratique pour ne pas perdre le statut est de le remettre dans la clause catch de l'exception en invoquant la méthode interrupt() du thread courant.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestInterruptThread {
4
5    public static void main(String[] args) throws InterruptedException {
6      System.out.println("debut");
7      Thread thread = new Thread(new Runnable() {
8        @Override
9        public void run() {
10         System.out.println("debut thread");
11         while (!Thread.currentThread().isInterrupted()) {
12           try {
13             Thread.sleep(500);
14             System.out.println("traitement du thread");
15           } catch (InterruptedException e) {
16             System.out.println("InterruptedException capturee");
17             Thread.currentThread().interrupt();
18           }
19         }
20         System.out.println("fin thread");
21       }
22     });
23
24     thread.start();
25
26     try {
27       Thread.sleep(100);
28     } catch (InterruptedException e) {
29       e.printStackTrace();
30     }
31
32     thread.interrupt();
33     thread.join();
34     System.out.println("fin");
35   }
36 }
```

Résultat :

```
1  debut
2  debut thread
3  InterruptedException capturee
4  fin thread
5  fin
```

L'invocation de la méthode interrupt() va remettre le statut interrupted du thread à true pour permettre une sortie de la boucle.

31.7.6. L'exception InterruptedException

La fin de l'exécution d'une méthode ordinaire dépend de la quantité de traitements à exécuter et des ressources disponibles pour le faire (CPU et mémoire). La fin de l'exécution d'une méthode bloquante est aussi dépendante d'un événement extérieur tel qu'un timeout, la libération d'un verrou, ... Ceci rend le temps d'attente difficilement prédictible voire même infini si l'événement ne survient jamais. Ce dernier cas de figure entraine un blocage infini du traitement : il est donc nécessaire d'avoir un mécanisme qui permette de sortir de cette situation.

Ce sont des méthodes bloquantes qui peuvent lever une exception de type InterruptedException. De nombreuses méthodes de classes du JDK couramment utilisées peuvent lever une exception de type InterruptedException :

- Object.wait()
- Thread.sleep(), Thread.join()
- Process.waitFor()
- SwingUtilities.invokeAndWait()
- de nombreuses méthodes des classes du package java.util.concurrent telles que Future.get(), BlockingQueue.take(), ExecutorService. awaitTermination(), ...
- ...

Une exception de type InterruptedException est levée par une méthode bloquante pour indiquer que la méthode interrupt() du thread courant a été invoquée par un autre thread, signifiant ainsi que cet autre thread demande au thread courant de s'interrompre.

Les méthodes bloquantes prennent en compte les demandes d'interruption en levant une exception de type InterruptedException. Une méthode ordinaire n'a pas l'obligation de faire de même mais si son temps de traitement peut être long, il est utile et pratique de périodiquement vérifier le statut interrupted du thread et de lever une exception de type InterruptedException.

Remarque : toutes les méthodes bloquantes ne lèvent pas d'exception de type InterruptedException : c'est par exemple le cas des méthodes des classes InputStream et OutputStream qui peuvent attendre la fin d'une opération de type I/O mais ne lève pas cette exception et ne s'arrête pas si le thread courant est interrompu.

L'obtention d'un verrou sur un moniteur en utilisant le mot clé synchronized ne peut pas être interrompue bien qu'étant bloquante.

Si l'exception InterruptedException n'était pas une exception de type checked, probablement personne ne prendrait en compte sa gestion. Comme celle-ci est obligatoire, elle consiste généralement à ne rien faire ou à simplement afficher un message dans un log. Cependant ignorer une exception de type InterruptedException est rarement une bonne idée car cette pratique fait perdre l'information qu'une demande d'interruption du thread a été faite.

Lorsqu'une méthode bloquante lève une exception de type InterruptedException, elle informe le thread courant qu'un autre thread vient de tenter de l'interrompre. Une prise en compte, adaptée au contexte, de cette exception est nécessaire pour assurer une meilleure réactivité de l'application.

Il est fréquent de rencontrer ou d'écrire du code qui intercepte une exception de type InterruptedException avec un bloc de code vide ou simplement journaliser l'exception avec un niveau de gravité plus ou moins important. Capturer une exception et l'ignorer n'est pas une bonne pratique. Se contenter de l'ajouter dans un journal revient aussi à l'ignorer, si ce n'est qu'il y a en une petite trace.

L'arrêt d'un thread en Java doit être coopératif entre le thread qui en fait la demande généralement en positionnant un booléen et les traitements du thread qui doivent périodiquement vérifier la valeur du booléen avant de poursuivre les traitements.

Il est possible d'utiliser la propriété booléenne interrupted du thread. Pour basculer la valeur de la propriété, il faut invoquer la méthode interrupt() du thread.

La méthode interrupt() n'interrompt pas l'exécution du thread : elle positionne simplement le statut interrupted du thread à true. Les traitements du thread ont la charge de tenir compte de ce statut et de faire les actions appropriées sachant qu'il n'existe pas de recommandations sur celles-ci.

Si le thread en cours exécute un traitement bloquant, alors une exception de type InterruptedException est levée.

Comme précisé dans la javadoc des méthodes concernées, lorsqu'une exception de type InterruptedException est levée, le statut interrupted du thread est réinitialisé.

31.8. Les messages de synchronisation entre threads

La classe Object contient les méthodes wait(), notify() et notifyAll() pour permettre de synchroniser des threads grâce à l'envoi de messages. Ces méthodes permettent la mise en oeuvre d'un mécanisme de communication par échanges de messages visant à synchroniser l'exécution de threads.

La méthode wait() met le thread courant en attente jusqu'à ce que l'objet reçoive une notification par les méthodes notify() ou notifyAll() : cette attente peut donc être potentiellement infinie.

La méthode wait() possède deux surcharges :

- wait(long *timeout*) : attend au plus la durée en millisecondes fournie en paramètre
- wait(long *timeout*, int *nanos*) : attend au plus la durée en millisecondes cumulée avec celle en nanosecondes fournies en paramètres

La méthode notifyAll() avertit tous les threads dont les méthodes wait() de la même instance sont invoquées.

La méthode notify() avertit un des threads dont la méthode wait() de la même instance est invoquée.

Il est important que les méthodes wait() et notifyAll() ne soient invoquées que par le thread qui possède le verrou sur le moniteur de l'instance.

Un cas classique d'utilisation de la synchronisation de threads est la mise en oeuvre du modèle de conception producer/consumer.

Dans l'exemple ci-dessous, un thread (producer) est utilisé pour produire des données qui sont consommées par un autre thread (consumer). Un objet partagé par les deux threads permet de stocker une valeur et de gérer sont accès par les threads en les synchronisant.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MaQueue {
4    private String  valeur;
5    private boolean disponible = false;
6
7    public synchronized String get() throws InterruptedException {
8      while (disponible == false) {
9        wait();
10     }
11     disponible = false;
12     notifyAll();
13     return valeur;
14   }
15
16   public synchronized void put(String valeur) throws InterruptedException {
17     while (disponible == true) {
18       wait();
19     }
20     this.valeur = valeur;
21     disponible = true;
22     notifyAll();
23   }
24 }
```

Les méthodes wait(), notify() et notifyAll() doivent être invoquées dans un bloc de code synchronized utilisant l'objet lui-même comme moniteur pour éviter de lever une exception de type IllegalMonitorStateException. Il peut y avoir une race condition lors de l'invocation des méthodes wait() and notify() si elles ne sont pas invoquées dans un bloc de code synchronized. Le moniteur de ce bloc synchronized doit obligatoirement être l'instance sur laquelle les méthodes wait(), notify() et notifyAll() vont être invoquées.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonProducer extends Thread {
4    private MaQueue maQueue;
5
6    public MonProducer(MaQueue maQueue) {
7      this.maQueue = maQueue;
8    }
9
10   public void run() {
11     int i = 0;
12     while (!Thread.currentThread().isInterrupted()) {
13       try {
14         i++;
15         maQueue.put("valeur-" + i);
16         System.out.println("Producer put : " + i);
17         sleep((int) (Math.random() * 1000));
18       } catch (InterruptedException e) {
```

```
        Thread.currentThread().interrupt();
        break;
      }
    }
  }
}
```

Exemple :

```
package com.jmdoudoux.dej.thread;

public class MonConsumer extends Thread {
  private MaQueue maQueue;

  public MonConsumer(MaQueue maQueue) {
    this.maQueue = maQueue;
  }

  public void run() {
    String value = null;
    while (!Thread.currentThread().isInterrupted()) {
      try {
        value = maQueue.get();
        System.out.println("Consumer get : " + value);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
  }
}
```

Exemple :

```
package com.jmdoudoux.dej.thread;

public class TestProducerConsumer {

  public static void main(String[] args) {
    MaQueue maQueue = new MaQueue();
    MonProducer producer = new MonProducer(maQueue);
    MonConsumer consumer = new MonConsumer(maQueue);

    consumer.start();
    producer.start();

    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    producer.interrupt();
    consumer.interrupt();
  }
}
```

C'est l'objet partagé qui assure la synchronisation des deux threads : comme il ne peut contenir qu'une seule donnée, il est nécessaire de bloquer le producer et de débloquer le consumer lorsqu'une donnée est déjà présente et inversement lorsque la donnée est consommée.

Cette synchronisation est nécessaire pour éviter au consumer de rater une valeur si le producer en envoie une autre alors que le consumer traite encore la valeur précédente ou que le consumer traite plusieurs fois le même message tant que le producer n'a pas ajouté une nouvelle valeur.

La synchronisation permet de garantir qu'une donnée ne sera traitée qu'une seule fois et qu'une nouvelle valeur ne pourra être ajoutée que s'il n'y a pas de valeur à traiter.

Elle utilise dans l'exemple deux mécanismes :

- un moniteur : celui utilisé est celui de l'instance de type MaQueue partagée
- les méthodes wait() et notifyAll() de l'instance de type MaQueue partagée pour permettre de bloquer un thread en attendant un message d'un autre thread

Au premier abord, il peut sembler bizarre que le thread qui attend ait posé le verrou sur le monitor, laissant présager que l'autre thread attendra indéfiniment puisqu'il attend la pose du verrou pour notifier l'autre thread.

Cela fonctionne pourtant bien car lorsque la méthode wait() est exécutée, elle libère automatiquement le verrou posé sur le monitor. Le verrou est de nouveau posé sur le monitor à la fin de l'exécution de la méthode wait(). Ceci permet au thread qui n'est pas en attente de poser le verrou sur le monitor libéré par l'invocation de la méthode wait() dans l'autre thread. Celui-ci pourra alors poser le verrou sur le monitor et envoyer une notification.

31.9. Les restrictions sur les threads

L'utilisation de threads présente plusieurs limitations.

Il n'est pas possible de relancer un thread qui s'est terminé : il est nécessaire de créer une nouvelle instance de type Thread et de la lancer. Il est cependant possible de lui passer en paramètre la même instance de Runnable.

La méthode clone() de la classe Thread renvoie toujours une exception de type CloneNotSupportedException.

La classe Thread n'est pas Serializable essentiellement car elle a besoin de ressources systèmes obtenues par la JVM. Ces ressources seront forcément différentes dans une autre JVM. C'est une très mauvaise idée de définir une classe qui héritent de la classe Thread et qui implémente Serializable : cette classe fille a la responsabilité de sérialiser les champs de sa classe mère Thread ce qui est complexe car la classe Thread possède des champs private.

Le nombre de threads qu'il est possible de lancer dans une JVM n'est pas illimité et dépend de plusieurs facteurs :

- les ressources systèmes : le microprocesseur, la mémoire disponible sur le système
- du système d'exploitation
- de l'implémentation de la JVM utilisée
- de la configuration implicite ou explicite de certains paramètres de la JVM notamment la taille par défaut de la pile et la taille du heap

L'option -Xss d'une JVM HotSpot permet de préciser la taille par défaut de la pile des threads.

Attention : atteindre le nombre maximal de threads peut rendre le système d'exploitation instable voire même le mettre en péril.

Plutôt que de lancer de très nombreux threads, il est possible pour de nombreux scénarios de lancer les threads dans un pool de threads par exemple en utilisant un ExecutorService. Ceci permet d'avoir un contrôle sur le nombre de threads lancés et donc sur les ressources utilisées.

31.10. Les threads et les classloaders

Les classes en Java sont chargées par un classloader. Par défaut, la hiérarchie de classloaders recherche par délégation une classe dans les jars système (bootstrap classloader) et dans le classpath (system classloader). Il est aussi possible de créer ses propres classloader pour rechercher une classe dans un autre endroit (solution généralement mises en oeuvre par les conteneurs des serveurs d'applications ou par le plug-in d'exécution d'applets). Une même classe chargée par deux classloaders différents sera chargées deux fois dans la JVM.

Des threads peuvent accéder à des classes partagées avec d'autres threads, indépendamment du classloader ou des classloaders utilisés pour les charger.

A chaque thread est assigné un classloader particulier nommé context classloader. Ce classloader peut être obtenu en invoquant la méthode getContextClassloader() de la classe Thread et modifié en utilisant la méthode setContextClassloader().

Le context classloader permet de charger des classes et des ressources dans des cas particuliers. Par exemple, le context classloader est utilisé par des serveurs d'applications ou pour la sérialisation d'objets en utilisant le protocole IIOP. Dans ce dernier cas, les classes de l'ORB sont chargées par le bootstrap classloader qui ne permettra probablement pas

de charger la classe applicative lors de la désérialisation. Dans ces cas, la solution est d'utiliser un context classloader qui sera utilisé pour charger les classes.

Le context classloader peut être modifié à tout moment.

Le context classloader par défaut d'un thread est le classloader de la classe de l'instance qui crée le thread : c'est généralement le classloader applicatif sauf si un classloader dédié est utilisé.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestThreadClassloader {
4
5    public static void main(String[] args) {
6
7      afficherInfo();
8
9      Thread thread = new Thread(new Runnable() {
10
11       @Override
12       public void run() {
13         afficherInfo();
14       }
15     });
16     thread.start();
17   }
18
19   private static void afficherInfo() {
20     System.out.println(Thread.currentThread().getName());
21     System.out.println(Thread.currentThread().getClass().getClassLoader());
22     System.out.println(Thread.currentThread().getContextClassLoader());
23   }
24 }
```

Résultat :

```
1  main
2  null
3  sun.misc.Launcher$AppClassLoader@1cde100
4  Thread-0
5  null
6  sun.misc.Launcher$AppClassLoader@1cde100
```

Sauf si un classloader personnalisé est utilisé, il n'est généralement pas nécessaire de modifier le context classloader.

31.11. Les threads et la gestion des exceptions

Une exception est propagée dans la pile d'appels du thread courant.

La méthode run() ne peut pas propager d'exception de type checked : les traitements de la méthode run() ne peuvent lever et propager que des exceptions de type unchecked (runtime et error).

Toutes les exceptions qui ne sont pas gérées explicitement dans le code des traitements du thread sont gérées par un mécanisme dédié nommé gestionnaire d'exceptions non capturées (uncaught exceptions handler) avant que le thread se termine.

Chaque thread possède un gestionnaire d'exceptions non capturées par défaut qui invoque la méthode uncaughtException() du groupe de threads auquel appartient le thread.

La classe ThreadGroup implémente l'interface Thread.UncauchtExceptionHandler. La JVM va invoquer la méthode uncauchtException() du ThreadGroup si le thread n'a pas de gestionnaire d'exceptions non capturées dédié.

L'implémentation par défaut de la méthode uncaughtException() affiche pour tout Throwable sauf ThreadDeath sur la sortie standard d'erreurs :

- Exception in thread suivi du nom du thread entre double quote
- la stacktrace du thread

Il est possible de définir explicitement son propre gestionnaire d'exceptions non capturées pour par exemple journaliser l'exception ou envoyer un mail.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class AlerteSurExceptionThreadGroup extends ThreadGroup {
4
5    public AlerteSurExceptionThreadGroup() {
6      super("Alerte sur Exception ThreadGroup");
7    }
8
9    public AlerteSurExceptionThreadGroup(String name) {
10     super(name);
11   }
12
13   public AlerteSurExceptionThreadGroup(ThreadGroup parent, String name) {
14     super(parent, name);
15   }
16
17   @Override
18   public void uncaughtException(Thread t, Throwable e) {
19     // actions pour envoyer l'alerte
20     System.err.println("Exception non capturee dans le thread " + t.getName());
21     e.printStackTrace();
22   }
23 }
```

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TestAlerteSurExceptionThreadGroup {
7
8    public static void main(String[] args) {
9      AlerteSurExceptionThreadGroup tg = new AlerteSurExceptionThreadGroup();
10
11     Thread t = new Thread(tg, new Runnable() {
12
13       @Override
14       public void run() {
15         List<byte[]> liste = new ArrayList<byte[]>();
16
17         // va lever une OutOfMemoryError
18         while (true) {
19           liste.add(new byte[1024]);
20         }
21       }
22     });
23     t.start();
24   }
25 }
```

Résultat :

```
Exception non capturee dans le thread Thread-0
java.lang.OutOfMemoryError: Java heap space
      at com.jmdoudoux.dej.thread.TestAlerteSurExceptionThreadGroup$1.run(
TestAlerteSurExceptionThreadGroup.java:19)
      at java.lang.Thread.run(Thread.java:662)
```

A partir de Java 5, il est possible de définir ou de modifier le gestionnaire d'exceptions non capturées d'un thread particulier en invoquant sa méthode setUncaughtExceptionHandler() qui attend en paramètre l'instance du gestionnaire à utiliser.

Exemple (code Java 5.0) :

```
package com.jmdoudoux.dej.thread;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.ArrayList;
import java.util.List;

public class TestAlerteSurExceptionThread {

  public static void main(String[] args) {
    Thread t = new Thread(new Runnable() {

      @Override
      public void run() {
        List<byte[]> liste = new ArrayList<byte[]>();

        // va lever une OutOfMemoryError
        while (true) {
          liste.add(new byte[1024]);
        }
      }
    });

    t.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {

      @Override
      public void uncaughtException(Thread t, Throwable e) {
        // actions pour envoyer l'alerte
        System.err.println("Exeception non capturee dans le thread " + t.getName());
        e.printStackTrace();
      }
    });

    t.start();
  }
}
```

La possibilité d'ajouter un gestionnaire dédié à un thread particulier par Java 5.0 est compatible avec la gestion des exceptions non capturées de la version précédente.

La méthode getUncaughtExceptionHandler() permet d'obtenir le gestionnaire d'exceptions non capturées qui sera invoqué au besoin par la JVM : soit celui défini explicitement soit celui par défaut.

La méthode static setDefaultExceptionHandler() permet de définir le handler qui sera utilisé par tous les nouveaux threads créés. Son invocation n'a aucun effet sur les threads déjà créés.

La méthode getDefaultExceptionHandler() permet d'obtenir le handler par défaut.

31.11.1. L'exception ThreadDeath

Une instance d'une exception de type ThreadDeath est levée par la JVM lors de l'invocation de la méthode stop() d'un thread.

La classe ThreadDead hérite de la classe Error même si c'est en fait une exception standard : ceci évite d'avoir à déclarer sa propagation dans la méthode run() et évite que celle-ci ne soit interceptée par un clause catch sur le type Exception.

La méthode uncaughtException() de la classe Thread gère par défaut de manière particulière une exception de type ThreadDeath en l'ignorant. Pour toutes les autres exceptions, elle affiche sur la sortie d'erreurs un message et la stacktrace.

Par défaut, la JVM va invoquer la méthode dispatchUncaughtException() de la classe Thead : celle-ci invoque la méthode getUncaughtExceptionHandler() qui renvoie l'objet de type UncaughtExceptionHandler explicitement associé au thread ou à défaut renvoie le ThreadGroup du thread puisqu'il implémente l'interface Thread.UncaughtExceptionHandler.

Par défaut, l'implémentation de la méthode uncaughtException(Thread, Throwable) de la classe ThreadGroup effectue plusieurs traitements :

- la méthode uncaughtException() du groupe de threads parent est invoquée avec les mêmes paramètres si une telle instance existe
- sinon elle invoque la méthode uncaughtException() du UncaughtExceptionHandler par défaut si celui-ci est défini
- sinon si l'exception n'est pas de type ThreadDeath alors elle affiche sur la sortie d'erreur le message «Exception in thread » suivi du nom du thread et la stacktrace

Même si cela n'est pas recommandé, il est possible de lever soi-même une exception de type ThreadDeath pour mettre fin à l'exécution d'un thread de manière silencieuse. Attention cependant, car les contraintes qui ont forcé le JDK lui-même à ne plus appliquer cette technique s'appliquent aussi pour une utilisation directe par le développeur. La seule vraie différence est que cette technique ne peut être utilisée dans tous les cas. Si le développeur est en mesure de garantir qu'au moment où l'exception sera levée, il ne peut pas y avoir de données inconsistantes, alors il est possible de l'utiliser. Contrairement à l'invocation de la méthode stop(), le compilateur ne dira rien si une exception de ThreadDeath est levée. Cependant, elle ne doit être une solution à n'utiliser que pour des besoins très spécifiques impliquant qu'il n'y pas d'autres solutions plus propres à mettre en oeuvre.

Les traitements d'un thread peuvent capturer cette exception uniquement si des traitements particuliers doivent être exécutés avant de terminer brutalement le thread : c'est par exemple le cas si des actions de nettoyage doivent être faites pour laisser le système dans un état propre (libération de ressources, ...)

Si une exception ThreadDeath est capturée alors il est important qu'elle soit relevée pour permettre au thread de s'arrêter.

31.12. Les piles

Lors de la création d'un nouveau thread, la JVM alloue un espace mémoire qui lui est dédié nommé pile (stack). La JVM stocke des frames dans la pile.

La pile d'un thread est un espace de mémoire réservée au thread pour stocker et gérer les informations relatives à l'invocation des différentes méthodes effectuée par les traitements du thread. Chaque invocation d'une méthode ajoute une entrée dans le pile qui contient entre autres une référence sur la méthode et ses paramètres.

C'est la raison pour laquelle la taille de la pile doit être suffisamment importante pour stocker les différentes invocations d'une méthode notamment si elle est invoquée de manière récursive et que ce nombre d'appels est important.

La pile permet de garder une trace des invocations successives de méthodes.

Chaque thread possède sa propre pile qui stocke :

- les variables locales sous la forme de types primitifs. Si c'est un objet, c'est la référence qui est stockée, l'objet lui-même est stocké dans le heap
- chaque invocation d'une méthode ajoute une entrée nommée frame en haut de la pile

Lorsque l'exécution de la méthode est terminée, la frame est retirée de la pile. Les variables qu'elle contient sont supprimées : si ces variables sont des objets, leurs références sont supprimées mais ils existent toujours dans le heap. Si aucune autre référence sur ces objets existe, le ramasse-miettes les détruira à sa prochaine exécution.

La première frame de la pile est la méthode run() du thread. Chaque frame contient les variables locales de la méthode en cours d'exécution :

- les paramètres de la méthode
- les variables locales
- l'instruction en cours d'exécution
- des informations utiles pour le thread

Par défaut, jusqu'à la version Java 6 u23, pour une variable locale qui est un objet :

- la frame de la pile contient une référence vers l'objet
- l'objet est stocké dans le heap

La JVM ne stocke que des primitives dans la pile pour lui permettre de conserver une taille la plus petite possible et ainsi permettre d'imbriquer plus d'invocations de méthodes. Tous les objets sont créés dans le heap et seulement des références sur ces objets sont stockées dans la pile.

Les informations stockées dans le heap et la pile ont un cycle de vie différent :

- les informations contenues dans la pile ont une durée de vie courte : leur portée est liée à la durée d'exécution de la méthode qui les a créées. Une fois que les traitements de la méthode sont terminés et qu'elle a renvoyé une valeur (ou void) les informations concernées sont retirées de la pile
- les objets sont créés dans le heap : leur cycle de vie est géré par la machine virtuelle jusqu'à leur destruction par le ramasse-miettes

La durée de vie des valeurs stockées dans la pile est liée à la méthode dans laquelle elles ont été créées : une fois l'exécution de la méthode terminée, elles sont supprimées.

31.12.1. Les threads et la mémoire

Bien que Java définisse la taille de chaque type de variable, la taille de la pile est dépendante de la plateforme et de l'implémentation de la JVM :

- l'espace requis pour stocker une variable dans la pile peut varier selon la plateforme, essentiellement pour optimiser les opérations réalisées par certains CPU
- une frame contient des informations utiles au thread qui sont spécifiques à l'implémentation de la JVM. La quantité de données requises est donc dépendante de l'implémentation voire même de la version de cette implémentation

La taille par défaut de la pile d'un thread est donc dépendante de l'implémentation de la JVM, du système d'exploitation et de l'architecture CPU.

Depuis la version 1.4 de Java, une surcharge du constructeur de la classe Thread permet de préciser la taille de la pile à utiliser. Par exemple, ceci peut être particulièrement utile pour un thread qui fait beaucoup d'invocations récursives d'une méthode.

Remarque : il n'y a aucune garantie que la même valeur fournie à plusieurs implémentations d'une JVM ait le même effet vu que la pile est dépendante du système d'exploitation utilisé.

Attention : l'implémentation de la JVM peut modifier cette valeur à sa guise notamment si celle-ci est trop petite, trop grande ou doit être un multiple d'une certaine taille pour respecter une contrainte liée au système d'exploitation.

Les spécifications de la JVM permettent à l'implémentation d'avoir une taille de pile fixe ou une taille dynamique qui peut varier selon les besoins.

Généralement, la JVM permet de configurer la taille des piles. Cette option n'est pas standard. Par exemple, avec la JVM Hotspot, il faut utiliser l'option -Xss

Résultat :

```
1 java -Xss1024k MonApplication
```

Une nouvelle frame est créée à chaque invocation d'une méthode. La frame est détruite lorsque l'exécution de la méthode se termine de manière normale ou à cause de la levée d'une exception.

Chaque frame contient un tableau des variables locales : la taille de ce tableau est déterminée par le compilateur et stockée dans le fichier .class. Les premiers éléments du tableau sont les paramètres utilisés lors de l'invocation de la méthode.

Chaque frame contient une pile de type LIFO des opérandes (operand stack). La taille de cette pile est déterminée par le compilateur. La JVM utilise cette pile pour charger ou utiliser des opérandes mais aussi pour préparer des variables à être passées en paramètre d'une méthode ou pour recevoir le résultat d'une méthode.

Plusieurs limitations de la mémoire liées à une pile peuvent lever une exception :

- une exception de type StackOverFlowError est levée si la pile est trop petite
- une exception de type OutOfMemoryError si le système ne peux pas allouer la mémoire requise pour la pile d'un nouveau thread ou si la taille de la pile ne peut pas être dynamiquement agrandie

Exemple :

```java
package com.jmdoudoux.dej.thread;

public class TestThreadOOME {

  public static void main(String[] args) {
    for (int i = 0; i < 1000; i++) {
      MonThread t = new MonThread();
      t.start();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
            at java.lang.Thread.start0(Native Method)
            at java.lang.Thread.start(Thread.java:640)
            at com.jmdoudoux.dej.thread.TestThreadOOME.main(TestThreadOOME.java:9)
```

Remarque : il est possible qu'une exception de type OutOfMemoryError soit levée lors de la création d'un thread s'il n'y a plus d'espace dans le heap pour stocker le nouvel objet de type Thread.

L'estimation de la taille optimale d'une pile est compliquée car elle doit tenir compte de plusieurs facteurs liés au code exécuté par le thread :

- le nombre de méthodes invoquées successivement
- le nombre de paramètres de chacune de ces invocations
- l'implémentation de la JVM : la taille requise pour chaque paramètre peut varier entre différentes implémentations

L'espace mémoire requis par la pile d'un thread n'est pas pris dans le heap mais dans la mémoire générale de la machine virtuelle. Ceci a un effet limitant sur le nombre de threads que la machine virtuelle pourra lancer.

Sur un système d'exploitation la taille maximale d'un processus est limitée (généralement en fonction de l'architecture du processeur et de son implémentation). Cette taille maximale doit permettre de contenir les différentes zones de mémoire de la JVM :

- heap
- stacks
- code

Par exemple, sur un Windows 32 bits, la taille maximale de mémoire allouable à un processus est de l'ordre de 2Go. Ces 2Go doivent contenir, le heap, la mémoire requise par la JVM (permgen, ...), les bibliothèques natives et les différentes piles des threads de la JVM. Ceci combiné à la taille d'une pile implique une limitation sur le nombre maximum de threads qui peuvent être exécutés dans une JVM.

Exemple :

```java
package com.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicInteger;

public class TestThreadsLimite {
  private final static AtomicInteger compteur = new AtomicInteger(0);

  public static void main(String[] argv) {
    try {
      for (;;) {
        Thread thread = new Thread(new Runnable() {
          public void run() {
            compteur.incrementAndGet();
            for (;;) {
              try {
                Thread.sleep(1000);
              } catch (Exception e) {
```

```
18                }
19              }
20            }
21          });
22          thread.setDaemon(true);
23          thread.start();
24        }
25      } catch (Throwable e) {
26        System.out.println("Thread numero " + compteur.get());
27        e.printStackTrace();
28      }
29    }
30  }
```

Résultat sur un Windows 32 bits :

Taille du heap (-Xmx)	Taille par défaut de la pile (-Xss)	Nombre de threads créés avant OOME
64m	10k	23272
64m	1024k	1817
512m	10k	18747
512m	1024k	1369
1024m	10k	11249
1024m	1024k	866

31.12.2. L'obtention d'informations sur la pile

La classe Thread possède plusieurs méthodes qui permettent d'obtenir des informations sur la pile d'un thread :

Méthode	Rôle
int countStackFrames()	deprecated
static void dumpStack()	Afficher la pile du thread courant sur la sortie d'erreur
StackTraceElement[] getStackTrace( )	Renvoyer un tableau de type StackTraceElement qui contient toutes les méthodes de la pile du thread
static Map getAllStackTraces( )	Obtenir une map contenant pour chaque thread (en clé) un tableau de type StackTraceElement (en valeur) qui contient toutes les méthodes de sa pile

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestGetStackTrace {
4
5    public static void main(String[] args) {
6      final StringBuilder str = new StringBuilder();
7      final StackTraceElement[] stack = Thread.currentThread().getStackTrace();
8      for (final StackTraceElement ste : stack) {
9        str.append(ste);
10       str.append("\n");
11     }
12     System.out.println(str.toString());
13
```

```
14      }
    }
```

Résultat :

```
1  java.lang.Thread.getStackTrace(Thread.java:1479)
2  com.jmdoudoux.dej.thread.TestGetStackTrace.main(TestGetStackTrace.java:7)
```

31.12.3. L'escape analysis

L'escape analysis est une analyse qui permet de déterminer si pour une stack frame un objet reste confiné dans un seul thread. Si tel est le cas, le compilateur peut optimiser les performances en allouant l'objet dans la pile voire même, si l'objet est petit, en stockant directement ses champs dans les registres.

Cette fonctionnalité permet donc à la JVM de détecter si un objet créé localement dans une méthode ne pourra pas être référencé en dehors de cette méthode. Ceci doit garantir que l'objet ne possèdera plus de référence à la fin de l'exécution de la méthode. Dans ce cas, l'objet est alloué dans la pile ainsi que toutes les variables de ses champs.

L'escape analysis est une technique utilisée par le compilateur C2 de la JVM Hotspot : elle permet l'analyse de l'utilisation d'un objet afin de potentiellement mettre en oeuvre certaines optimisations :

- allocation d'un objet dans la pile plutôt que dans le heap si sa portée reste dans la méthode ou le thread
- pour ces objets, ne pas générer les traitements de pose de verrous (locks) puisqu'ils ne sont utilisés que dans le contexte local du thread

L'allocation d'objets dans la pile améliore les performances car cela évite à ces objets d'être gérés par le ramasse-miettes. Les objets sont créés dans la stack frame : dès que la méthode est terminée, la stack frame et tout ce qu'elle contient est supprimé.

La pile est par nature non fragmentée : les différentes stack frames sont empilées et dépilées dans l'ordre inverse. A contrario, la fragmentation est une contrainte importante dans la gestion du heap. Les objets deviennent récupérables par le ramasse-miette dans un ordre aléatoire par rapport à leur création dans le heap. Lorsque le ramasse-miettes récupère les objets inutiles, cela laisse des espacse vides non contigus. Deux grandes stratégies sont alors utilisables :

- compacter le heap : allonge le temps du stop the world requis par le ramasse-miettes mais rend la création d'une nouvelle instance rapide car il suffit de l'ajouter après la dernière instance créée
- ne pas compacter le heap : allonge le temps de création d'une nouvelle instance qui doit trouver un espace vide suffisant mais diminue le temps de pause du ramasse-miettes

L'allocation d'un objet dans la pile n'implique pas ces mécanismes. Si un objet est alloué dans la pile, alors cette donnée est hors de la portée du ramasse-miettes qui ne travaille que sur le heap et la permgen. L'objet est immédiatement supprimé dès que la stack frame est retirée de la pile.

A partir de la version 6 update 14, la JVM Hotspot (version 14) propose un support de l'escape analysis. Son activation se fait en utilisant l'option -XX:+DoEscapeAnalysis de la JVM Hotspot.

A partir de la version Java 6u23, il est activé par défaut lorsque le mode C2 du compilateur JIT est utilisé.

Cette fonctionnalité est disponible dans d'autres langages notamment C# et dans d'autres JVM notamment la J9 d'IBM.

La version de Java utilisée dans la suite de cette section est la 6u43 sur un Windows XP 32 bits.

Résultat :

```
1  C:\Java\TestThreads\src>java -version
2  java version "1.6.0_43"
3  Java(TM) SE Runtime Environment (build 1.6.0_43-b01)
4  Java HotSpot(TM) Client VM (build 20.14-b01, mixed mode)
```

La classe de test effectue une boucle pour créer une instance d'un objet dont la portée ne sort pas de la méthode.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestEscapeAnalysis {
```

```
4
5     private static class MonBean {
6       private long        valeur;
7       private static long compteur;
8
9       public MonBean() {
10        valeur = compteur++;
11      }
12    }
13
14    public static void main(String[] args) {
15      System.out.println("debut");
16      long startTime = System.currentTimeMillis();
17
18      for (long i = 0; i < 1000000000L; ++i) {
19        MonBean monBean = new MonBean();
20      }
21
22      long duree = System.currentTimeMillis() - startTime;
23      System.out.println("fin compteur=" + MonBean.compteur);
24      System.out.println("Temps d'execution : " + duree);
25    }
26  }
```

Les exécutions vont utiliser plusieurs configurations différentes mais à chaque fois la JVM affiche des informations sur l'activité du ramasse-miettes.

Résultat :

```
1  C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
2  -XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
3  debut
4  fin compteur=1000000000
5  Temps d'execution : 1063
6  Heap
7   PSYoungGen      total 15040K, used 1715K [0x1eaf0000, 0x1fbb0000, 0x24040000)
8    eden space 12928K, 13% used [0x1eaf0000,0x1ec9cd38,0x1f790000)
9    from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
10   to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
11  PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
12   object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
13  PSPermGen       total 16384K, used 1783K [0x10040000, 0x11040000, 0x14040000)
14   object space 16384K, 10% used [0x10040000,0x101fdf70,0x11040000)
```

Comme la version de Java utilisée est la 6u43, l'escape analysis est activé par défaut.

Résultat :

```
1  C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+PrintGCDetails
2  -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
3  debut
4  fin compteur=1000000000
5  Temps d'execution : 1078
6  Heap
7   PSYoungGen      total 15040K, used 1971K [0x1eaf0000, 0x1fbb0000, 0x24040000)
8    eden space 12928K, 15% used [0x1eaf0000,0x1ecdcd38,0x1f790000)
9    from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
10   to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
11  PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
12   object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
13
```

```
 PSPermGen       total 16384K, used 1783K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101fdf70,0x11040000)
```

Lors de la désactivation de l'escape analysis, le temps d'exécution est multiplié par quatre.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:-DoEscapeAnalysis
-XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
[GC [PSYoungGen: 12928K->192K(15040K)] 12928K->192K(49408K), 0.0014329 secs]
[GC [PSYoungGen: 13120K->176K(27968K)] 13120K->176K(62336K), 0.0004556 secs]
[GC [PSYoungGen: 26032K->200K(27968K)] 26032K->200K(62336K), 0.0009065 secs]
[GC [PSYoungGen: 26056K->184K(53824K)] 26056K->184K(88192K), 0.0011468 secs]
[GC [PSYoungGen: 51896K->184K(53824K)] 51896K->184K(88192K), 0.0009158 secs]
[GC [PSYoungGen: 51896K->192K(83392K)] 51896K->192K(117760K), 0.0008146 secs]
[GC [PSYoungGen: 83328K->0K(83392K)] 83328K->152K(117760K), 0.0014555 secs]
[GC [PSYoungGen: 83136K->0K(86976K)] 83288K->152K(121344K), 0.0008565 secs]
...
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0003433 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0007660 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0007937 secs]
fin compteur=1000000000
Temps d'execution : 4031
Heap
 PSYoungGen      total 87296K, used 26193K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 30% used [0x1eaf0000,0x20484420,0x24020000)
  from space 64K, 0% used [0x24020000,0x24020000,0x24030000)
  to   space 64K, 0% used [0x24030000,0x24030000,0x24040000)
 PSOldGen        total 34368K, used 152K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14066060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ff968,0x11040000)
```

Le facteur d'amélioration des performances lié à l'utilisation de l'escape analysis peut être important.

L'escape analysis ne fonctionne qu'avec le mode C2 du compilateur (active avec l'option -server de la JVM Hotspot)

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -client -verbose:gc -XX:+DoEscapeAnalysis
-XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
Unrecognized VM option '+DoEscapeAnalysis'
Could not create the Java virtual machine.
```

Avec le mode C1 du compilateur (activé avec l'option -client de la JVM Hotspot), l'activité du ramasse-miettes est importante et le temps d'exécution est multiplié par dix.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -client -verbose:gc -XX:+PrintGCDetails
-cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [DefNew: 4480K->0K(4992K), 0.0001198 secs] 4602K->122K(15936K), 0.0002436 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001372 secs] 4602K->122K(15936K), 0.0003056 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001307 secs] 4602K->122K(15936K), 0.0002646 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001333 secs] 4602K->122K(15936K), 0.0003079 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001316 secs] 4602K->122K(15936K), 0.0002752 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001349 secs] 4602K->122K(15936K), 0.0002760 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001436 secs] 4602K->122K(15936K), 0.0003056 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001316 secs] 4602K->122K(15936K), 0.0002682 secs]
```

```
[GC [DefNew: 4480K->0K(4992K), 0.0001416 secs] 4602K->122K(15936K), 0.0003037 secs]
...
[GC [DefNew: 4480K->0K(4992K), 0.0001631 secs] 4602K->122K(15936K), 0.0002998 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001620 secs] 4602K->122K(15936K), 0.0003003 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001310 secs] 4602K->122K(15936K), 0.0002685 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001349 secs] 4602K->122K(15936K), 0.0002752 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001361 secs] 4602K->122K(15936K), 0.0002788 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001313 secs] 4602K->122K(15936K), 0.0002838 secs]
fin compteur=1000000000
Temps d'execution : 10078
Heap
 def new generation   total 4992K, used 2904K [0x10040000, 0x105a0000, 0x15590000)
  eden space 4480K,  64% used [0x10040000, 0x10316068, 0x104a0000)
  from space 512K,   0% used [0x10520000, 0x10520000, 0x105a0000)
  to   space 512K,   0% used [0x104a0000, 0x104a0000, 0x10520000)
 tenured generation   total 10944K, used 122K [0x15590000, 0x16040000, 0x20040000)
   the space 10944K,   1% used [0x15590000, 0x155aeaf0, 0x155aec00, 0x16040000)
 compacting perm gen  total 12288K, used 1752K [0x20040000, 0x20c40000, 0x24040000)
   the space 12288K,  14% used [0x20040000, 0x201f6080, 0x201f6200, 0x20c40000)
No shared spaces configured.
```

Si l'instance de type MonBean sort de la portée de la méthode, celle-ci est instanciée dans le heap.

Exemple :

```
package com.jmdoudoux.dej.thread;


public class TestEscapeAnalysis {

  private static MonBean courant = null;

  private static class MonBean {
    private long        valeur;
    private static long compteur;

    public MonBean() {
      valeur = compteur++;
    }
  }

  public static void main(String[] args) {
    System.out.println("debut");
    long startTime = System.currentTimeMillis();

    for (long i = 0; i < 1000000000L; ++i) {
      MonBean monBean = new MonBean();
      courant = monBean;
    }

    long duree = System.currentTimeMillis() - startTime;
    System.out.println("fin compteur=" + MonBean.compteur);
    System.out.println("Temps d'execution : " + duree);
  }
}
```

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
 -XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [PSYoungGen: 83472K->16K(87296K)] 83612K->156K(121664K), 0.0005009 secs]
```

```
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0007736 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005297 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005481 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005425 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006979 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006744 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005339 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006666 secs]
fin compteur=1000000000
Temps d'execution : 4438
Heap
 PSYoungGen      total 87296K, used 7023K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 8% used [0x1eaf0000,0x1f1c7f00,0x24020000)
  from space 64K, 25% used [0x24020000,0x24024000,0x24030000)
  to   space 64K, 0% used [0x24030000,0x24030000,0x24040000)
 PSOldGen        total 34368K, used 140K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14063060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ffa98,0x11040000)
```

L'instance est aussi créée dans la pile si elle est utilisée en paramètre d'une méthode invoquée tout en ne sortant pas de la portée du thread.

Exemple :

```java
package com.jmdoudoux.dej.thread;

public class TestEscapeAnalysis {

  private static class MonBean {
    private long        valeur;
    private static long compteur;

    public MonBean() {
      valeur = compteur++;
    }
  }

  public static void main(String[] args) {
    System.out.println("debut");
    long startTime = System.currentTimeMillis();

    for (long i = 0; i < 1000000000L; ++i) {
      MonBean monBean = new MonBean();
      traiter(monBean);
    }

    long duree = System.currentTimeMillis() - startTime;
    System.out.println("fin compteur=" + MonBean.compteur);
    System.out.println("Temps d'execution : " + duree);
  }

  private static MonBean traiter(MonBean monBean) {
    return monBean;
  }
}
```

Avec l'escape analysis activée, l'activité du ramasse-miettes est très faible.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
-XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
fin compteur=1000000000
Temps d'execution : 1062
Heap
 PSYoungGen      total 15040K, used 1572K [0x1eaf0000, 0x1fbb0000, 0x24040000)
  eden space 12928K, 12% used [0x1eaf0000,0x1ec791f8,0x1f790000)
  from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
  to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
 PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
 PSPermGen       total 16384K, used 1784K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101fe128,0x11040000)
```

Avec l'escape analysis désactivée, l'activité du ramasse-miettes est beaucoup plus intense et le temps d'exécution est multiplié par quatre.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:-DoEscapeAnalysis
-XX:+PrintGCDetails -cp . com.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0004464 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0007208 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0007322 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006540 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0005763 secs]
...
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0003632 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006383 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006906 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0004864 secs]
fin compteur=1000000000
Temps d'execution : 4030
Heap
 PSYoungGen      total 87296K, used 29781K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 34% used [0x1eaf0000,0x20805710,0x24020000)
  from space 64K, 0% used [0x24030000,0x24030000,0x24040000)
  to   space 64K, 0% used [0x24020000,0x24020000,0x24030000)
 PSOldGen        total 34368K, used 148K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14065060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ffb20,0x11040000)
```

L'endroit où est alloué un objet est uniquement géré par la JVM. Les possibilités pour le développeur d'influencer ce choix sont restreintes car il n'est pas possible d'indiquer dans le code que cet objet doit être instancié dans la pile :

- configuration de certaines options de la JMV
- bien tenir compte de la portée des variables en limitant celle-ci au strict minimum

L'endroit où un objet est alloué importe peu sur la bonne exécution des traitements, cependant la mise en oeuvre de ces fonctionnalités peut significativement améliorer les performances.

31.12.4. Les restrictions d'accès sur les threads et les groupes de threads

Les restrictions d'accès aux fonctionnalités des classes Thread et ThreadGroup reposent sur l'utilisation d'un SecurityManager.

Les classes Thread et ThreadGroup possède une méthode checkAccess() qui va invoquer la méthode checkAccess() du SecurityManager associé à la JVM. Si l'accès n'est pas autorisé alors une exception de type SecurityException est levée.

Plusieurs méthodes de la classe ThreadGroup invoquent la méthode checkAccess() pour obtenir la permission d'exécution par le SecurityManager :

- ThreadGroup(ThreadGroup, String)
- destroy()
- getParent()
- resume()
- setDaemon(boolean)
- setMaxPriority(int)
- stop()
- suspend()
- enumerate(Thread[]) et enumerate(Thread[], boolean)
- enumerate(ThreadGroup[]) et enumerate(ThreadGroup[], boolean)
- interrupt()

Plusieurs méthodes de la classe Thread invoquent la méthode checkAccess() pour obtenir la permission d'exécution par le SecurityManager :

- Les constructeurs qui attendent en paramètre un groupe de threads
- stop()
- suspend()
- resume()
- setPriority(int)
- setName(String)
- setDaemon(boolean)
- setUncaughtExceptionHandler(UncaughtExceptionHandler)

Sans SecurityManager, il n'y a pas de restrictions d'accès pour modifier l'état d'un thread ou d'un groupe de threads par un autre thread.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestMonThreadSecManager {
4
5    public static void main(String[] args) {
6      final ThreadGroup threadGroup1 = new ThreadGroup("groupe1");
7      final Thread t1 = new Thread(threadGroup1, new Runnable() {
8
9        @Override
10       public void run() {
11         try {
12           Thread.sleep(2000);
13         } catch (InterruptedException e) {
14         }
15         System.out.println("fin thread 1");
16       }
17     }, "thread 1");
18     t1.start();
19
20     ThreadGroup threadGroup2 = new ThreadGroup("groupe2");
21     Thread t2 = new Thread(threadGroup2, new Runnable() {
22
23       @Override
24       public void run() {
25         t1.setPriority(Thread.MIN_PRIORITY);
26         System.out.println("fin thread 2");
27       }
28     }, "thread 2");
29     t2.start();
30
31     Thread t3 = new Thread(threadGroup2, new Runnable() {
32
33       @Override
34       public void run() {
35         threadGroup1.setMaxPriority(Thread.MIN_PRIORITY);
36         System.out.println("fin thread 3");
37       }
```

```
38      }, "thread 3");
39      t3.start();
40    }
41  }
```

Résultat :

```
1  fin thread 2
2  fin thread 3
3  fin thread 1
```

Il est possible de définir son propre SecurityManager en créant une classe fille de la classe SecurityManager avec les méthodes checkAccess(Thread) et checkAccess(ThreadGroup) redéfinies selon les besoins.

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class MonThreadSecManager extends SecurityManager {
4
5    private Thread      threadPrincipal;
6    private ThreadGroup threadGroupPrincipal;
7
8    public MonThreadSecManager(Thread threadPrincipal) {
9      this.threadPrincipal = threadPrincipal;
10     this.threadGroupPrincipal = threadPrincipal.getThreadGroup();
11   }
12
13   public void checkAccess(Thread t) {
14
15     if (t != null) {
16       Thread threadCourant = Thread.currentThread();
17       ThreadGroup threadGroupCourant = threadCourant.getThreadGroup();
18
19       if (!threadPrincipal.equals(threadCourant)) {
20         System.out.println("thread        " + t);
21         System.out.println("threadCourant " + threadCourant);
22
23         if (!t.getThreadGroup().equals(threadGroupCourant))
24           throw new SecurityException("Can't modify the thread");
25       }
26     }
27   }
28
29   public void checkAccess(ThreadGroup g) {
30
31     if (g != null) {
32       Thread threadCourant = Thread.currentThread();
33       ThreadGroup threadGroupCourant = threadCourant.getThreadGroup();
34
35       if (!threadGroupPrincipal.equals(threadGroupCourant)) {
36         System.out.println("threadGroup        " + g);
37         System.out.println("threadGroupCourant " + threadGroupCourant);
38
39         if (!g.equals(threadGroupCourant))
40           throw new SecurityException("Can't modify the thread group");
41       }
42     }
43   }
44 }
```

L'implémentation du SecurityManager ci-dessus effectue certains contrôles :

- le thread principal et son groupe de threads sont autorisés notamment pour permettre la création des threads et des groupes de threads
- l'accès à un thread n'est possible que si le thread courant appartient au même groupe que lui
- l'accès à un groupe de threads n'est possible que si le thread courant lui appartient

Exemple :

```
1  package com.jmdoudoux.dej.thread;
2
3  public class TestMonThreadSecManager {
4
5    public static void main(String[] args) throws InterruptedException {
6
7      if (System.getSecurityManager() == null) {
8        System.setSecurityManager(new MonThreadSecManager(Thread
9            .currentThread()));
10     }
11
12     final ThreadGroup threadGroup1 = new ThreadGroup("groupe1");
13     final Thread t1 = new Thread(threadGroup1, new Runnable() {
14
15       @Override
16       public void run() {
17         try {
18           Thread.sleep(2000);
19         } catch (InterruptedException e) {
20         }
21         System.out.println("fin thread 1");
22       }
23     }, "thread 1");
24     t1.start();
25
26     ThreadGroup threadGroup2 = new ThreadGroup("groupe2");
27     Thread t2 = new Thread(threadGroup2, new Runnable() {
28
29       @Override
30       public void run() {
31         t1.setPriority(Thread.MIN_PRIORITY);
32         System.out.println("fin thread 2");
33       }
34     }, "thread 2");
35     t2.start();
36     t2.join();
37
38     Thread t3 = new Thread(threadGroup2, new Runnable() {
39
40       @Override
41       public void run() {
42         threadGroup1.setMaxPriority(Thread.MIN_PRIORITY);
43         System.out.println("fin thread 3");
44       }
45     }, "thread 3");
46     t3.start();
47
48     t1.join();
49   }
50 }
```

Résultat :

```
thread        Thread[thread 1,5,groupe1]
threadCourant Thread[thread 2,5,groupe2]
Exception in thread "thread 2" java.lang.SecurityException: Can't modify the thread
    at com.jmdoudoux.dej.thread.MonThreadSecManager.checkAccess(MonThreadSecManager.java:16)
    at java.lang.Thread.checkAccess(Thread.java:1306)
    at java.lang.Thread.setPriority(Thread.java:1056)
    at com.jmdoudoux.dej.thread.TestMonThreadSecManager$2.run(TestMonThreadSecManager.java:33)
    at java.lang.Thread.run(Thread.java:662)
threadGroup        java.lang.ThreadGroup[name=groupe1,maxpri=10]
threadGroupCourant java.lang.ThreadGroup[name=groupe2,maxpri=10]
Exception in thread "thread 3" java.lang.SecurityException: Can't modify the thread group
    at com.jmdoudoux.dej.thread.MonThreadSecManager.checkAccess(MonThreadSecManager.java:32)
    at java.lang.ThreadGroup.checkAccess(ThreadGroup.java:299)
    at java.lang.ThreadGroup.setMaxPriority(ThreadGroup.java:246)
    at com.jmdoudoux.dej.thread.TestMonThreadSecManager$3.run(TestMonThreadSecManager.java:44)
    at java.lang.Thread.run(Thread.java:662)
fin thread 1
```
