Dans ce document, la description des classes de l'API ne prétend aucunement être exhaustive. Reportez-vous à l'API en question pour connaître tous les détails de cette classe.

Chapitre III – Thread

Chapitre III – Thread

Chapitre III – Thread

Chapitre III – Thread

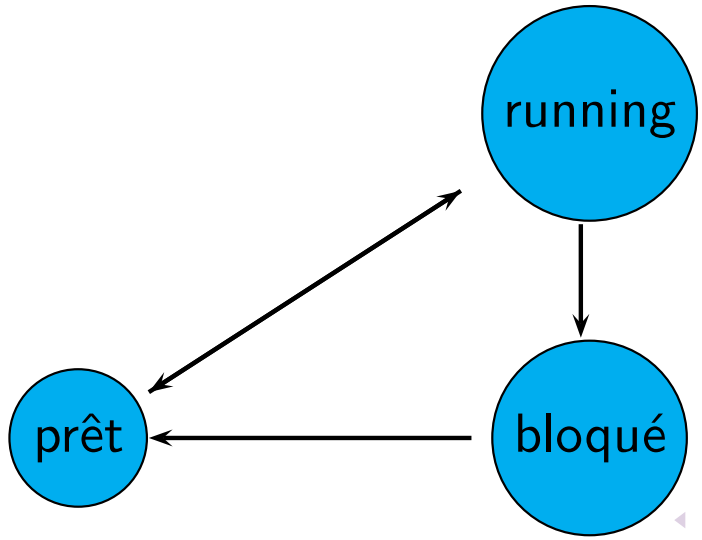
Les Threads : notion de processus

Un processus est un programme en cours d'exécution. Le système d'exploitation alloue à chaque processus une partie de mémoire (pour stocker ses instructions, variables, ...) et il lui associe des informations (identifieur, priorités, droits d'accès ...).
Un processus s'exécute sur un processeur du système. Il a besoin de ressources : le processeur qui l'exécute, de la mémoire, des entrées sorties. Certaines ressources ne possèdent qu'un point d'accès et ne peuvent donc être utilisées que par un processus à la fois (par exemple, une imprimante).
On dit alors que les processus sont en *exclusion mutuelle* s'ils partagent une même ressource qui est dite *critique*. Il est nécessaire d'avoir une politique de synchronisation pour de telle ressource partagée.

notion de processus

Par exemple concernant l'impression, il y a un processus qui gère les demandes et la file d'attente de ces demandes. Un tel processus - « invisible » - toujours en fonctionnant tant que le système fonctionne est appelé un *démon*.
Un démon - deamon - est un processus qui s'exécute en tâche de fond comme le service d'impression. En général un démon est lancé par le système d'exploitation au démarrage et stoppe à l'arrêt du système.
Un processus peut être :

- en cours d'exécution - *running*
- prêt à s'exécuter mais sans processeur libre pour l'exécuter
- bloqué (par manque de ressources)



Les threads

Un *thread* ou *processus léger* est un processus à l'intérieur d'un processus.
En Java, lorsqu'on lance la machine virtuelle pour exécuter un programme, on lance un processus ; ce processus est composé de plusieurs threads : le thread principal (qui correspond au `main`) et d'autres threads comme le ramasse-miettes.
Donc un processus est composé de plusieurs threads (ou tâches) et va devoir partager ses ressources entre ses différents threads.
En Java on a deux moyens de créer un thread :

1. étendre la classe `Thread` : on aura alors un objet qui *contrôle une tâche*,
2. implémenter l'interface `Runnable` : on aura lors un objet qui *représente le code à exécuter*.

Hériter de la classe `Thread`

Lorsque l'on hérite de la classe `Thread`, il faut réécrire la méthode `void run()` pour qu'elle exécute le comportement attendu.
Puis une fois un objet de type `Thread` créé, cet objet peut invoquer la méthode `void start()` de la classe `Thread` qui elle-même invoque la méthode `run()` réécrite.

Exemple

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
      while (true) {
         System.out.println(s);
         try { sleep(100);}
         catch(InterruptedException e){}
} }
}
public class TicTac {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
   }
}
```

A l'exécution on a un affichage continu de TIC TAC .... qui fait régulièrement une pause de 100 millisecondes.

Aperçu de la classe Thread

- constantes
  - static int MAX_PRIORITY : priorité maximale = 10
  - static int MIN_PRIORITY : priorité minimale = 1
  - static int NORM_PRIORITY : priorité normale = 5

- constructeurs
  - Thread() ;
  - Thread(Runnable cible, String nom)
  - ...
- méthodes
  - void start() invoque la méthode run
  - void run() : exécute le thread et peut lancer InterruptedException
  - int getPriority() : renvoie le niveau de priorité
  - static Thread currentThread() : renvoie le thread en cours d'exécution
  - static void sleep(long millis) : suspend l'activité du thread en cours d'exécution pendant millis millseconde
  - static void yield() : suspend l'activité du thread en cours d'exécution et permet aux autres threads de s'exécuter
  - void join() : attend la fin de l'objet Thread qui l'invoque
  - void interrupt() : interrompt `this`
  - static boolean interrupted() : teste si le thread courant a été interrompu
  - booleanisInterrupted() : test si `this` a été interrompu
  - boolean isAlive() : renvoie vrai si le thread n'a pas fini
  - String toString() : renvoie le nom du thread, sa priorité, son groupe de la classe ThreadGroup

Exemple

```
public class MaTache{
public static void main(String[] arg) throws InterruptedException{
System.out.println(Thread.currentThread());
Thread tacheInitiale = Thread.currentThread();
tacheInitiale.setName("tache initiale");
Thread.sleep(1000); // sleep est une méthode de classe
System.out.println(tacheInitiale);
// le nom du thread a été changé mais pas le nom de son groupe
System.out.println(tacheInitiale.isAlive());
Thread maTache = new Thread();
maTache.setName("ma tache");
System.out.println(maTache);
System.out.println(maTache.isAlive());
}
}
```

A l'exécution on a :

```
Thread[main,5,main]
Thread[tache initiale,5,main]
true
Thread[ma tache,5,main]
false
```

implémenter l'interface `Runnable`

L'interface `Runnable` contient uniquement la méthode `run()` qui est à implémenter.
Pour lancer un thread avec une classe implémentant `Runnable` on utilise le constructeur de la classe `Thread` qui prend en paramètre un objet `Runnable`.

Exemple

```
class TestThread2 implements Runnable{
   String s;
   public TestThread2(String s){this.s=s; }
   public void run() {
      while (true) {
         System.out.println(s);
         try { Thread.sleep(100);}
         catch(InterruptedException e){}
}}}
public class TicTac2 {
   public static void main(String arg[]){
      Thread tic=new Thread(new TestThread2("TIC"));
      Thread tac=new Thread(new TestThread2("TAC"));
      tic.start();
      tac.start();
   }
}
```

la méthode `join()`

La méthode `join()` de la classe `Thread` invoquée par un objet `Thread` t met en attente le thread en cours d'exécution jusquà ce que t soit terminé.
La méthode `join()` lance une exception de type `InterruptedException` il faut donc l'utiliser dans un bloc `try catch`.

la méthode join()

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
     for (int i =1; i<=2; i++) {
         System.out.print(s+ " ");
         try { sleep(100);}
         catch(InterruptedException e){}}
} }
}
public class TicTac3 {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
      try{tic.join();}
       catch(InterruptedException e){}
       System.out.println("c'est fini ");
   }
}
```

Exécution

TIC TAC TIC TAC c'est fini

remarque : sans l'instruction join() l'affichage de « c'est fini » se fait avant

la méthode join()

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
     for (int i =1; i<=2; i++) {
         System.out.print(s+ " ");
         try { sleep(100);}
         catch(InterruptedException e){}}
} }
}
public class TicTac3 {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
      try{tic.join();}
       catch(InterruptedException e){}
       System.out.println("c'est fini ");
   }
}
```

Exécution TIC TAC TIC TAC c'est fini

remarque : sans l'instruction join() l'affichage de≪ c'est fini ≫ se fait avant

Gestion des threads avec `synchronized`

Les threads peuvent partager des ressources. Il faut alors s'assurer que cette ressource ne sera utilisée que par un seul thread en même temps.
Pour cela on utilise un mécanisme de *verrou* : tout objet (ou tableau) possède un verrou qui peut être ouvert ou fermé. Un thread t1 peut fermer un verrou sur un objet (si le verrou n'est pas déjà fermé) et lorsqu'il termine la portion de code verrouillée il rouvre le verrou.
Pour éviter qu'un autre thread t2 ne puisse exécuter une portion de code sur l'objet verrouillé il faut que cette portion de code ait le *même mécanisme* de verrou sur cet objet.
On parle alors de *synchronisation* entre t1 et t2 et on utilise le mot `synchronized` à cet effet.
On peut synchroniser

- une méthode m : `synchronized void m()` - ici `this` est l'objet sur lequel le verrou est posé
- un objet o : `synchronized(o)`...instructions ... - ici o est l'objet sur lequel le verrou est posé

Gestion des threads

Pendant l'exécution d'une portion de code marqué `synchronized` par un thread t1, tout autre thread t2 tentant d'exécuter une portion de code marquée `synchronized` relative au même objet est suspendu.
Remarques :

- attention une méthode non synchronisée peut modifier `this` même si `this` est verrouillée par une méthode synchronisée
- une méthode statique peut être synchronisée, elle verrouille alors sa classe empêchant une autre méthode statique de s'exécuter pendant sa propore exécution
- une méthode statique synchronisée n'empêche pas les modifications sur les instances de la classe par des méthodes d'instance.

Exemple sans synchronisation

```
public class  CompteJoint{
String nomH;
String nomF;
String numCompte;
int solde=0;
public CompteJoint(String sH, String sF, String numC){
numCompte=numC ; nomH= sH; nomF = sF;}
public String toString(){return ("le compte de " + nomH + " et "+ nomF + " numéro :"+numCompte )
public void depot(int somme){
int resultat = solde;
try{Thread.sleep(100);} // temps de traitement
catch(Exception e){}
solde= somme +  resultat;
System.out.println("depot de " + somme);
}////
public class GuichetBanque extends Thread {
CompteJoint cj;
int id;
public GuichetBanque(CompteJoint cj,int n){this.cj=cj;this.id=n;}
public void run(){
System.out.println("début de la transaction sur "+
cj + " au guichet numéro " + id);
        cj.depot(100);
        System.out.println("fin de la transaction sur "+
cj + " au guichet numéro " + id);
}}
```

Exemple sans synchronisation

```
public class TestGuichetCompteJoint{
public static void main(String[] arg){
CompteJoint  unCompte = new CompteJoint("Paul", "Eve", "00100100");
GuichetBanque gb1 = new GuichetBanque(unCompte, 1);
GuichetBanque gb2 = new GuichetBanque(unCompte, 2);
gb1.start();
gb2.start();
try{gb1.join();
gb2.join();}
catch(InterruptedException e){}
System.out.println("votre solde est "+ unCompte.solde);
}
}
```

Exécution :

début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 1
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 2
depot de 100
depot de 100
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 1
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 2
votre solde est 100

Exécution :
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 1
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 2
depot de 100
depot de 100
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 1
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 2
votre solde est 100

Les deux objets `GuichetBanque` lisent le solde du compte avant de le créditer de 100. Donc chacun d'eux part d'un solde de 0.

Exemple avec synchronisation

```
public class  CompteJointSync{
String nomH;
String nomF;
String numCompte;
int solde=0;
public CompteJointSync(String sH, String sF, String numC){
numCompte=numC ; nomH= sH; nomF = sF;}
public String toString(){return ("le compte de " + nomH + " et "+ nomF + " numéro :"+numCompte )
public synchronized void depot(int somme){
int resultat = solde;
try{Thread.sleep(100);} // temps de traitement
catch(Exception e){}
solde= somme +  resultat;
System.out.println("depot de " + somme);
}////
public class GuichetBanqueSync extends Thread {
CompteJoint cj;
int id;
public GuichetBanqueSync(CompteJointSync cj,int n){this.cj=cj;this.id=n;}
public void run(){
System.out.println("début de la transaction sur "+
cj + " au guichet numéro " + id);
        cj.depot(100);
        System.out.println("fin de la transaction sur "+
cj + " au guichet numéro " + id);
}}
```

Exemple avec synchronisation

```
public class TestGuichetCompteJointSync{
public static void main(String[] arg){
CompteJointSync  unCompte = new CompteJointSync("Paul", "Eve", "00100100");
GuichetBanqueSync gb1 = new GuichetBanqueSync(unCompte, 1);
GuichetBanqueSync gb2 = new GuichetBanqueSync(unCompte, 2);
gb1.start();
gb2.start();
try{gb1.join();
gb2.join();}
catch(InterruptedException e){}
System.out.println("votre solde est "+ unCompte.solde);
}
}
```

Exécution :
début de la transaction sur le compte compte de Paul et Eve au guichet numéro 1
début de la transaction sur le compte compte de Paul et Eve au guichet numéro 2
depot de 100
fin de la transaction sur le compte compte de Paul et Eve au guichet numéro 1
depot de 100
fin de la transaction sur le compte compte de Paul et Eve au guichet numéro 2
votre solde est 200

Ici l'objet `GuichetBanqueSync gb1` invoque la méthode synchronisée `depot` et donc verrouille `cj`.
Pendant la pause de gb1, l'objet gb2 ne peut pas accéder au compte `cj` par la méthode synchronisée `depot` donc il doit attendre la fin de gb1 pour exécuter la méthode synchronisée `depot`.
Quand il lit le solde de l'objet `cj` ce solde a été crédité par gb1.

wait() et notify()

La classe `Object` a les méthodes suivantes :

- `public final void wait()`
- `public final void wait(long maxMilli)`
- `public final void wait(long maxMill, int maxNano)`
- `public final void notify()`
- `public final void notifyAll()`

Les méthodes `wait` mettent en attente le thread en cours d'exécution et les méthodes `notify`, `notifyAll` interrompent cette attente. Les `wait` doivent être invoqués dans une portion de code synchronisée ; le thread mis en attente déverouille alors l'objet qui a invoqué `wait`.

Exemple `wait notify`

Dans cet exemple on a 4 classes :

- une classe Producteur qui place les objets dans un entrepôt
- une classe Consommateur qui prend les objets dans un entrepôt
- une classe Entrepot
- une classe Test

- un Producteur ne peut mettre un objet que si l'entrepôt n'est pas plein ; il doit donc attendre qu'un Consommateur ait vidé l'entrepôt,
- un Consommateur ne peut pas prendre un objet si l'entrepôt est vide ; il doit donc attendre qu'un Producteur ait rempli l'entrepôt.

Entrepot fait la sunchronisation

```
public class Entrepot{
private static final int NB_MAX = 3;
private int nbObjet = 0;
String s;
public Entrepot(String s){this.s=s;}
public int getNbObjet(){return this.nbObjet;}
public boolean estVide(){return nbObjet==0;}
public boolean estPlein(){return nbObjet==NB_MAX;}
public String toString(){return (this.s +" (" + this.nbObjet + " objets)");}
public synchronized void mettre(){
try{ while(estPlein()) {wait();
System.out.println("prod endormi");}} //fin try
catch(Exception e){}
nbObjet++;
notifyAll();
}//fin mettre()
public synchronized void prendre(){
try{while(estVide()){ wait();
System.out.println("cons endormi");}}
catch(Exception e){}
nbObjet--;
notifyAll();
}//fin prendre()
}
```

```
public class Producteur extends Thread{
Entrepot e;
String nom;
public Producteur(Entrepot e, String s){this.e=e; this.nom=s;}
public void run(){
for (int i=1; i<=4; i++)
{
System.out.println("avant prod  reste "+e);
e.mettre();
System.out.println(" apres prod il y a " + e);
        }
}}
///
public class Consommateur extends Thread{
Entrepot e;
String nom;
public Consommateur(Entrepot e, String s){this.e=e; this.nom=s;}
public void run(){
for (int i=1; i<=4; i++)
{
System.out.println("avant cons "+e);
       e.prendre();
System.out.println("apres cons reste "+e);
}
}}///
public class TestEntrepot{
public static void main(String[] arg){
Entrepot e= new Entrepot("entrepot");
Producteur p= new Producteur(e, "prod");
Consommateur c= new Consommateur(e, "cons");
 p.start();
 c.start();
 try{p.join();c.join();}
 catch(Exception exc){}
 System.out.println("final "+e);
}}
```

Exécution

```
avant prod reste entrepot (0 objets)
apres prod il y a entrepot (1 objets)
avant prod reste entrepot (1 objets)
apres prod il y a entrepot (2 objets)
avant prod reste entrepot (2 objets)
apres prod il y a entrepot (3 objets)
avant prod reste entrepot (3 objets)
prod endormi
avant cons entrepot (3 objets)
apres cons reste entrepot (2 objets)
avant cons entrepot (2 objets)
apres cons reste entrepot (1 objets)
avant cons entrepot (1 objets)
apres cons reste entrepot (0 objets)
avant cons entrepot (0 objets)
cons endormi
apres prod il y a entrepot (1 objets)
apres cons reste entrepot (0 objets)
final entrepot (0 objets)
```

Exécution

```
avant prod reste entrepot (0 objets)
apres prod il y a entrepot (1 objets)
avant prod reste entrepot (1 objets)
apres prod il y a entrepot (2 objets)
avant prod reste entrepot (2 objets)
apres prod il y a entrepot (3 objets)
avant prod reste entrepot (3 objets)
prod endormi
avant cons entrepot (3 objets)
apres cons reste entrepot (2 objets)
avant cons entrepot (2 objets)
apres cons reste entrepot (1 objets)
avant cons entrepot (1 objets)
apres cons reste entrepot (0 objets)
avant cons entrepot (0 objets)
cons endormi
apres prod il y a entrepot (1 objets)
apres cons reste entrepot (0 objets)
final entrepot (0 objets)
```

Producteur et Consommateur font la synchronsation

Dans cette version la classe `Entrepot` n'organise pas la synchronisation qui est laissée à chacun des deux threads.

```
public class Entrepot{
//....
public void mettre(){nbObjet++;}
public void prendre(){nbObjet--;}
}
public class Producteur extends Thread{
Entrepot e;
String nom;
public Producteur(Entrepot e, String s){this.e=e; this.nom=s;}
public void run(){
try{for (int i=1; i<=4; i++)
synchronized(e){
while(e.estPlein()){e.wait();} // e verrouille l'accès au Producteur
System.out.println("avant prod "+e);
e.mettre();
System.out.println(" prod " + e);
e.notifyAll(); } }
catch(Exception exc){} } }
public class Consommateur extends Thread{
Entrepot e;
String nom;
public Consommateur(Entrepot e, String s){this.e=e; this.nom=s;}
public void run(){
try{for (int i=1; i<=4; i++)
     synchronized(e) {
while(e.estVide()){e.wait();}// e verrouille l'accès au Consommateur
System.out.println("avant cons "+e);
e.prendre();
System.out.println("cons "+e);
e.notifyAll(); } }
catch(Exception exc){} } }
```

Exécution

```
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant prod entrepot (1 objets)
prod entrepot (2 objets)
avant prod entrepot (2 objets)
prod entrepot (3 objets)
avant cons entrepot (3 objets)
cons entrepot (2 objets)
avant cons entrepot (2 objets)
cons entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
final entrepot (0 objets)
```

Exécution

```
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant prod entrepot (1 objets)
prod entrepot (2 objets)
avant prod entrepot (2 objets)
prod entrepot (3 objets)
avant cons entrepot (3 objets)
cons entrepot (2 objets)
avant cons entrepot (2 objets)
cons entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
final entrepot (0 objets)
```

Exercices

exercice 1 : Ecrire une classe `Compteur` qui hérite de la classe `Thread` ; elle a un attribut de type `String` ; sa méthode `run()` compte de 1 à n en faisant une pause aléatoire de 0 à 5s entre deux incrémentations, affiche chaque valeur incrémentée avec son nom puis affiche un message de fin.
Tester cette classe dans une classe `TestCompteur` qui lance plusieurs `Compteur`.
Modifier la méthode `run()` de la classe `Compteur` pour que le thread affiche le message de fin avec son ordre d'arrivée.
Tester la modification.

Exercices

exercice 2 : Quel est le problème du programme suivant

```
class MonObjet {
public MonObjet () {}
public synchronized void action1 (MonObjet o) {
try{Thread.currentThread().sleep(200);}
catch (InterruptedException ex) { return ; }
o.action2(this); }
public synchronized void action2 (MonObjet o) {
 try{Thread.currentThread().sleep(200);}
catch (InterruptedException ex) { return ; }
o.action1(this); } }
class MonThread extends Thread {
private MonObjet obj1 , obj2;
public Thread(MonObjet o1, MonObjet o2) {
obj1 = o1;
obj2 = o2; }
public void run() {
obj1.action1(obj2); } }
class Deadlock {
public static void main (String[] args) {
MonObjet o1 = new MonObjet (); MonObjet o2 = new MonObjet ();
MonThread t1 = new MonThread(o1,o2); t1.setName("t1");
MonThread t2 = new MonThread(o2,o1); t2.setName("t2");
t1.start(); t2.start();
}}
```