Les threads

La classe Thread

Pour rappel, un thread est une « **unité d'exécution** » ou encore « **sous-processus** » ou « **processus léger** » travaillant au sein d'un processus qui constitue une espèce de « conteneur » de données et de threads.

Pour gérer les threads, Java utilise la classe **Thread** définie dans le package java.lang. Cette classe présente essentiellement les caractéristiques :

- Elle possède une méthode **run()** → il s'agit de la fonction exécutée par le sous-processus
- Elle implémente l'interface **Runnable** → cet interface ne possède que la méthode run() → elle représente tout objet qui peut être « threadé »

De manière allégée, le code la classe thread est le suivant

```
public class Thread implements Runnable
{
  private char name[];
  private int priority;
  …
  private boolean daemon = false;
  …
  private Runnable target; /* What will be run. */

  public final static int MIN_PRIORITY = 1;
  …
  public static native Thread currentThread();
  public static native void sleep(long millis) throws InterruptedException;
  public static void sleep(long millis, int nanos)throws InterruptedException {…}
  private void init(ThreadGroup g, Runnable target, String name){…}

  public Thread() {
    init(null, null, "Thread-" + nextThreadNum());
  }

  public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum());
  }

  public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum());
  }

  public Thread(String name) {
    init(null, null, name);
  }
```

```
  public Thread(ThreadGroup group, String name) {
    init(group, null, name);
  }

  public Thread(Runnable target, String name) {
    init(null, target, name);
  }

  public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name);
  }

  public synchronized native void start() {…}

  public void run() {
    if (target != null) {
      target.run();
    }
  }

  private void exit() {…}

  public final void stop() {…}

  public final synchronized void stop(Throwable o) {…}
  public void interrupt() {…}
  public static boolean interrupted() {…}
  public boolean isInterrupted() {…}
  public void destroy() {…}
  public final native boolean isAlive() {…}
  public final void join() throws InterruptedException { … }
  public final void suspend() {…}
  public final void resume() {…}
  public final void setPriority(int newPriority) {…}
  public final int getPriority() {return priority;}
  public final void setName(String name) {…}
  public final String getName() {…}
  public final ThreadGroup getThreadGroup() {return group;}
  …
  public final void setDaemon(boolean on) {…}
  public final boolean isDaemon() {return daemon;}
  …
  public String toString() {…}
  …
}
```

Quelques remarques :

- On retrouve la plupart des méthodes classiques de gestion des threads : **start()**, **stop()**, **join()**, **exit()**, …
- Un thread possède notamment un nom, une priorité d'exécution et le fait d'être démon ou pas
- Les méthodes **natives** dialoguent directement avec le système d'exploitation hôte → elles sont écrites en C/C++

Attention ! Il ne faut pas confondre

- La classe Thread ou un objet instanciant la classe Thread et,
- Le sous-processus (le thread) en cours d'exécution représenté par cet objet.

Instancier un objet de classe Thread n'est pas lancer un thread (un sous-processus) !

La vie d'un thread en Java, c'est

- exécuter sa méthode **run(),** c'est-à-dire
- exécuter la méthode **run()** de sa variable membre **target** (objet implémentant l'interface **Runnable**.

Comment créer/lancer un thread ?

Il y a deux méthodes pour créer et lancer un thread :

1. Créer une **classe** héritant de la classe Thread, instancier un objet de cette classe et lancer le sous-processus en utilisant la méthode start().
2. Créer un objet (peu importe la classe) implémentant l'interface **Runnable**, instancier un objet de la classe Thread en utilisant un des constructeurs prenant en paramètre un Runnable, lancer le sous-processus en utilisant la méthode start().

Dans les deux cas,

- Le sous-processus démarre par l'appel de la méthode **start()**
- La vie du thread ainsi créé est l'exécution de sa méthode **run()**, il s'arrête quand cette fonction est terminée

Exemple de création d'un thread (1ère méthode)

Dans l'exemple qui suit, on crée une classe MonThread qui hérite de la classe Thread. Le thread principal instancie alors deux objets de cette classe et lance les 2 threads correspondant.

Voici la classe **MonThread** (**MonThread.java**) :

```
public class MonThread extends Thread
{
    private String message;
    private int    duree;

    public MonThread(String m,int d)
    {
        message = m;
        duree = d;
    }

    @Override
    public void run()
    {
        System.out.println("Thread secondaire (" + message + ") démarre...");
        try
        {
            sleep(duree);
        }
        catch (InterruptedException ex) { }
        System.out.println("Thread secondaire (" + message + ") se termine.");
    }
}
```

On remarquera au passage l'utilisation de la méthode statique **sleep()** dont le traitement de l'exception doit se faire dans la méthode run().

Voici le programme (**TestCreationThread1.java**) :

```
public class TestCreationThread1
{
    public static void main(String args[])
    {
        MonThread th1 = new MonThread("wagner",5000);
        MonThread th2 = new MonThread("charlet",3000);

        System.out.println("Lancement des threads secondaires...");
        th1.start();
        th2.start();

        System.out.println("Fin thread principal.");
    }
}
```

dont un exemple d'exécution fournit

```
# java Bases.TestCreationThread1
Lancement des threads secondaires...
Thread secondaire (wagner) démarre...
Fin thread principal.
Thread secondaire (charlet) démarre...
Thread secondaire (charlet) se termine.
Thread secondaire (wagner) se termine.
#
```

On remarque que

- Il n'y aucune synchronisation, les threads s'exécutent indépendamment les uns des autres sans s'attendre
- Le thread principal se termine avant les threads secondaires sans provoquer l'arrêt du processus

Exemple de création d'un thread (2ème méthode)

Dans l'exemple qui suit, on crée une classe **MaClasseAThreader** qui implémente l'interface **Runnable**. Le thread principal instancie alors deux objets de cette classe, puis deux objets de la classe **Thread** et lance les 2 threads correspondant.

Voici la classe **MaClasseAThreader** (**MaClasseAThreader.java**) :

```
public class MaClasseAThreader implements Runnable
{
    private String message;
    private int duree;

    public MaClasseAThreader(String m,int d)
    {
        message = m;
        duree = d;
    }

    @Override
    public void run()
    {
        System.out.println("Thread secondaire (" + message + ") démarre...");
        try
        {
            Thread.sleep(duree);
        }
        catch (InterruptedException ex) { }
        System.out.println("Thread secondaire (" + message + ") se termine.");
    }
}
```

Voici le programme (**TestCreationThread2.java**) :

```
public class TestCreationThread2
{
    public static void main(String args[])
    {
        MaClasseAThreader tache1 = new MaClasseAThreader("wagner",5000);
        MaClasseAThreader tache2 = new MaClasseAThreader("charlet",3000);
```

```
        Thread th1 = new Thread(tache1);
        Thread th2 = new Thread(tache2);

        System.out.println("Lancement des threads secondaires...");
        th1.start();
        th2.start();

        System.out.println("Fin thread principal.");
    }
}
```

dont l'exécution est exactement la même que dans l'exemple précédent.

Une première synchronisation : l'attente de la fin d'un thread

Pour cela, la classe **Thread** dispose de la méthode d'instance **join()**.

Exemple de l'attente de la fin d'un thread

Nous reprenons la classe **MonThread** de l'exemple ci-dessus. Le thread principal crée toujours deux threads secondaires mais va maintenant se synchroniser sur la fin de ces deux threads.

Le code de la classe **MonThread** est le même et voici le programme (**TestJoin.java**) :

```
public class TestJoin
{
    public static void main(String args[]) throws InterruptedException
    {
        MonThread th1 = new MonThread("wagner",5000);
        MonThread th2 = new MonThread("charlet",3000);

        System.out.println("Lancement des threads secondaires...");
        th1.start();
        th2.start();

        System.out.println("Attente de la fin des threads secondaires...");
        th1.join();
        th2.join();

        System.out.println("Fin thread principal.");
    }
}
```

dont un exemple fournit

```
# java SynchroJoin.TestJoin
Lancement des threads secondaires...
Thread secondaire (wagner) démarre...
Attente de la fin des threads secondaires...
Thread secondaire (charlet) démarre...
Thread secondaire (charlet) se termine.
Thread secondaire (wagner) se termine.
Fin thread principal.
#
```

Interrompre un thread

Il s'agit ici d'interrompre (arrêter) un thread en cours d'exécution, la demande venant d'un autre thread du même processus.

Mais,

- La méthode **stop()** est deprecated
- En effet, cette méthode stoppe brutalement le thread → si celui-ci est en train d'exécuter une section critique (voir plus loin), cela laisse peut laisser le processus dans une situation incohérente et/ou provoquer un deadlock

Dès lors,

- On utilise la méthode **interrupt()** qui a pour effet de lancer une exception instance de la classe **InterrupedException**
- Cette exception est captée et traitée par le thread visé qui peut alors agir en conséquence
- Le thread visé peut utiliser une variable membre reflétant son état, cette variable sera alors mise à jour dans le handler de l'exception → il pourra alors se terminer (ou non !) après avoir proprement terminé ce qu'il a à faire

Exemple d'interruption d'un thread

Dans cet exemple, le thread principal lance un thread instance de la classe **ThreadDormeur** qui tourne dans une boucle infinie dans laquelle il exécute la méthode **sleep**. Au bout d'un certain temps, le thread principal va interrompre ce thread.

Le code du **ThreadDormeur** (**ThreadDormeur.java**) est

```
public class ThreadDormeur extends Thread
{
    private boolean interrompu;

    @Override
    public void run()
    {
        interrompu = false;
        System.out.println("Thread secondaire démarre...");
        while(!interrompu)
        {
            try
            {
                System.out.println("Thread secondaire attends 5 secondes...");
                sleep(5000);
            }
            catch (InterruptedException ex)
            {
                System.out.println("Thread secondaire est interrompu !");
                interrompu = true;
            }
        }
        System.out.println("Thread secondaire se termine.");
    }
}
```

On remarque

- La présence de la variable membre « **interrompu** » qui représente l'état du thread
- La méthode **sleep(5000)** est bloquante → en cas d'interruption, le thread continue son exécution dans le handler (catch) où il met sa variable **interrompu** à jour → ceci lui permet de sortir de sa boucle infinie et de terminer proprement sa méthode run()

Le programme (**TestInterreupt.java**) est

```
public class TestInterrupt
{
    public static void main(String args[]) throws InterruptedException
    {
        ThreadDormeur th = new ThreadDormeur();

        System.out.println("Lancement du thread secondaire...");
        th.start();

        Thread.sleep(12000);
        System.out.println("Interruption thread secondaire...");
        th.interrupt();

        System.out.println("Fin thread principal.");
    }
}
```

dont un exemple d'exécution fournit

```
# java TestInterrupt
Lancement du thread secondaire...
Thread secondaire démarre...
Thread secondaire attends 5 secondes...
Thread secondaire attends 5 secondes...
Thread secondaire attends 5 secondes...
Interruption thread secondaire...
Fin thread principal.
Thread secondaire est interrompu !
Thread secondaire se termine.
#
```

On observe que le thread secondaire a correctement terminé sa fonction **run()**

La synchronisation : les moniteurs et les méthodes synchronized

Il s'agit d'assurer qu'une ressource commune ne soit pas accédée simultanément par plusieurs threads → **sections critiques** / **exclusion mutuelle**

Pour cela, Java propose de déclarer une méthode quelconque « **synchronized** » :

Lorsqu'un thread invoque une méthode **synchronized** d'un objet, celui-ci est verrouillé : un autre thread qui invoque une méthode synchronized (la même ou une autre) du même objet restera bloqué jusqu'à ce que le verrou soit enlevé par la fin de l'exécution de la première méthode

Cela définit ce que l'on appelle un « **monitor** » :

Un « **monitor** » désigne un ensemble de données et de procédures tel qu'il ne peut y avoir, à un instant donné, qu'un seul thread (ou processus) qui y soit actif, autrement dit qui se sert d'une de ces procédures

On dit encore que le thread (ou processus) « **prend le monitor pour un objet donné** »

Exemple de synchronisation par utilisation d'un « monitor »

Dans l'exemple qui suit,

- Trois threads (« Client ») vont manipuler simultanément un objet instanciant la classe **Compte** qui possède une méthode de dépôt et une méthode de retrait sur ce compte. Ceux deux méthodes sont « **synchronized** » afin d'assurer que les threads ne modifient pas simultanément la variable membre privée « **solde** » de cet objet → L'objet instanciant la classe Compte est le « **monitor** »
- Les 3 threads vont de manière aléatoire déposer/retirer une somme générée aléatoirement également
- Chaque thread client réalisera au total 2 opérations sur le compte

Voici le code du « **monitor** » (fichier **Compte.java**) :

```
public class Compte
{
    private int solde;

    public Compte()
    {
        solde = 0;
    }

    public synchronized void depot(int valeur)
    {
        System.out.println("Thread Client (" + Thread.currentThread().getName() + " -
LOCK) veut déposer " + valeur);
        try { Thread.sleep(1000); } catch (InterruptedException ex) { }
        solde += valeur;
        System.out.println("    Depot --> Solde actualisé = " + solde);
        System.out.println("Thread Client (" + Thread.currentThread().getName() + " -
UNLOCK) a terminé son dépôt");
    }

    public synchronized int retrait(int valeur)
    {
        System.out.println("Thread Client (" + Thread.currentThread().getName() + " -
LOCK) veut faire un retrait de " + valeur);
        try { Thread.sleep(1000); } catch (InterruptedException ex) { }
        int montant;
        if (solde >= valeur)
        {
            montant = valeur;
            solde -= montant;
        }
        else
        {
            montant = solde;
            solde = 0;
        }
        System.out.println("    Retrait --> Solde actualisé = " + solde);
        System.out.println("Thread Client (" + Thread.currentThread().getName() + " -
UNLOCK) a retiré " + montant);
        return montant;
    }
}
```

On remarque

- L'utilisation de la méthode de classe **currentThread()** permettant de récupérer la référence du thread en cours
- L'utilisation de la méthode d'instance **getName()** permettant de récupérer le nom du thread

Voici le code des threads clients (fichier **ThreadClient.java**) :

```
public class ThreadClient extends Thread
{
    private Compte compte;

    public ThreadClient(Compte c)
    {
        compte = c;
    }

    @Override
    public void run()
    {
        for(int i=0 ; i<2 ; i++)
        {
            int montant = (int) (Math.random() * 1000);
            if (Math.random() > 0.333)
            {
                compte.depot(montant);
            }
            else
            {
                int retrait = compte.retrait(montant);
            }
            Thread.yield();
        }
    }
}
```

On remarque que tous les threads clients dispose d'une référence vers l'objet **compte** qui est unique et commun à tous les threads, ainsi que l'utilisation de la méthode de classe **yield()** qui permet à un thread de « lâcher » le processeur au profit d'un autre thread.

Voici le code du programme de test (fichier **TestCompte.java**) :

```
public class TestCompte
{
    public static void main(String args[])
    {
        Compte compte = new Compte();

        ThreadClient th1 = new ThreadClient(compte);
        ThreadClient th2 = new ThreadClient(compte);
        ThreadClient th3 = new ThreadClient(compte);

        th1.start();
        th2.start();
        th3.start();

        System.out.println("Thread principal (" + Thread.currentThread().getName() +
") se termine.");
    }
}
```

dont un exemple d'exécution fournit :

```
# java TestCompte
Thread Client (Thread-0 - LOCK) veut déposer 410
Thread principal (main) se termine.
    Depot --> Solde actualisé = 410
Thread Client (Thread-0 - UNLOCK) a terminé son dépôt
Thread Client (Thread-2 - LOCK) veut déposer 292
    Depot --> Solde actualisé = 702
Thread Client (Thread-2 - UNLOCK) a terminé son dépôt
Thread Client (Thread-1 - LOCK) veut faire un retrait de 485
    Retrait --> Solde actualisé = 217
Thread Client (Thread-1 - UNLOCK) a retiré 485
Thread Client (Thread-2 - LOCK) veut déposer 39
    Depot --> Solde actualisé = 256
Thread Client (Thread-2 - UNLOCK) a terminé son dépôt
Thread Client (Thread-0 - LOCK) veut faire un retrait de 256
    Retrait --> Solde actualisé = 0
Thread Client (Thread-0 - UNLOCK) a retiré 256
Thread Client (Thread-1 - LOCK) veut faire un retrait de 234
    Retrait --> Solde actualisé = 0
Thread Client (Thread-1 - UNLOCK) a retiré 0
#
```

On remarque bien que lorsqu'un thread exécute une des méthodes **synchronized**, aucun des autres threads ne peut exécuter une de ces méthodes en même temps.

Remarques

- Les moniteurs de Java sont **réentrants** : lorsqu'un thread exécute une méthode synchronized, il peut sans problème appeler une autre méthode synchronized du même moniteur → ont dit que le thread « **possède le moniteur** »

- Une **méthode statique** d'une classe peut être **synchronized** → le verrou implicite ainsi créé n'a aucun effet sur les objets instanciant cette classe

- L'attribut synchronized n'est pas hérité

La synchronisation : l'attente d'un événement

Notons que les méthodes synchronized

- et les moniteurs assurent qu'un seul thread à la fois n'accède à une section critique → elles permettent l'équivalent de ce que l'on peut faire avec les **mutex** Posix (**pthread_mutex_t**)
- ne permettent pas la surveillance atomique et **l'attente de la réalisation d'un événement** → il manque pour cela la notion de **variable de condition** de la norme Posix (**pthread_cond_t**)

Pour cela, Java fournit **deux méthodes dérivées de la classe Object appelées obligatoirement au sein d'une méthode synchronized**, donc au sein d'un monitor :

- **public final void wait()** → place le thread appelant en attente d'une notification par un autre thread → le moniteur est automatiquement libéré
- **public final native void notify()** → choisit un thread en attente sur le monitor, le réveille, lui rendant automatiquement ce monitor → s'il n'y a aucun thread en attente, la notification est perdue

Il existe également une version avec « **Time Out** » de la méthode **wait** :

- **public final void wait(long milli)** → même comportement que la méthode wait() sauf qu'elle se débloque automatiquement au bout de milli millisecondes

Exemple d'attente d'un événement sans time out

Dans l'exemple qui suit,

- Un objet de classe **Conteneur** fait office de moniteur : il contient un **nombre** ainsi qu'un booléen « **vide** » indiquant si ce conteneur est vide ou a été déjà rempli par un thread
- Les méthodes synchronized **setValeur()** et **getValeur()** du moniteur attendent respectivement que le conteneur soit vide, ou qu'il contienne une valeur à retirer
- Deux threads « **Producteurs** » vont à intervalle de temps aléatoire introduire un nombre (aléatoire également) dans le conteneur, et se terminer au bout de 3 insertions chacun

- Trois threads « **Consommateurs** » vont se mettre en attente sur « Tant que le conteneur est vide, j'attends ». Une fois réveillé, ils extraient la valeur contenue et l'affichent avant de se remettre en attente du même événement

Voici le code du **moniteur** (fichier **Conteneur.java**) :

```
public class Conteneur
{
    private int valeur;
    private boolean vide;

    public Conteneur()
    {
        vide = true;
    }

    public synchronized void setValeur(int v) throws InterruptedException
    {
        while(!vide) wait();
        valeur = v;
        vide = false;
        notify();
    }

    public synchronized int getValeur() throws InterruptedException
    {
        while(vide) wait();
        vide = true;
        int tmp = valeur;
        notify();
        return tmp;
    }
}
```

Le code d'un thread « **Producteur** » (fichier **Producteur.java**) est

```
public class Producteur extends Thread
{
    private Conteneur conteneur;

    public Producteur(Conteneur c)
    {
        conteneur = c;
    }

    @Override
    public void run()
    {
        for (int i=0 ; i<3 ; i++)
        {
```

```
            try
            {
                Thread.sleep((long) (Math.random()*6000));
                int nombre = (int) (Math.random()*10);
                System.out.println("Producteur (" + getName() + ") veut déposer " +
nombre + " dans le conteneur");
                conteneur.setValeur(nombre);
                System.out.println("Producteur (" + getName() + ") a terminé son
dépôt.");
            }
            catch (InterruptedException ex) { }
        }
    }
}
```

Tandis que le code d'un thread « **Consommateur** » (fichier **Consommateur.java**) est

```
public class Consommateur extends Thread
{
    private Conteneur conteneur;

    public Consommateur(Conteneur c)
    {
        conteneur = c;
    }

    @Override
    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("Consommateur (" + getName() + ") attend une
valeur...");
                int nombre = conteneur.getValeur();
                System.out.println("Consommateur (" + getName() + ") a extrait la
valeur " + nombre);
            }
            catch (InterruptedException ex) { }
        }
    }
}
```

Le code du programme de test (fichier **TestMoniteurWait.java**) est

```
public class TestMoniteurWait
{
    public static void main(String args[]) throws InterruptedException
    {
        Conteneur conteneur = new Conteneur();
```

```
        new Consommateur(conteneur).start();
        new Consommateur(conteneur).start();
        new Consommateur(conteneur).start();

        Producteur p1 = new Producteur(conteneur);
        Producteur p2 = new Producteur(conteneur);
        p1.start();
        p2.start();

        p1.join();
        p2.join();

        System.exit(0);
    }
}
```

On remarque que

- Les 3 threads consommateurs sont créés et lancés à la volée
- Le thread principal attend la fin des 2 threads producteurs avant de terminer le processus par l'appel de **System.exit(0)** → sans cela, le processus ne s'arrêterait jamais car les 3 threads consommateurs attendraient indéfiniment sur leur **wait()**

Un exemple d'exécution fournit

```
# java TestMoniteurWait
Consommateur (Thread-0) attend une valeur...
Consommateur (Thread-1) attend une valeur...
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-3) veut déposer 0 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-0) a extrait la valeur 0
Consommateur (Thread-0) attend une valeur...
Producteur (Thread-3) veut déposer 5 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 5
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-3) veut déposer 5 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-1) a extrait la valeur 5
Consommateur (Thread-1) attend une valeur...
Producteur (Thread-4) veut déposer 7 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-0) a extrait la valeur 7
Consommateur (Thread-0) attend une valeur...
Producteur (Thread-4) veut déposer 5 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 5
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-4) veut déposer 0 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-1) a extrait la valeur 0
Consommateur (Thread-1) attend une valeur...
#
```

Exemple d'attente d'un événement avec time out

Nous reprenons l'exemple précédent mais

- Un compteur de threads producteurs est modélisé par une **variable statique** dans la classe **Producteur**
- La méthode **getValeur()** du moniteur utilise la méthode **wait(long milli)** → au bout des 5000 millisecondes, si le conteneur est vide et que le nombre de threads producteurs est passé à 0, la méthode retourne -1
- Le thread principal ne doit plus attendre la fin des thread producteurs, ni arrêter le processus → les threads consommateurs s'arrêtent d'eux-mêmes lorsqu'il n'y a plus de producteur à l'issue du « **Time Out** »

Le code du thread « **Producteur** » devient (fichier **Producteur.java**) :

```
public class Producteur extends Thread
{
    private Conteneur conteneur;
    public static int nbProducteurs = 0;

    public Producteur(Conteneur c)
    {
        conteneur = c;
        nbProducteurs++;
    }

    @Override
    public void run()
    {
        for (int i=0 ; i<3 ; i++)
        {
            try
            {
                Thread.sleep((long) (Math.random()*6000));
                int nombre = (int) (Math.random()*10);
                System.out.println("Producteur (" + getName() + ") veut déposer " +
nombre + " dans le conteneur");
                conteneur.setValeur(nombre);
                System.out.println("Producteur (" + getName() + ") a terminé son
dépôt.");
            }
            catch (InterruptedException ex) { }
        }
        nbProducteurs--;
    }
}
```

Le code du **moniteur** devient (fichier **Conteneur.java**) :

```
public class Conteneur
{
    private int valeur;
    private boolean vide;

    public Conteneur()
    {
        vide = true;
    }

    public synchronized void setValeur(int v) throws InterruptedException
    {
        while(!vide) wait();
        valeur = v;
        vide = false;
        notify();
    }

    public synchronized int getValeur() throws InterruptedException
    {
        while(vide && Producteur.nbProducteurs > 0) wait(5000);
        if (vide && Producteur.nbProducteurs == 0) return -1;
        vide = true;
        int tmp = valeur;
        notify();
        return tmp;
    }
}
```

Le code du thread « **Consommateur** » (fichier Consommateur.java) devient

```
public class Consommateur extends Thread
{
    private Conteneur conteneur;

    public Consommateur(Conteneur c)
    {
        conteneur = c;
    }

    @Override
    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("Consommateur (" + getName() + ") attend une
valeur...");
                int nombre = conteneur.getValeur();
                if (nombre == -1)
```

```
                {
                    System.out.println("Consommateur (" + getName() + ") se termine
(plus de producteur)");
                    return;
                }
                System.out.println("Consommateur (" + getName() + ") a extrait la
valeur " + nombre);
            }
            catch (InterruptedException ex) { }
        }
    }
}
```

Le programme de test devient (**TestMoniteurWaitTimeOut.java**) :

```
public class TestMoniteurWaitTimeOut
{
    public static void main(String args[]) throws InterruptedException
    {
        Conteneur conteneur = new Conteneur();

        new Producteur(conteneur).start();
        new Producteur(conteneur).start();

        new Consommateur(conteneur).start();
        new Consommateur(conteneur).start();
        new Consommateur(conteneur).start();

        System.out.println("Thread principal (" + Thread.currentThread().getName() +
") se termine.");
    }
}
```

dont un exemple d'exécution fournit

```
# java TestMoniteurWaitTimeOut
Consommateur (Thread-2) attend une valeur...
Consommateur (Thread-3) attend une valeur...
Thread principal (main) se termine.
Consommateur (Thread-4) attend une valeur...
Producteur (Thread-1) veut déposer 6 dans le conteneur
Producteur (Thread-1) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 6
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-0) veut déposer 5 dans le conteneur
Producteur (Thread-0) a terminé son dépôt.
Consommateur (Thread-4) a extrait la valeur 5
Consommateur (Thread-4) attend une valeur...
Producteur (Thread-1) veut déposer 3 dans le conteneur
Producteur (Thread-1) a terminé son dépôt.
Consommateur (Thread-3) a extrait la valeur 3
Consommateur (Thread-3) attend une valeur...
```

```
Producteur (Thread-1) veut déposer 2 dans le conteneur
Producteur (Thread-1) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 2
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-0) veut déposer 9 dans le conteneur
Producteur (Thread-0) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 9
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-0) veut déposer 8 dans le conteneur
Producteur (Thread-0) a terminé son dépôt.
Consommateur (Thread-4) a extrait la valeur 8
Consommateur (Thread-4) attend une valeur...
Consommateur (Thread-4) se termine (plus de producteur)
Consommateur (Thread-2) se termine (plus de producteur)
Consommateur (Thread-3) se termine (plus de producteur)
#
```

Les threads démons

On peut distinguer deux catégories de threads :

1. Les « **user threads** » : ce sont en général les threads créés et lancés par le programmeur d'une application multi-threadée → le processus qui les contient ne se termine que lorsque tous ces threads se sont terminés
2. Les « **deamon threads** » : ce sont des threads en attente d'une demande de service d'autres threads, ils tournent généralement en boucle infinie

Un processus Java s'arrête lorsqu'elle ne contient plus que des deamon threads → en effet plus aucun thread ne pourrait leur demander un service

Il est possible de transformer un thread en **démon** en utilisant la méthode

- **public final void setDaemon(boolean on)** avec on = true

Exemple de threads consommateurs démons

Nous reprenons l'exemple ci-dessus « **Exemple d'attente d'un événement sans time out** ». Dans cet exemple, le thread principal devait attendre la fin des threads producteurs puis terminer le processus par l'appel de **System.exit(0)**, sans quoi le

processus ne se serait jamais arrêté vu que les 3 processus consommateurs étaient en attente d'une « demande de service ».

Nous conservons ici le code des threads producteurs, consommateur et celui du moniteur. Par contre, nous allons rendre les 3 threads consommateurs **démons**. Ainsi, lorsque les 2 threads producteurs auront disparu, le processus s'arrêtera automatiquement.

Voici le code du nouveau programme de test (**TestMoniteurWaitDaemon.java**) :

```
public class TestMoniteurWaitDaemon
{
    public static void main(String args[]) throws InterruptedException
    {
        Conteneur conteneur = new Conteneur();

        Consommateur c1 = new Consommateur(conteneur);
        Consommateur c2 = new Consommateur(conteneur);
        Consommateur c3 = new Consommateur(conteneur);

        c1.setDaemon(true); c1.start();
        c2.setDaemon(true); c2.start();
        c3.setDaemon(true); c3.start();

        Producteur p1 = new Producteur(conteneur);
        Producteur p2 = new Producteur(conteneur);
        p1.start();
        p2.start();
    }
}
```

dont un exemple d'exécution fournit :

```
# java TestMoniteurWaitDaemon
Consommateur (Thread-0) attend une valeur...
Consommateur (Thread-1) attend une valeur...
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-3) veut déposer 2 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-0) a extrait la valeur 2
Consommateur (Thread-0) attend une valeur...
Producteur (Thread-4) veut déposer 5 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 5
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-3) veut déposer 0 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-1) a extrait la valeur 0
Consommateur (Thread-1) attend une valeur...
Producteur (Thread-4) veut déposer 5 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-0) a extrait la valeur 5
Consommateur (Thread-0) attend une valeur...
```

```
Producteur (Thread-3) veut déposer 6 dans le conteneur
Producteur (Thread-3) a terminé son dépôt.
Consommateur (Thread-2) a extrait la valeur 6
Consommateur (Thread-2) attend une valeur...
Producteur (Thread-4) veut déposer 2 dans le conteneur
Producteur (Thread-4) a terminé son dépôt.
Consommateur (Thread-1) a extrait la valeur 2
Consommateur (Thread-1) attend une valeur...
#
```

On remarque que le processus se termine alors que les 3 threads consommateurs ne se sont pas terminés.

Les priorités

Les priorités des threads correspondant au « **temps processeur** » qui leur est attribué par le scheduler. Sans spécification, les threads d'une application Java ont une priorité définie par la constante **NORM_PRIORITY**

Il est néanmoins possible de leur attribuer une autre priorité à l'aide de leur méthode

- **public void setPriority(int newPriority)**

dont le paramètre newPriority peut prendre la valeur **MIN_PRIORITY** ou **MAX_PRIORITY**, constantes de la classe Thread.

Les threads de priorité supérieure auront un temps machine supérieur aux threads priorité inférieure.

Il est à noter que modifier les priorités des threads ne constituent pas un moyen de synchronisation mais plutôt d'optimisation.

Les groupes de threads

Il peut être pratique de répartir les threads d'un processus en **groupes** et **sous-groupes** afin d'interagir avec eux de manière groupée → l'intérêt est aussi la **sécurité** car un **thread ne peut modifier le comportement d'un thread appartenant à un autre groupe**

Il est à noter que

- un thread appartient toujours à un groupe
- par défaut, le thread principal appartient au groupe appelé « **main** »
- un groupe peut contenir un autre groupe → on peut aboutir à une hiérarchie de groupes de threads

Un groupe de threads est représenté par la classe **ThreadGroup** qui possède les constructeurs

- **public ThreadGroup(String name)**
- **public ThreadGroup(ThreadGroup parent,String name)**

et les méthodes :

- **public final String getName()** → retourne le nom du groupe
- **public int activeCount()** → retourne le nombre de threads du groupe au moment de l'appel
- **public int enumerate(Thread list[])** → remplit le tableau passé en paramètre avec tous les threads actifs dans le groupe et ses sous-groupes

Un thread peut connaître le groupe dans lequel il est grâce à sa méthode

- **public final ThreadGroup getThreadGroup()**

Exemple de création et d'analyse de groupes de threads

Dans l'exemple qui suit, le thread principal

- crée 3 threads qui feront partie du groupe courant, c'est-à-dire « **main** »
- crée un groupe de threads appelé « **MonGroupe** », ainsi que 2 threads qui en feront partie
- analyse le groupe courant, ainsi que le groupe « MonGroupe »

- interromp de manière « groupée » les threads des 2 groupes

Voici le code de classe **MonThread** utilisée (fichier **MonThread.java**) :

```
public class MonThread extends Thread
{
    public MonThread(String nom)
    {
        super(nom);
    }

    public MonThread(ThreadGroup tg,String nom)
    {
        super(tg,nom);
    }

    @Override
    public void run()
    {
        try
        {
            System.out.println("Je suis le thread " + getName() + " et je
m'endors...");
            while (true) sleep(2000);
        }
        catch (InterruptedException ex)
        {
            System.out.println("Je suis le thread " + getName() + " et je me termine
!");
        }
    }
}
```

Et voici le code du programme de test (**TestThreadGroup.java**) :

```
public class TestThreadGroup
{
    public static void main(String args[]) throws InterruptedException
    {
        new MonThread("TH1").start();
        new MonThread("TH2").start();
        new MonThread("TH3").start();

        // Création d'un groupe de threads
        ThreadGroup monGroupe = new ThreadGroup("MonGroupe");
        Thread th4 = new MonThread(monGroupe,"TH4");
        Thread th5 = new MonThread(monGroupe,"TH5");
        th4.start();
        th5.start();

        Thread.sleep(1000);

        // Analyse du groupe courant
        System.out.println("--- Groupe courant ---");
        ThreadGroup tg = Thread.currentThread().getThreadGroup();
```

```
        System.out.println("Groupe parent : " + tg.getParent().getName());
        System.out.println("Nom du groupe : " + tg.getName());
        System.out.println("Nb Threads    : " + tg.activeCount());

        Thread liste[] = new Thread[tg.activeCount()];
        tg.enumerate(liste);
        for (int i=0 ; i<liste.length ; i++)
            System.out.println("Thread : " + liste[i].getName());

        // Analyse du groupe créé
        System.out.println("--- Groupe créé ---");
        System.out.println("Groupe parent : " + monGroupe.getParent().getName());
        System.out.println("Nom du groupe : " + monGroupe.getName());
        System.out.println("Nb Threads    : " + monGroupe.activeCount());

        liste = new Thread[monGroupe.activeCount()];
        monGroupe.enumerate(liste);
        for (int i=0 ; i<liste.length ; i++)
            System.out.println("Thread : " + liste[i].getName());

        // Interruption des threads
        System.out.println("Thread principal interromp le groupe créé...");
        monGroupe.interrupt();
        Thread.sleep(1000);
        System.out.println("Thread principal interrompt le groupe courant...");
        Thread.currentThread().getThreadGroup().interrupt();
    }
}
```

dont un exemple d'exécution fournit :

```
# java TestThreadGroup
Je suis le thread TH1 et je m'endors...
Je suis le thread TH2 et je m'endors...
Je suis le thread TH3 et je m'endors...
Je suis le thread TH4 et je m'endors...
Je suis le thread TH5 et je m'endors...
--- Groupe courant ---
Groupe parent : system
Nom du groupe : main
Nb Threads    : 6
Thread : main
Thread : TH1
Thread : TH2
Thread : TH3
Thread : TH4
Thread : TH5
--- Groupe créé ---
Groupe parent : main
Nom du groupe : MonGroupe
Nb Threads    : 2
Thread : TH4
Thread : TH5
Thread principal interrompt le groupe créé...
Je suis le thread TH4 et je me termine !
```

```
Je suis le thread TH5 et je me termine !
Thread principal interrompt le groupe courant...
Je suis le thread TH1 et je me termine !
Je suis le thread TH2 et je me termine !
Je suis le thread TH3 et je me termine !
#
```

On remarque que les threads du sous-groupe « MonGroupe » font bien partie du groupe « main »

La communication par pipes

Deux ou plusieurs threads peuvent communiquer (s'envoyer des données) en utilisant un **tube de communication** (équivalent des pipes du C).

En Java, les pipes sont matérialisés par deux classes : **PipedInputStream** et **PipedOutputStream**, dérivées respectivement de InputStream et OutputStream. Ces deux classes représentent respectivement la sortie et l'entrée du pipe.

Leurs constructeurs sont

- **public PipedInputStream()**
- **public PipedInputStream(PipedOutputStrem src) throws IOException**

et

- **public PipedOutputStream()**
- **public PipedOutputStream(PipedInputStream snk) throws IOException**

Important :

Le point important est que ces flux pipes seront toujours instanciés par paire : l'un se connecte en fait sur l'autre, l'ordre n'a pas d'importance

Et

Ils seront toujours associés à deux threads, l'un d'écriture sur le canal et l'autre de lecture sur ce même canal

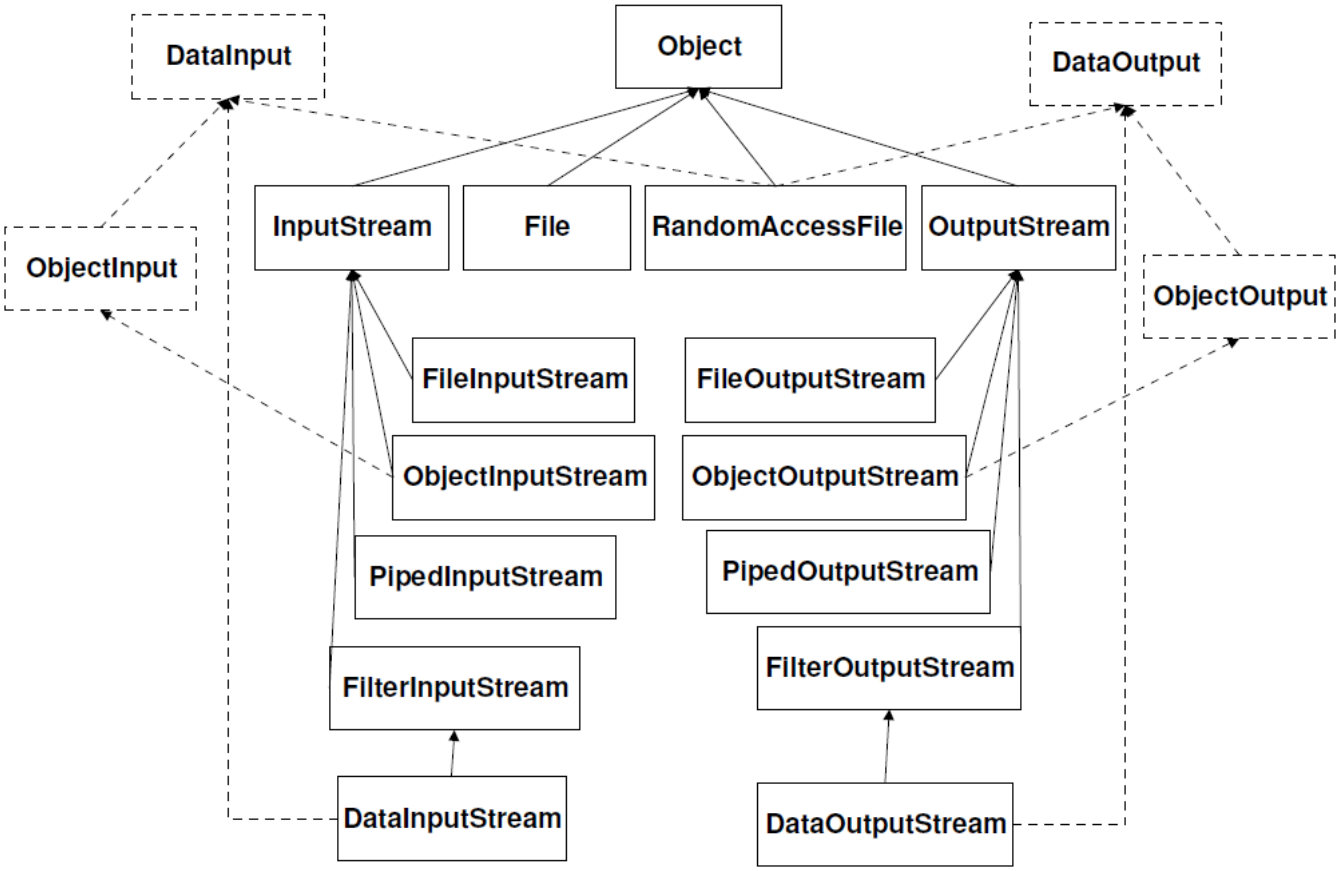
Ils possèdent respectivement les méthodes (bas niveau) :

- **public synchronized int read(byte b[],int off,int len) throws IOException**
- **public void write(byte b[],int off,int len) throws IOException**

mais l'idéal est de "monter" un flux haut niveau sur ces flux bas niveaux, comme par exemple

- **DataInputStream** et **DataOutputStream**
- **ObjectInputStream** et **ObjectOutputStream** (permettant la sérialisation)

Le schéma des flux peut être complété :



Exemple de communication par pipe

Dans l'exemple qui suit, le thread principal va créer

- un pipe de communication en instanciant un objet de la classe **PipedInputStream** et un autre de la classe **PipedOutputStream** et les « apparier » en passant le premier en paramètre au constructeur de l'autre
- un **ThreadEcrivain** qui va monter sur le flux reçu en paramètre un **DataOutputStream** sur lequel il va écrire 5 entiers de manière aléatoire
- un **ThreadLecteur** qui va monter sur le flux reçu en paramètre un **DataInputStream** sur lequel il va lire les entiers écrits par le ThreadEcrivain

Voici le code du thread écrivain (fichier **ThreadEcrivain.java**) :

```
import java.io.*;

public class ThreadEcrivain extends Thread
{
    private OutputStream os;

    public ThreadEcrivain(OutputStream x)
    {
        os = x;
    }

    @Override
    public void run()
    {
        DataOutputStream dos = new DataOutputStream(os);

        System.out.println("Thread Ecrivain (" + getName() + ") démarre...");

        for (int i=0 ; i<5 ; i++)
        {
            try
            {
                sleep((long) (Math.random()*5000));
                int nombre = (int) (Math.random()*10);
                System.out.println("Thread Ecrivain (" + getName() + ") écrit " +
nombre);
                dos.writeInt(nombre);
            }
            catch (InterruptedException ex) { }
            catch (IOException ex) { }
        }

        System.out.println("Thread Ecrivain (" + getName() + ") se termine.");
    }
}
```

Et le code du thread lecture (fichier **ThreadLecteur.java**) :

```
import java.io.*;

public class ThreadLecteur extends Thread
{
    private InputStream is;

    public ThreadLecteur(InputStream x)
    {
        is = x;
    }

    @Override
    public void run()
    {
        DataInputStream dis = new DataInputStream(is);

        System.out.println("Thread Lecteur (" + getName() + ") démarre...");

        for (int i=0 ; i<5 ; i++)
        {
            try
            {
                int nombre = dis.readInt();
                System.out.println("Thread Lecteur (" + getName() + ") a lu " +
nombre);
            }
            catch (IOException ex) { }
        }

        System.out.println("Thread Lecteur (" + getName() + ") se termine.");
    }
}
```

Voici le code du programme de test (fichier **TestPipe.java**) :

```
import java.io.*;

public class TestPipe
{
    public static void main(String args[]) throws IOException
    {
        PipedOutputStream pos = new PipedOutputStream();
        PipedInputStream pis = new PipedInputStream(pos);

        new ThreadLecteur(pis).start();
        new ThreadEcrivain(pos).start();
    }
}
```

dont un exemple d'exécution fournit :

```
# java TestPipe
Thread Lecteur (Thread-0) démarre...
Thread Ecrivain (Thread-1) démarre...
Thread Ecrivain (Thread-1) écrit 1
Thread Lecteur (Thread-0) a lu 1
Thread Ecrivain (Thread-1) écrit 8
Thread Lecteur (Thread-0) a lu 8
Thread Ecrivain (Thread-1) écrit 1
Thread Lecteur (Thread-0) a lu 1
Thread Ecrivain (Thread-1) écrit 3
Thread Ecrivain (Thread-1) écrit 8
Thread Ecrivain (Thread-1) se termine.
Thread Lecteur (Thread-0) a lu 3
Thread Lecteur (Thread-0) a lu 8
Thread Lecteur (Thread-0) se termine.
#
```

Les threads et AWT/Swing

Le but ici est de threader une application graphique **AWT/Swing** :

- Plusieurs threads différents pourraient simultanément accéder et manipuler le même composant graphique (bouton, champ de texte, liste, barre de progression, …) de la fenêtre graphique
- Les méthodes d'accès de ces différents composants devraient alors être **synchronized** → cela deviendrait vite lourd à gérer
- Donc, l'idée qui a été mise en place est qu'un processus AWT/Swing lance un **thread dédié à la gestion exclusive de l'interface graphique** → tout ce qui concerne le GUI doit être exécuté par ce thread → on appelle ce thread le thread « **dispatcher** »

Le thread « **dispatcher** »

- est une instance de la classe **EventDispatchThread** (qui hérite de **Thread**)
- porte le nom « **AWT-EventQueue-0** » et fait partie du groupe « **main** »
- tourne en boucle en attente d'un événement (clic de souris, touche du clavier, redimensionnement fenêtre, … mais également en provenance d'autres threads qui auraient « posté » une demande d'action sur le GUI) et agit en conséquence en « transmettant » l'événement (« Event ») aux « listeners » concernés

- exécute les demandes de manière séquentielle (ce qui permet de s'affranchir de méthode synchronized vu qu'il est le seul à manipuler le GUI)

Tous les événements en attente d'être exécutés par le thread « **dispatcher** » sont stockés dans une file d'attente instance de la classe **EventQueue**. Cette classe possède la méthode

- **public static void invokeLater(Runnable runnable)**

qui demande au thread dispatcher d'exécuter la méthode run() de l'objet Runnable reçu en paramètre lorsque la file aura été vidée des événements qui s'y trouvaient déjà.

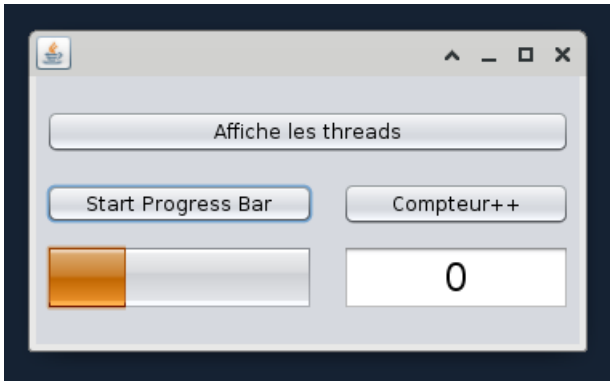
C'est donc ainsi qu'un thread quelconque peut faire exécuter une action sur le GUI par le thread dispatcher. Pour cela, il doit appeler la méthode

```
java.awt.EventQueue.invokeLater(…) ;
```

qui permet de récupérer une référence vers la file d'attente des événements et d'y insérer une demande d'action sur le GUI.

Exemple d'application Swing threadée

Voici un aperçu de l'application exemple considérée :



Dans cette application :

- un clic sur le bouton « **Compteur++** » incrémente de 1 le compteur situé juste en-dessous
- un clic sur le bouton « **Start Progress Bar** » créé et lance un thread qui va « augmenter » la barre de progression de 10% toutes les secondes
- un clic sur le bouton « **Affiche les threads** » permet d'afficher en console les différents threads présents dans l'application

Le code du thread qui souhaite manipuler la barre de progression (fichier **ThreadProgressBar.java**) est :

```
import javax.swing.JProgressBar;

public class ThreadProgressBar extends Thread
{
    JProgressBar pb;

    public ThreadProgressBar(JProgressBar pb)
    {
        this.pb = pb;
    }

    @Override
    public void run()
    {
        System.out.println("Méthode run() de ThreadProgressBar : je suis le thread "
+ Thread.currentThread().getName() + "...");
        for (int i=0 ; i<=100 ; i+=10)
        {
            try
            {
                java.awt.EventQueue.invokeLater(new ModifieProgressBar(pb,i));
                Thread.sleep(1000);
            }
            catch (InterruptedException ex) { }
        }
    }
}
```

On observe que le **ThreadProgressBar**

- possède une référence vers l'objet **pb** qu'il souhaite modifier mais il ne le modifie pas directement
- instancie un objet de la classe **ModifieProgressBar** (voir plus bas) et l'insère dans la file d'attente des événements du GUI

Le code de la classe **ModifieProgressBar** (fichier **ModifieProgressBar.java**) est :

```
import javax.swing.JProgressBar;

public class ModifieProgressBar implements Runnable
{
    JProgressBar pb;
    int valeur;

    public ModifieProgressBar(JProgressBar pb, int v)
    {
        this.pb = pb;
        valeur = v;
    }
```

```
    @Override
    public void run()
    {
        System.out.println("Méthode run() de ModifieProgressBar : je suis le thread "
+ Thread.currentThread().getName() + "...");
        pb.setValue(valeur);
    }
}
```

Cette classe correspond, par l'intermédiaire de sa méthode **run()**, à une action à réaliser sur le GUI, ici modifier la valeur du **JProgressBar**

Voici le code de **l'application GUI** (fichier **JFrameTest.java**) :

```
public class JFrameTest extends javax.swing.JFrame
{
    private int compteur;

    public JFrameTest() {
        initComponents();

        compteur = 0;
        jTextField1.setText(String.valueOf(compteur));
    }

    private void initComponents() {
        …
    }

    private void jButtonAfficheActionPerformed(java.awt.event.ActionEvent evt) {
        // Analyse du groupe courant
        System.out.println("Méthode jButtonAfficheActionPerformed : je suis le thread
" + Thread.currentThread().getName() + "...");

        System.out.println("--- Groupe courant ---");
        ThreadGroup tg = Thread.currentThread().getThreadGroup();

        System.out.println("Groupe parent : " + tg.getParent().getName());
        System.out.println("Nom du groupe : " + tg.getName());
        System.out.println("Nb Threads    : " + tg.activeCount());

        Thread liste[] = new Thread[tg.activeCount()];
        tg.enumerate(liste);
        for (int i=0 ; i<liste.length ; i++)
            System.out.println("Thread : " + liste[i].getName());

    }

    private void jButtonProgressBarActionPerformed(java.awt.event.ActionEvent evt) {
        System.out.println("Méthode jButtonProgressBarActionPerformed : je suis le
thread " + Thread.currentThread().getName() + "...");

        new ThreadProgressBar(jProgressBar1).start();
    }
```

```
    private void jButtonCompteurActionPerformed(java.awt.event.ActionEvent evt) {
        System.out.println("Méthode jButtonCompteurActionPerformed : je suis le
thread " + Thread.currentThread().getName() + "...");
        compteur++;
        jTextField1.setText(String.valueOf(compteur));
    }

    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
        …

        System.out.println("Méthode main : je suis le thread " +
Thread.currentThread().getName() + "...");

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new JFrameTest().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton jButtonAffiche;
    private javax.swing.JButton jButtonCompteur;
    private javax.swing.JButton jButtonProgressBar;
    private javax.swing.JProgressBar jProgressBar1;
    private javax.swing.JTextField jTextField1;
    // End of variables declaration
}
```

dont un exemple d'exécution fournit

```
# java JFrameTest
Méthode main : je suis le thread main...
Méthode jButtonCompteurActionPerformed : je suis le thread AWT-EventQueue-0...
Méthode jButtonProgressBarActionPerformed : je suis le thread AWT-EventQueue-0...
Méthode run() de ThreadProgressBar : je suis le thread Thread-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode jButtonAfficheActionPerformed : je suis le thread AWT-EventQueue-0...
--- Groupe courant ---
Groupe parent : system
Nom du groupe : main
Nb Threads    : 3
Thread : AWT-EventQueue-0
Thread : DestroyJavaVM
Thread : Thread-0
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode jButtonCompteurActionPerformed : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode jButtonCompteurActionPerformed : je suis le thread AWT-EventQueue-0...
```

```
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
Méthode run() de ModifieProgressBar : je suis le thread AWT-EventQueue-0...
[student@moon classes]$
```

On remarque que

- le thread principal ne vit que très peu de temps : le temps de créer un objet Runnable à la volée dans lequel la fenêtre est instanciée et affichée, et d'insérer cet objet dans la file d'attente du thread dispatcher
- Au moment du clic sur le bouton « Affiche les threads », 3 threads sont actifs
  - Le thread dispatcher "**AWT-EventQueue-0**"
  - Le thread "**DestroyJavaVM**" : il s'agit du garbage collector
  - Le thread "**Thread-0**" : le thread "utilisateur" (instance de la classe ThreadProgressBar)
- C'est systématiquement le **thread dispatcher** qui exécute les méthodes du GUI → c'est bien lui qui exécute la méthode **run()** de l'objet instanciant **ModifieProgressBar**

Les Timers

Dans de nombreuses applications, il est nécessaire que certaines tâches soient réalisées de manière régulière (répétitivement ou pas) en fonction de l'écoulement du temps :

- On pourrait imaginer créer manuellement un thread dédié à chacune de ces tâches et d'y planifier les moments d'exécution
- Java propose pour cela des classes appelées « **Timer** » dont les plus courantes sont celles des packages **java.util** et **javax.swing**

La classe Timer du package java.util

Cette classe permet de planifier des tâches dans un contexte plus général, que l'application soit graphique ou pas.

A chaque objet instanciant la classe **Timer** est associé un thread qui exécutera toutes les tâches associées à ce Timer.

La classe **Timer** présente plusieurs constructeurs (permettant de préciser le nom du thread ou encore le fait qu'il soit démon ou pas) mais le plus simple est d'utiliser son constructeur par défaut.

Une tâche est une instance dérivée de la classe abstraite **TimerTask** qui implémente l'interface **Runnable** et dont il faut redéfinir la méthode **run()** → la tâche à exécuter

Pour planifier les tâches, la classe **Timer** dispose des méthodes

- **public void schedule(TimerTask task, long delay)**
- **public void schedule(TimerTask task, Date time)**

qui permettent de préciser le délai d'attente ou la date d'exécution d'une tâche qui sera exécutée une seule fois.

Elle dispose encore des méthodes

- **public void schedule(TimerTask task, long delay,long period)**
- **public void schedule(TimerTask task, Date firstTime, long period)**

qui permettent en plus de préciser l'intervalle de temps entre 2 répétitions de la tâche qui sera donc exécutée indéfiniment à intervalle de temps régulier.

Exemple d'utilisation de la classe Timer de java.util

Dans l'exemple qui suit, le thead principal va créer 2 instances de la classe **Timer** et leur assigner à chacun 2 tâches, une à réaliser une seule fois et l'autre répétitivement.

Voici le code de la tâche à exécuter (fichier **TacheAExecuter.java**) :

```
import java.util.*;

public class TacheAExecuter extends TimerTask
{
    private String nom;

    public TacheAExecuter(String n)
    {
        nom = n;
    }

    @Override
    public void run()
```

```
    {
        System.out.println("[" + Thread.currentThread().getName() + "] J'exécute la
tâche " + nom + "...");
    }
}
```

et le programme de test (fichier **TestTimerUtil.java**) :

```
import java.util.*;

public class TestTimerUtil
{
    public static void main(String args[]) throws InterruptedException
    {
        Timer timer1 = new Timer();
        Timer timer2 = new Timer();

        timer1.schedule(new TacheAExecuter("OneShot1"),3500);
        timer1.schedule(new TacheAExecuter("Repetitif1"),1000,3000);
        timer2.schedule(new TacheAExecuter("OneShot2"),5200);
        timer2.schedule(new TacheAExecuter("Repetitif2"),6000,1500);

        Thread.sleep(10000);
        System.exit(0);
    }
}
```

dont un exemple d'exécution fournit

```
# java TestTimerUtil
[Timer-0] J'exécute la tâche Repetitif1...
[Timer-0] J'exécute la tâche OneShot1...
[Timer-0] J'exécute la tâche Repetitif1...
[Timer-1] J'exécute la tâche OneShot2...
[Timer-1] J'exécute la tâche Repetitif2...
[Timer-0] J'exécute la tâche Repetitif1...
[Timer-1] J'exécute la tâche Repetitif2...
[Timer-1] J'exécute la tâche Repetitif2...
#
```

On observe que 2 threads différents de nom « Timer-0 » et « Timer-1 » ont été créés et qu'ils s'occupent des tâches attribuées à leur objet Timer.

La classe Timer du package javax.swing

Cette classe **Timer** est d'un niveau d'abstraction supérieur :

- Elle utilise le mécanisme d'action « **Event-Listener** » de **AWT**
- Après chaque période de temps spécifiée, un **ActionEvent** est délivré à tous les **ActionListener** enregistrés par l'objet **Timer**
- L'objet **Timer** est en réalité un « **bean threadé** » qui émet des événements

La classe **Timer** dispose du constructeur

- **public Timer(int delay, ActionListener listener)**

où **delay** est le délai (en millisecondes) d'attente avant la première émission d'un ActionEvent, puis le délai entre deux émissions consécutives, et **listener** est le premier ActionListener enregistré dans la « mailing list » du Timer.

La classe **Timer** dispose en plus des méthodes

- **public void addActionListener(ActionListener listener)** → permet d'ajouter un autre listener à la « mailing list » du Timer
- **public void start()** → permet de faire démarrer le timer
- **public void stop()** → permet d'arrêter le timer
- **public void setDelay(int delay)** → permet de spécifier le temps entre 2 envois consécutifs d'événements
- **public void setRepeats(boolean flags)** → permet de spécifier que l'envoi d'événements soit répétitifs (true) ou pas (false)

Une fois le timer démarré,

- Un thread unique (un seul pour tous les objets instanciant la classe **Timer**) est créé et émet les événements ActionEvent
- Ce n'est pas ce thread qui exécute les méthodes **actionPerformed()** des **ActionListeners** mais bien le **thread « dispatcher »** → les événements émis sont donc placés dans la file d'attente des événements → cela assure que tous les événements émis par tous les timers seront traités de manière « thread-safe »

Exemple d'utilisation de la classe Timer de javax.swing

Voici un aperçu de l'application exemple considérée :

Dans cette application :

- Un clic sur le bouton « **Start** » fait démarrer un **Timer** qui va émettre des événements toutes les secondes
- Deux **listeners** sont enregistrés dans la « mailing list » du **Timer** :
  - La fenêtre elle-même qui va incrémenter le compteur visible dans la fenêtre graphique
  - Un objet instance de la classe **MyActionListener** qui se contente d'afficher un message en console
- Le bouton « **Stop** » permet d'arrêter le **Timer**, c'est-à-dire lui demande d'arrêter l'émission des événements

Le code de la classe **MyActionListener** (fichier **MyActionListener.java**) est

```
import java.awt.event.*;

public class MyActionListener implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("[" + Thread.currentThread().getName() + "] Dans
MyActionListener...");
    }
}
```

Tandis que le code de la fenêtre est (fichier **JFrameTimer.java**) est

```
import java.awt.event.*;
import javax.swing.Timer;

public class JFrameTimer extends javax.swing.JFrame implements ActionListener
{
    private int compteur;
    private Timer timer;
```

```
    public JFrameTimer()
    {
        initComponents();

        compteur = 0;
        jTextField1.setText(String.valueOf(compteur));

        timer = new Timer(1000,this);
        timer.addActionListener(new MyActionListener());
    }

    private void initComponents()
    {
      …
    }

    private void jButtonStartActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.out.println("[" + Thread.currentThread().getName() + "] Start !");
        timer.start();
    }

    private void jButtonStopActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.out.println("[" + Thread.currentThread().getName() + "] Stop !");
        timer.stop();
    }

    public static void main(String args[])
    {
        /* Set the Nimbus look and feel */
        …

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new JFrameTimer().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton jButtonStart;
    private javax.swing.JButton jButtonStop;
    private javax.swing.JTextField jTextField1;
    // End of variables declaration

    @Override
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("[" + Thread.currentThread().getName() + "] Dans
JFrameTimer...");
        compteur++;
        jTextField1.setText(String.valueOf(compteur));
    }
```

```
}
```

dont un exemple d'exécution fournit

```
# java JFrameTimer
[AWT-EventQueue-0] Start !
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Stop !
[AWT-EventQueue-0] Start !
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Dans MyActionListener...
[AWT-EventQueue-0] Dans JFrameTimer...
[AWT-EventQueue-0] Stop !
#
```

On remarque que

- Quelle que soit la méthode exécutée, c'est bien le **thread dispatcher** qui l'exécute.
- A chaque **ActionEvent** émis, tous les listeners enregistrés par le timer sont « mis au courant »