

JAVA

Applications « multithreadées »

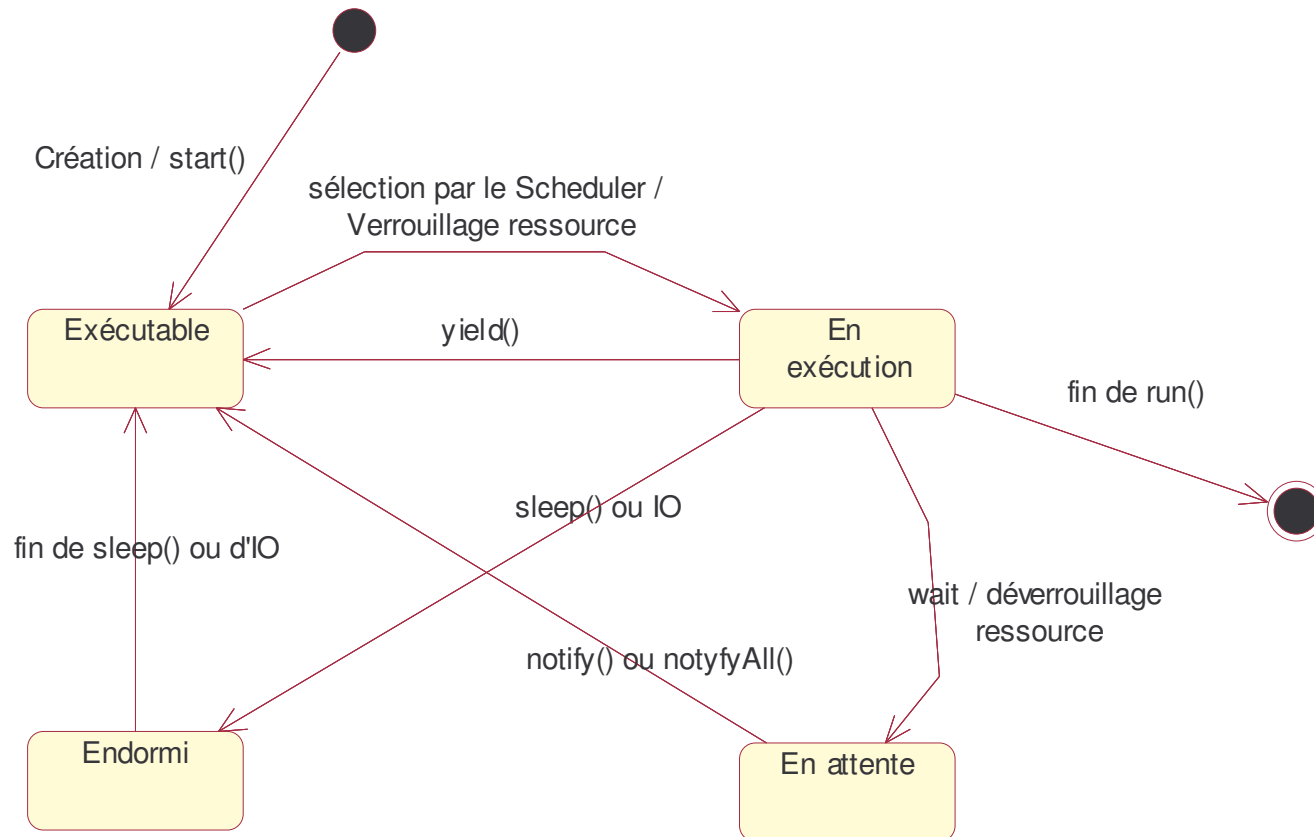
Principe

- Systèmes d'exploitation
 - Multitâche de processus
 - Chaque processus est un programme qui a son propre espace d'adressage
 - Communication interprocessus coûteuse

- Application JAVA
 - Multitâche de thread
 - Un Thread = une partie de l'application (un morceau de code)
 - Les thread partagent le même espace d'adressage
 - Communication interthread peu coûteuse
 - Permet d'exploiter au mieux le temps processeur alloué à l'application (saisie, chargement de fichier, etc.)
 - C'est la machine virtuelle qui gère l'ordonnancement des threads

- Attention : pas compatible 100% SWING, car la plupart des méthodes de SWING ne sont pas synchronisées

Les états d'un thread



L'interface Runnable

□ `abstract void run()`

- Contient le code qui doit s'exécuter en parallèle du reste de l'application
 - Le thread se termine lorsque `run()` se termine
-

La classe Thread

- ❑ Implémente Runnable
 - ❑ Constructeurs
 - **Thread(String nom)**
 - ❑ Construit un thread à partir de son nom
 - ❑ Nécessite la redéfinition de `run()` (qui ne fait rien par défaut)
 - **Thread(Runnable cible, String nom)**
 - ❑ Inutile de redéfinir `run()`
 - ❑ Appel automatiquement `cible.run()`
-

La classe Thread

- ❑ `static Thread currentThread()`
 - Renvoie le thread courant
 - ❑ `void start()`
 - Par défaut, appelle `run()`
 - Peut être redéfinie pour faire autre chose en plus
 - ❑ `static void sleep(int d)`
 - Fait une pause de `d` millisecondes
 - Libère le processus pour les autres threads
-

Définition d'un Thread

Deux moyens

■ **Implémentation de Runnable**

- Nécessite au minimum un constructeur et la redéfinition de run()

■ **Extension de Thread**

- Nécessite au minimum un constructeur et la redéfinition de run()
 - Permet de redéfinir start() entre autre...
 - Pas toujours possible si l'on étend déjà une autre classe
-

Exemple

- L'application Thread2
 - Crée 2 thread : ThreadNombre et ThreadAlphabet
 - Affiche « Princ : i » suivi d'une pause de 500ms
 - i varie de 0 à 5
 - ThreadNombre
 - Affiche « j » avec une pause de 100ms
 - j varie de 0 à 9
 - ThreadAlphabet
 - Affiche « L » avec une pause de 300ms
 - L varie de 'A' à 'H'
-

Grâce à la classe Thread

```
class ThreadNombre extends Thread
{
    public ThreadNombre()
    {
        super("Nombre");
        start();
    }

    public void run()
    {
        System.out.println("Thread Nombre : "+Thread.currentThread());
        for (int i=0; i<10; i++)
        {
            System.out.println(i);
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                System.out.println("ThreadNombre : Erreur");
            }
        }
        System.out.println("Thread Nombre terminé");
    }
}
```

Grâce à l'interface Runnable

```
class ThreadAlphabet implements Runnable
{
    public ThreadAlphabet() throws Exception
    {
        Thread t = new Thread(this, "Alphabet");
        t.start();
    }

    public void run()
    {
        System.out.println("Thread Alphabet : "+Thread.currentThread());
        for (int i=0; i<7; i++)
        {
            System.out.println((char)('A'+i));
            try {
                Thread.sleep(300);
            }
            catch (InterruptedException e)
            {
                System.out.println("ThreadAlphabet : Erreur");
            }
        }
        System.out.println("Thread Alphabet terminé");
    }
}
```

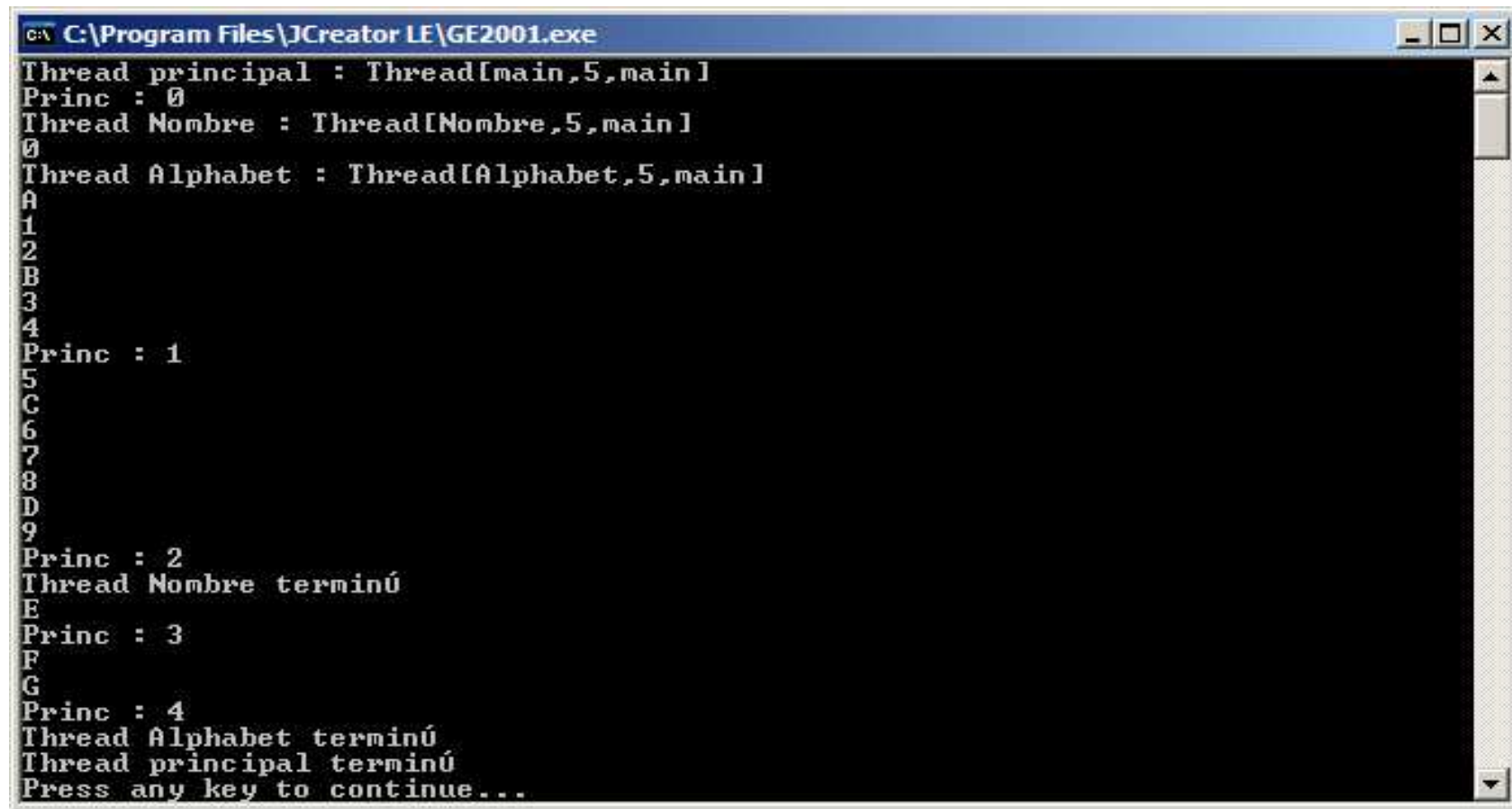
Le programme principal

```
public class Thread2
{
    public static void main(String [] args) throws Exception
    {
        System.out.println("Thread principal : "+Thread.currentThread());

        ThreadNombre N = new ThreadNombre();
        ThreadAlphabet A = new ThreadAlphabet();

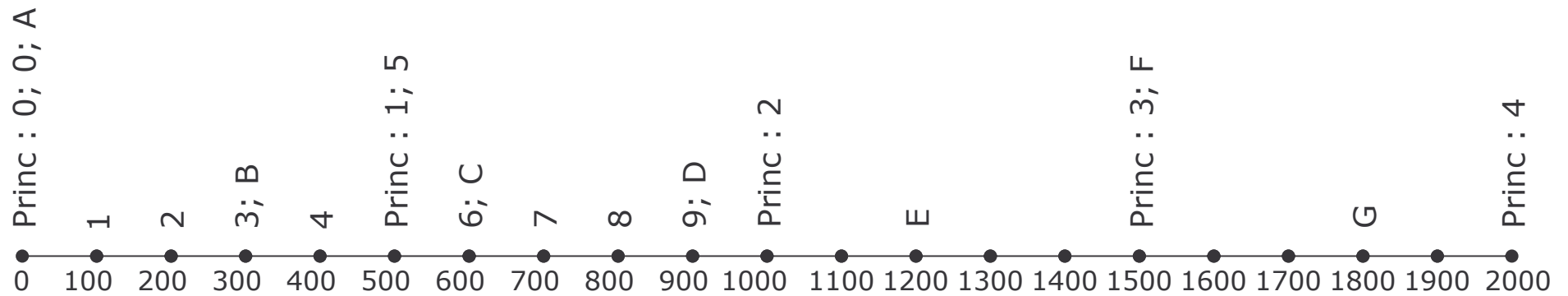
        for (int i=0; i<5; i++)
        {
            System.out.println("Thread principal : "+i);
            Thread.sleep(500);
        }
        System.out.println("Thread principal terminé");
    }
}
```

Exécution du programme



```
C:\Program Files\JCreator LE\GE2001.exe
Thread principal : Thread[main,5,main]
Princ : 0
Thread Nombre : Thread[Nombre,5,main]
0
Thread Alphabet : Thread[Alphabet,5,main]
A
1
2
B
3
4
Princ : 1
5
C
6
7
8
D
9
Princ : 2
Thread Nombre terminú
E
Princ : 3
F
G
Princ : 4
Thread Alphabet terminú
Thread principal terminú
Press any key to continue...
```

Exécution du programme



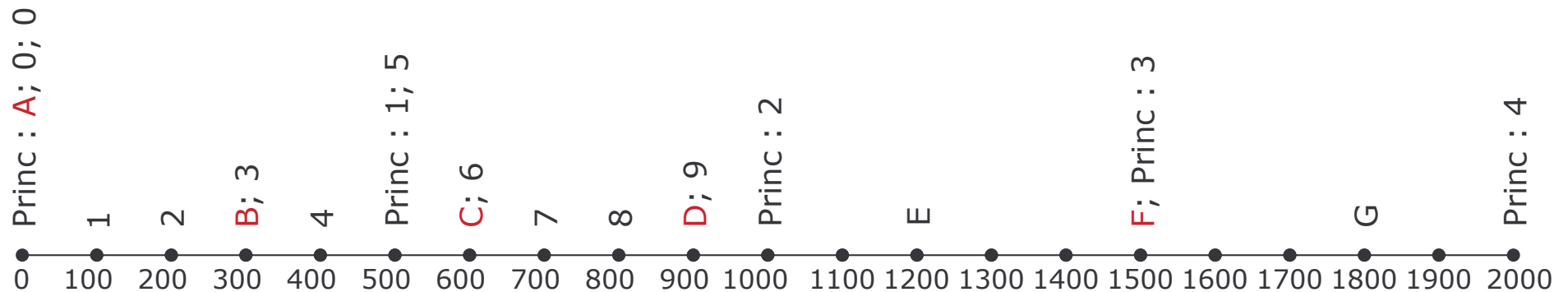
Priorité d'un thread

□ Classe Thread (suite)

- void **setPriority**(int P) : permet de modifier la priorité du thread
 - P compris entre MIN_PRIORITY et MAX_PRIORITY ([1,10])
 - NORM_PRIORITY (5) est la valeur par défaut
 - Les stratégies de sélection et la quantité de temps n'est pas accordée qu'en fonction de la priorité (dépend des machines virtuelles)
-

Exemple

```
public ThreadAlphabet() throws Exception
{
    Thread t = new Thread(this, "Alphabet");
    t.setPriority(Thread.MAX_PRIORITY);
    t.start();
}
```



isAlive() et join()

□ Classe Thread (suite)

- boolean **isAlive()** : renvoie false si le thread est terminé
 - void **join()** : attend que le thread cible soit terminé pour poursuivre
-

Exemple

```
for (int i=0; i<5; i++)  
{  
    System.out.println("Princ : "+i);  
    Thread.sleep(100);  
}
```

```
C:\Program Files\JCreator LE\GE2001.e  
Thread Alphabet : Thread[Alph  
A  
Princ : 1  
1  
Princ : 2  
2  
Princ : 3  
B  
3  
Princ : 4  
4  
Thread principal terminú  
5  
C  
6  
7  
8  
D  
9  
Thread Nombre terminú  
E  
F  
G  
Thread Alphabet terminú  
Press any key to continue...
```

```
for (int i=0; i<5; i++)  
{  
    System.out.println("Princ : "+i);  
    Thread.sleep(100);  
}  
N.join();
```

Pour se terminer,
le thread principal
attend que N soit
terminé

```
C:\Program Files\JCreator LE\GE20  
Thread Alphabet : Thread[Al  
A  
1  
Princ : 1  
2  
Princ : 2  
3  
B  
Princ : 3  
4  
Princ : 4  
5  
6  
C  
7  
8  
9  
D  
Thread Nombre terminú  
Thread principal terminú  
E  
F  
G  
Thread Alphabet terminú  
Press any key to continue...
```

Synchronisation

❑ Problème des accès concurrents

```
class ThreadAppelant extends Thread
{
    private Tableau T;

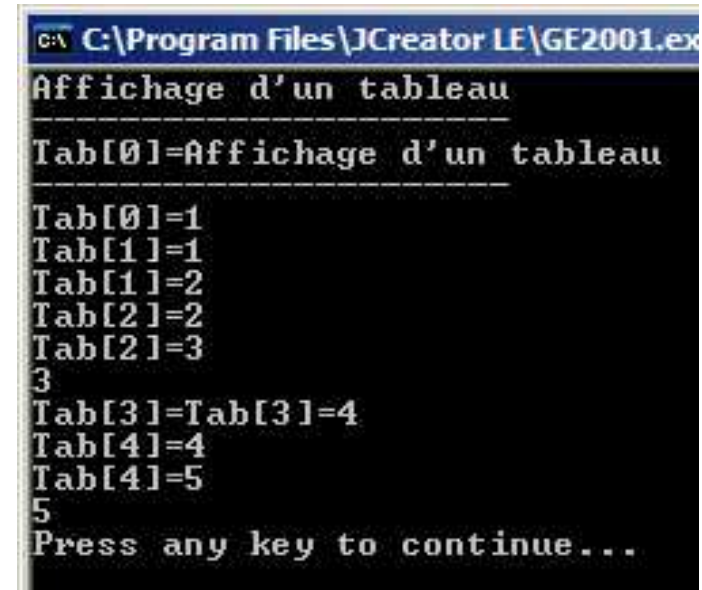
    public ThreadAppelant(Tableau T)
    {
        super("Appelle un Thread");
        this.T = T;

        start();
    }

    public void run()
    {
        T.Afficher();
    }
}

public class Thread3
{
    public static void main(String [] args) throws Exception
    {
        int [] Tab1 = {1,2,3,4,5};
        Tableau T = new Tableau(Tab1);

        ThreadAppelant T1 = new ThreadAppelant(T);
        ThreadAppelant T2 = new ThreadAppelant(T);
    }
}
```



```
C:\Program Files\JCreator LE\GE2001.exe
Affichage d'un tableau
-----
Tab[0]=Affichage d'un tableau
-----
Tab[0]=1
Tab[1]=1
Tab[1]=2
Tab[2]=2
Tab[2]=3
3
Tab[3]=Tab[3]=4
Tab[4]=4
Tab[4]=5
5
Press any key to continue...
```

T est partagé par les 2 threads
D'où l'affichage mélangé

Synchronisation

- Synchronisation de la fonction appelée
 - Dès que la fonction est appelée par un thread, les autres threads ne peuvent plus y accéder

```
public synchronized void Afficher()  
{  
    system.out.println("Affichage d'un tableau");  
    system.out.println("-----");  
  
    for (int i=0; i<Tab.length; i++)
```

- Synchronisation toujours prévue...



```
C:\Program Files\JCreator LE\G  
Affichage d'un tableau  
-----  
Tab[0]=1  
Tab[1]=2  
Tab[2]=3  
Tab[3]=4  
Tab[4]=5  
Affichage d'un tableau  
-----  
Tab[0]=1  
Tab[1]=2  
Tab[2]=3  
Tab[3]=4  
Tab[4]=5  
Press any key to continu
```

Synchronisation

- Synchronisation par l'appelant
 - La fonction de la ressource partagée n'est pas forcément synchronisée
 - C'est l'appelant qui doit s'occuper de la synchronisation
 - Tous les appelants doivent s'en occuper sinon cela ne sert à rien

```
synchronized (Ressource)
{
    ...
    Ressource.fonction(...);
    ...
}
```

```
public void run()
{
    synchronized (T)
    {
        T.Afficher();
    }
}
```

wait(), notify(), notifyAll()

- ❑ `wait()`, `notify()` et `notifyAll()` sont des méthodes de la class `Object`
 - ❑ Une fonction synchronisée peut attendre qu'une condition soit vérifiée pour se terminer
 - ❑ D'où, blocage de la ressource partagée
 - ❑ Appeler `wait()` dans la méthode de la ressource met le thread appelant en veille dans une *file d'attente* et libère la ressource
 - ❑ `notify()` dans une méthode de la ressource réveille au hasard un thread de la file d'attente de la ressource
 - ❑ `notifyAll()` les réveille tous
-

Exemple du producteur/consommateur

- Soit une Pile
 - C'est la ressource partagée
 - La taille de la pile est limitée
 - Dépiler (consommer) un élément n'est possible que si la pile n'est pas vide (d'où blocage définitif)
 - Empiler (produire) un élément n'est possible que si la pile n'est pas pleine (d'où blocage définitif)
 - Soit deux Threads, le consommateur et le producteur
 - Le consommateur demande une valeur et attend de la recevoir
 - Le producteur envoie une valeur et attend son insertion
-

Exemple

```
class Pile
{
    private final int taille = 10;
    private int [] Tab;
    private int nb;

    public Pile()
    {
        Tab = new int [taille];
        nb = 0;
    }

    public boolean EstPleine()
    {
        return nb == taille;
    }

    public boolean EstVide()
    {
        return nb == 0;
    }

    public synchronized void Empiler(int v)
    {
        try
        {
            while (EstPleine())
            {
                System.out.println("La pile est pleine");
            }
        }
    }
}
```

Thread et SWING

- Le travail réalisé dans un thread ne peut pas interférer avec l'interface SWING
 - Les méthodes SWING ne sont pas synchronisées
 - Possibilité de consulter des infos
 - Difficulté de mettre à jour des infos
 - EventQueue
 - Classe java.awt
 - Donne accès au thread de gestion des évènements (EventDispatchThread)
 - C'est lui qui déclenche l'exécution du « run() » quand le moment est venu :
static void **invokeLater**(Runnable runnable) (asynchrone)
static void **invokeAndWait**(Runnable runnable) (synchrone)
 - Les threads ne sont plus exécutés en parallèle des évènements de l'interface mais sont intégrés dans la file des évènements
-

Exemple

```
class MauvaisThread implements Runnable
{
    private DefaultListModel L;

    public MauvaisThread(DefaultListModel L)
    {
        this.L = L;
        Thread t = new Thread(this, "Mauvais");
        t.start();
    }

    public void run()
    {
        while (true)
        {
            Integer i = new Integer((int)(Math.random()*100));

            if (L.contains(i))
                L.removeElement(i);
            else
                L.addElement(i);

            Thread.yield();
        }
    }
}
```

Timer et TimerTask

- `import java.util.*;`

 - TimerTask
 - Surcharger la fonction void run()

 - Timer
 - `Timer()`
 - void schedule(TimerTask t, Date début, long period)
 - void schedule(TimerTask t, long délai, long period)
 - Period et délai en millisecondes

 - L'objet Timer exécute la méthode `run()` de l'objet TimerTask dans un thread
-