Interfaces Homme - Machine

SMI - S6
Année universitaire 2014-2015

Chapitre 6
Les threads en Java

Prof. M.D. RAHMANI
mrahmani@fsr.ac.ma

La programmation concurrente en Java

Plan:

- Introduction à la programmation concurrente,
- Les threads en Java,
  - Définition de threads en Java,
  - Déclaration, création et lancement d'un thread,
  - Terminaison d'un thread,
- Synchronisation,
  - Section critique,
  - Méthodes synchronisées
- Les Timers

Introduction à la programmation concurrente (1)

- Programmation standard:
  - Définition: Un *processus* représente l'environnement d'exécution d'un programme, les données et le code de l'application.
  - Un programme exécute des instructions séquentiellement.
  - Le développeur maîtrise la suite des opérations.
  - L'exécution est ***prévisible***.
  - L'exécution est ***reproductible***.

Introduction à la programmation concurrente (2)

- La programmation concurrente:
  - Un ***processus*** référence d'une part un espace mémoire pour stocker les données de l'application et d'autre part un ensemble de ***threads*** pour exécuter le code qui manipulera les données.
  - Un ***processus*** est décomposé en ***threads*** (*processus légers* ou *fils d'exécution*).
  - Un ***thread*** correspond à une tâche (plus en moins indépendant des autres) qui s'exécute.
- Problèmes:
  - Difficulté à synchroniser les tâches.
  - Gestion des ressources partagées.
  - L'exécution est ***imprévisible***.
  - Problème de ***reproductibilité***.

Introduction à la programmation concurrente (3)

- Programmation parallèle (ou répartie):
  - Des processus s'exécutent sur plusieurs processeurs.
- Programmation concurrente:
  - Les tâches sont gérées par un même processeur, éventuellement sur plusieurs cœurs.
- Les mécanismes de *synchronisation* sont différents.

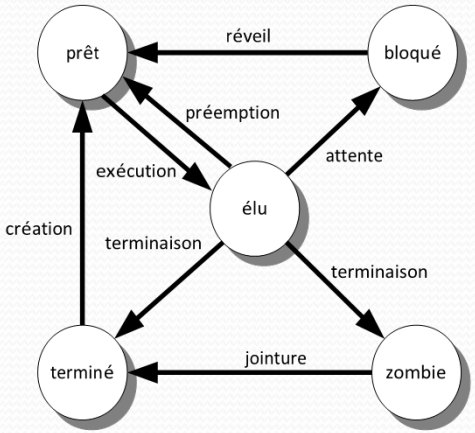
Introduction à la programmation concurrente (4)

- Anatomie d'un *processus*:
  - Un processus possède:
    - un code à exécuter,
    - un espace d'adressage *(stack segment, data segment, text segment* (code)),
    - une priorité,
    - un identifiant,
    - un contexte d'exécution
  - Les processus sont gérés par le système d'exploitation,
  - Plusieurs processus peuvent s'exécuter en parallèle.

Introduction à la programmation concurrente (5)

- Anatomie d'un *thread*:
  - Un thread est un fil d'exécution dans un processus:
    - les threads d'un même processus se partagent l'espace d'adressage du processus,
    - ils possèdent:
      - leur propre pile,
      - leur propre contexte d'exécution,
    - ils ont un cycle de vie semblable à celui d'un processus.

création ⟶ Prêt, Bloqué, Elu, Zombie, Terminé



Introduction à la programmation concurrente (6)

- Thread vs processus:
  - Commun:
    - sont des entités indépendantes, une fois créés,
    - les créateurs de processus et de threads ont le contrôle sur eux,
  - Non commun:
    - les processus ont un espace d'adressage; les threads non,
    - les processus enfants ne peuvent pas exercer de contrôle sur le processus parent; n'importe quel thread peut exercer un contrôle sur le thread principal et donc sur le processus entier.

Les threads en Java (1)

- Utilité des threads:
  - Puissance de modélisation: un monde multithreads,
  - Puissance d'exécution: parallélisme,
  - Légèreté grâce au partage des données,

- En Java, la gestion des threads est intégrée au langage:
  - Simplicité d'utilisation,
  - Les threads sont des objets: Classe définie dans le package: **`java.lang`**
  - Java 5.0 a intégré une bibliothèque étendue d'utilitaires pour la gestion de la concurrence: **`java.util.concurrent:`** *Pools de threads*, *queues*, *collections synchronisées*, *verrous spéciaux* ...

Les threads en Java (2)

- **Exemples:**
  - le langage Java est ***multi-threads***, c'est à dire qu'il peut exécuter du code à plusieurs endroits de façon indépendante.
  - dans un EDI comme Eclipse, un thread s'occupe de ce que tape l'utilisateur, un autre est en charge de souligner les erreurs, un 3ème de la coloration syntaxique, etc...
  - lors de la création d'un programme en Java, la ***JVM*** crée au moins un ***thread*** qu'elle affecte au **`main`** qui est le point d'entrée du programme.

Définition d'un thread Java

- La classe `java.lang.Thread` est au centre de la gestion des threads:
  - Les threads sont des objets:
    - Pour créer de nouveaux threads: `Thread()` et `Thread(Runnable cible)`
    - `void run()` // le corps d'un thread
      - `void start()` // lance l'exécution du thread, le thread s'exécute jusqu'à ce que // sa méthode `run` se termine,
      - `void interrupt()` // interrompt l'exécution du thread,
      - `static sleep(long millis)` // suspend l'exécution du thread,

Déclaration, création et lancement des threads (1)

En Java, un thread est représenté par une instance de la classe `java.lang.Thread`.

Le code qu'il doit exécuter est défini dans sa méthode `run()`, et un simple appel à la méthode `start()` permet de le démarrer.

```
public class NouveauThread {
   public static void main(String[] args) {
        Thread t = new Thread() { // on peut nommer le thread new Thread("Mon thread");
           public void run() {
                System.out.println("Je suis bien dans le thread: "+
                Thread.currentThread().getName()); // je suis dans le thread: Thread-0
           }
        };
        t.start(); // à tester avec t.run() donne je suis dans le thread: main
        System.out.println("Je suis dans le thread: "+
                Thread.currentThread().getName()); // je suis dans le thread : main
   }
}
```

Execution: "Je suis dans le thread: main, Je suis bien dans le thread: Thread-0"

Déclaration, création et lancement des threads (2)

- Comment créer un thread ?

  En Java, deux solutions sont possibles :

  - **Dériver de la class `Thread` :**

    On peut créer une nouvelle classe qui dérive de la classe `java.lang.Thread`. Il suffit ensuite de redéfinir la méthode `run()`. C'est cette méthode que le thread va exécuter.

  - **Implémenter l'interface `java.lang.Runnable` :**

    Au lieu d'une classe dédiée, on peut simplement implémenter l'interface `java.lang.Runnable` et redéfinir la méthode `run()`.

    Ensuite on appelle le constructeur de `Thread` avec pour argument un objet de l'interface `Runnable`.

  - **Dans tous les cas :**

    Pour lancer l'exécution d'un thread, on doit exécuter la méthode `start()` et en aucun cas exécuter nous-même la méthode `run()`.

    (L'exécution se déroulerait alors dans le processus courant !).

Création d'un thread par un héritage (1)

- Créer une classe qui hérite de la class `java.lang.Thread`
- Redéfinir la méthode `run()` pour y inclure les traitements à exécuter par le thread

```
class MonThread extends Thread {
    MonThread () {
            // code du constructeur
    }
    public void run () {
            // code à exécuter dans le thread
    }
}
```

- Pour créer et exécuter un tel thread, il faut instancier un objet et appeler sa méthode `start()` qui va créer le thread et elle-même appeler la méthode `run()`.

```
MonThread t = new MonThread ();
t. start ();
```

Création d'un thread par un héritage (2)

- Exemple:

Création d'un thread qui affiche 10 fois son nom et un entier croissant, puis s'endort pour une période comprise entre 0 et 1000 millisecondes

Création d'un thread par un héritage (3)

```
class ThreadSimple extends Thread {
      public ThreadSimple(String nom) {
          super(nom);
      }
      public void run() {
          for(int i=0; i<10; i++) {
             System.out.println(getName()+" "+i);
          try { // la méthode sleep doit toujours être appelée dans un bloc try...catch
             sleep((long) (Math.random() * 1000));
          } catch (InterruptedException e) {
          }
          System.out.println(getName() + " terminé");
      }}
 }
```

Remarques:

La méthode `sleep` doit être appelée dans un bloc `try...catch` car c'est par le lancement d'une exception `InterruptedException` que le thread endormi est réveillé si le temps indiqué est écoulé, il en est de même et pour la même raison, de la méthode `wait`.

Création d'un thread par un héritage (4)

Un programme qui fait tourner en parallèle 2 exemplaires de ce thread:

```
public class Test2Threads {
   public static void main(String[] args) {
       new ThreadSimple("Taza").start();
       new ThreadSimple("Casa Blanca").start();
   }
}
```

Remarque:

2 exécutions successives peuvent donner lieu à des imbrications "Taza" et "Casa Blanca" différentes!

Résultat:

Taza 0
Casa Blanca 0
Taza 1
Casa Blanca 1 Casa Blanca 2 Casa Blanca 3 ....

Création et démarrage d'un thread : l'interface `Runnable` (1)

- Déclarer une classe qui implémente l'interface `Runnable`
- Cette interface dispose d'une méthode : `run()`.
- Redéfinir sa seule et unique méthode `run()` pour y inclure les traitements à exécuter dans le thread.
- La classe Thread a un constructeur

```
new Thread(Runnable cible).
```

- L'argument du constructeur est donc toute instance de classe implémentant la méthode `run()`.

Création et démarrage d'un thread : l'interface `Runnable` (2)

- La classe se déclare comme dans l'exemple précédent, mais on implémente `Runnable` au lieu d'hériter de `Thread` :

```
class MonThread2 implements Runnable {
   MonThread2 () {
        //code du constructeur
   }
   public void run () {
        //code à exécuter dans le thread
   }
}
```

- Pour créer et lancer un thread, on crée d'abord une instance de `MonThread2`, puis une instance de `Thread` sur laquelle on appelle la méthode `start()` :

```
MonThread2 p = new MonThread2 ();
   Thread t = new Thread (p);
   t. start ();
```

Création et démarrage d'un thread : l'interface `Runnable` (3)

Exemple:

```
class MonThread2 implements Runnable {
    public void run () {
        System .out. println ("Je suis un thread !");
    }
    public static void main ( String args []) {
    // Un thread possède optionnellement un nom symbolique
        Thread t = new Thread (new MonThread2 (), "Mon Thread ");
    // MonThread.run () est démarré dans un nouveau thread après l'appel de start ()
        t. start ();
        System .out. println ("Ce code s'exécute en // de run ()");
    }
}
```

Choix: la classe `Thread` ou l'interface `Runnable`

	Avantages	**Incovénients**
extends java.lang.Threa	*Chaque thread a ses données* qui lui sont propres	On ne peut plus hériter d'une autre classe
implements java.lang.Runnable	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de la classe *sont partagés pour tous les threads qui y sont basés*. Dans certains cas, il peut s'avérer que cela soit un atout.

L'interface `Runnable` (1)

L'utilisation de la classe `java.lang.Thread` a le défaut de lier fortement la définition du traitement à exécuter (le "***quoi***") au thread particulier qui l'exécute ("***le comment***").

Pour pouvoir exécuter le même traitement par plusieurs threads ou relancer un traitement après la mort du thread associé, nous devons dissocier le "***quoi***" du "***comment***".

L'interface `java.lang.Runnable` permet d'encapsuler un traitement sous la forme d'un composant autonome et réutilisable.

Pour être réellement exécuté, un `Runnable` doit être passé en paramètre à un Thread ou un `ExecutorService`.

L'interface Runnable (2)

Exemple: un même Runnable est passé à 2 threads:

```
public class NouveauThreadAvecRunnable {
   public static void main(String[] args) {
        // le traitement encapsulé dans un Runnable
        Runnable traitement = new Runnable() { //#1
           public void run() {
                System.out.println("Je suis dans le thread: "
                   + Thread.currentThread().getName());
           }
        };
        Thread t1 = new Thread(traitement, "Premier thread"); //#2
        t1.start();
        Thread t2 = new Thread(traitement, "Second thread");
        t2.start();
        System.out.println("Je suis dans le thread: "
                + Thread.currentThread().getName());
   }
}
```

- En #2 nous passons le traitement à exécuter , encapsulé dans un Runnable en #1, à 2 threads.
- L'ordre d'exécution par le processeur des différents thread est imprévisible.

La classe `Thread`: propriétés et méthodes (1)

- `void destroy()` : met fin brutalement au thread.
- `int getPriority()` : renvoie la priorité du thread.
- `void setPriority(int)` : modifie la priorité d'un thread
- `ThreadGroup getThreadGroup()` : renvoie un objet qui encapsule le groupe auquel appartient le thread.
- `boolean isAlive()` : renvoie un booléen qui indique si le thread est actif ou non.
- `boolean isInterrupted()` : renvoie un booléen qui indique si le thread a été interrompu.
- `void start()` : démarrer le thread et exécuter la méthode `run()`.
- `currentThread()` : donne le thread actuellement en cours d'exécution.
- `setName()` : fixe le nom du thread.
- `getName()` : donne le nom du thread.

La classe `Thread`: propriétés et méthodes (2)

- `isAlive()` : indique si le thread est actif (`true`) ou non (`false`).
- `void resume()` : reprend l'exécution du `thread()` préalablement suspendu par `suspend()`.
- `void run()` : elle doit contenir le code qui sera exécuté par le thread.
- `void start()` : lance l'exécution d'un thread
- `void suspend()` : suspend le thread jusqu'au moment où il sera relancé par la méthode `resume()`.
- `void yield()` : indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter.
- `void sleep(long)` : mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type `InterruptedException` si le thread est réactivé avant la fin du temps.
- `void join()` : opération bloquante - attend la fin du thread pour passer à l'instruction suivante

La gestion de la priorité d'un `Thread` (1)

- La priorité du nouveau thread est égale à celle du thread dans lequel il est créé.
- Un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread,
- La priorité d'un thread varie de 1 à 10. La classe `Thread` définit trois constantes :
  - MIN_PRIORITY : priorité inférieure (0)
  - NORM_PRIORITY : priorité standard (5 : la valeur par défaut)
  - MAX_PRIORITY : priorité supérieure (10)
- Pour déterminer la priorité d'un thread on utilise la méthode `getPriority()` pour la modifier on utilise `setPriority(int)`.

Déclaration, création et lancement des threads (1)

Un programme qui affiche un cadre dont la barre de titre affiche constamment l'heure courante à la seconde près.

Pour cela, il suffit de créer, en même temps que le cadre, un 2ème thread qui dort la plupart du temps, se réveillant chaque seconde pour mettre à jour le texte affiché dans le cadre de titre:

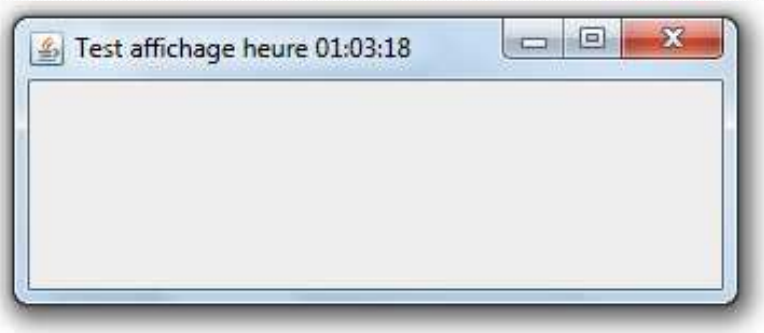
```
import java.text.DateFormat;
import java.util.Calendar;
import java.util.Date;

import javax.swing.JFrame;
public class CadreAvecHeure extends JFrame {
	private String titre;
```

Déclaration, création et lancement des threads (2)

```
public CadreAvecHeure(String titre) {
        super(titre);
        this.titre = titre;
        new Thread(new Runnable() {
            public void run() {
                gererAffichageHeure();
            }
        }).start(); // implémentation de Runnable
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 400);
        setVisible(true);
}
public void gererAffichageHeure(){
     while (true) {
        try {
                Thread.sleep(1000);
        }  catch(InterruptedException e) {
        }
```

Déclaration, création et lancement des threads (3)

```
                Date h = Calendar.getInstance().getTime();
                String s = DateFormat.getTimeInstance().format(h);
                setTitle(titre + " " + s);
            }
        }
        public static void main(String[] args){
            new CadreAvecHeure("Test affichage heure");
         }
}
```



Cycle de vie d'un `Thread`

Un thread en cours de traitement s'exécute jusqu'à ce qu'il soit:

- achevé, ou stoppé par un appel à la méthode `stop()`,
- ou en sortant de la méthode `run()`,
- ou interrompu pour passer la main par `yield()`,
- ou mis en sommeil par `sleep()`,
- ou désactivé temporairement par `suspend()` ou `wait()`.

Terminaison d'un thread (1)

Lorsqu'un thread atteint la fin de sa méthode `run()`, il se termine normalement et libère les ressources qui lui ont été allouées.

Mais pour provoquer une terminaison anticipée, la classe Thread comporte une méthode `stop()`, mais cette méthode est désapprouvée (*deprecated*) car elle risque de laisser dans un état indéterminé certains des objets qui dépendent du thread stoppé.

La manière propre et fiable de terminer un thread consiste à modifier la valeur d'une variable que le thread consulte régulièrement.

Par exemple, dans l'exemple précédent, nous allons ajouter une méthode `arreterPendule()` qui stoppe le rafraîchissement de l'heure affichée.

Terminaison d'un thread (2)

```
public class CadreAvecHeure2 extends JFrame {
   String titre;
   boolean go;
   public CadreAvecHeure2(String titre) {
        // comme avant
   }
   private void gererAffichageHeure2() {
        go = true;
        while(go) {
               try {
                   Thread.sleep(1000);
               } catch(InterruptedException e) {
               }
               Date d = Calendar.getInstance().getTime();
               String s = DateFormat.getTimeInstance().format(d);
               setTitle(titre + " " + s);
        }
   }
```

Terminaison d'un thread (3)

```
    public void arreterLaPendule() {
         go = false;
    }
    ...
}
```

Terminaison d'un thread (4)

```
import java.text.DateFormat;
import java.util.Calendar;
import java.util.Date;
import javax.swing.*;
public class CadreAvecHeure2 extends JFrame {
     String titre;
     static boolean go;
     public CadreAvecHeure2(String titre) {
        super(titre);
        this.titre = titre;
        new Thread(new Runnable() {
                public void run() {
                   gererAffichageHeure2();
                }
        }).start(); // implémentation de l'interface Runnable
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 400);
        setVisible(true);
    }
```

Terminaison d'un thread (5)

```
    private void gererAffichageHeure2() {
        go = true;
        while(go) {
           try {
                Thread.sleep(1000);
           } catch(InterruptedException e) {
           }
        Date d = Calendar.getInstance().getTime();
        String s = DateFormat.getTimeInstance().format(d);
        setTitle(titre + " " + s);
        }
    }
    public static void arreterLaPendule() {
        go = false;
    }
    public static void main(String[] args) {
        new CadreAvecHeure2("Test heure avec arrêt");
        arreterLaPendule();
    }
}
```

Synchronisation des threads (1)

- Plusieurs threads peuvent accéder à un objets concurrent (problème d'accès concurrent) => Introduction de la notion de **section critique**
- On ajoute le mot clé **`synchronized`** dans l'en-tête des méthodes.
- Ces méthodes servent à construire ce que l'on appelle des "***moniteurs***" c'est-à-dire des structures de données qui sont protégées de telle manière que seules des procédures qui travaillent en exclusion mutuelles puissent accéder aux objets.

Synchronisation des threads (2)

- Plusieurs threads veulent accéder à un compte en banque

=> utiliser la commande `synchronized` qui fait en sorte que les méthodes soient exécutées en exclusion mutuelle.

```
public class Compte
  int solde = 0;
  public synchronized void deposer (int s) {
       int so = solde + s;
       solde = so;
  }
  public synchronized void retirer (int s) {
       int so = solde - s;
       solde = so;
  }
}
```

Notion de verrou

- Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé

  => mettre en place la notion de section critique.
- Pour verrouiller un objet par un thread, il faut utiliser le mot clé synchronized.
- Il existe deux façons de définir une section critique.
  - soit on synchronise un ensemble d'instructions sur un objet :

```
synchronized ( object ) {
    // Instructions de manipulation d'une ressource partagée.
}
```

  - soit on synchronise directement l'exécution d'une méthode pour une classe donnée.

```
public synchronized void meth (int param ) {
    // Le code de la méthode synchronisée.
}
```

L'exclusion mutuelle

- Chaque objet Java possède un verrou dont la clé est gérée par la JVM.
- Lorsqu'un thread souhaite accéder à une méthode synchronisée d'un objet, il demande la clé de cet objet à la JVM, entre dans la méthode, puis ferme le verrou à clé.
- De cette façon, aucun autre thread ne peut accéder aux méthodes synchronisées de cet objet.
- Lorsque le thread sort de la méthode synchronisée, il ouvre de nouveau le verrou et rend la clé à la JVM.
- Un autre thread peut alors accéder aux méthodes synchronisées de l'objet.

Synchronisation

Lorsqu'il démarre, un thread travaille sur les données du thread dans lequel il a été crée.

Par exemple dans le programme `CadreAvecHeure`, le thread fils qui exécute la méthode `gererAffichageHeure` accède aux données `this` (atteinte lors de l'appel de `setTitle`) et `go` qui appartiennent au thread père qui l'a crée.

Cela pose le problème de l'accès concurrent aux données, source potentielle de blocage ou d'intégrité des données.

Pour résoudre ce type de problème d'accès concurrent nous utilisons des verrous qui protègent les ***sections critiques***.

Méthodes synchronisées (1)

Pour une illustration d'utilisation des threads, nous allons présenter un problème classique de la programmation parallèle:

Une implémentation du modèle des producteurs - consommateurs, considérons le cas suivant:

- Un certain nombre de fois (*par exemple 10*) chaque producteur "fabrique" un produit (numéroté), le dépose dans un entrepôt qui ne peut en contenir qu'un, puis dort un temps variable.
- Un certain nombre de fois (*par exemple 10*) chaque consommateur prend le produit qui se trouve dans l'entrepôt et le "consomme".

- A noter que les producteurs et les consommateurs ne se connaissent pas et ne prennent aucune mesure pour prévenir les conflits.
  - si l'entrepôt est vide, les consommateurs ne peuvent pas consommer,
  - s'il est plein les producteurs ne peuvent pas produire,
- C'est l'entrepôt qui gère ces conflits, il est indépendant du nombre de producteurs et de consommateurs qui traitent avec lui.

Méthodes synchronisées (2)

Le code du producteur:

```
public class Producteur extends Thread {
   private Entrepot entrepot;
   private String nom;
   public Producteur(Entrepot entrepot, String nom) {
        this.entrepot = entrepot;
        this.nom = nom;
   }

   public void run() {
        for(int i=0; i<10; i++) {
            entrepot.deposer(i);
            System.out.println("Le producteur " + this.nom + " produit " + i);
            try {
                sleep((int) (Math.random() * 100));
            } catch(InterruptedException e) {
            }
        }
   }
}
```

Méthodes synchronisées (3)

Le code du consommateur:

```
public class Consommateur extends Thread {
   private Entrepot entrepot;
   private String nom;
   public Consommateur(Entrepot entrepot, String nom) {
        this.entrepot = entrepot;
        this.nom = nom;
   }

   public void run() {
       int valeur = 0;
        for(int i=0; i<10; i++) {
            valeur = entrepot.prendre();
            System.out.println("Le consommateur "+this.nom+" consomme "+valeur);
        }
   }
}
```

Méthodes synchronisées (4)

Le code de l'entrepôt:

```
public class Entrepot {
   private int contenu;
   private boolean disponible = false;

   public synchronized int prendre() {
        while (disponible == false) {
           try {
                wait();
           } catch(InterruptedException e) {
           }
        }
        disponible = false;
        notifyAll();
        return contenu;
   }
```

Méthodes synchronisées (5)

Le code de l'entrepôt (suite):

```
   public synchronized void deposer(int valeur) {
        while (disponible == true) {
           try {
                 wait();
           } catch(InterruptedException e) {
           }
        }
        contenu = valeur;
        disponible = true;
        notifyAll();
   }
}
```

Remarque:

- Le fonctionnement de l'entrepôt: *les méthodes* ***prendre*** *et* ***deposer*** *étant s****ynchronized***, *un seul thread peut s'y trouver à un instant donné*.

Méthodes synchronisées (6)

Le code du programme principal:

```
public class EssaiPC {
   public static void main(String[] args) {
        Entrepot entrepot = new Entrepot();
        Producteur producteur = new Producteur(entrepot, "A");
        Consommateur consommateur = new Consommateur(entrepot, "B");
        producteur.start();
        consommateur.start();
   }
}
```

Affichage:

- Le producteur A produit 0
- Le consommateur B consomme 0
- Le producteur A produit 1
- Le consommateur B consomme 1

Méthodes synchronisées (7)

Le fait que les méthodes `prendre` et `deposer` sont *synchronisées*, un seul thread peut se trouver à l'entrepôt à un instant donné.

Si un *consommateur* appelle la méthode `prendre`,

- si un produit est disponible (`disponible == true`) alors `disponible` devient `false` et une notification est envoyée à tous les *threads* qui sont en attente d'une notification sur cet entrepôt; en même temps, le *consommateur* obtient le produit,

- s'il n'y a pas de produit disponible (`disponible==false`) alors le *thread* appelle **`wait`** et se met donc en attente d'une notification sur cet entrepôt.
- l'appel de **`notifyAll`** débloque tous les *threads* en attente, si parmi eux il y'a des *producteurs*, au plus un d'entre eux trouve `disponible== false` et peut donc produire et déposer, tous les autres (*producteurs et consommateurs*) se *rebloquent* (à cause du **`while`**) et attendent une nouvelle notification.
- Le fonctionnement de l'*entrepôt* vu du côté de la méthode `deposer` est symétrique du précédent.

Les Timers (1)

Les threads peuvent gérér le temps:

Exemples:

Un Thread peut repousser une action dans le temps:

- Dans 3 secondes, si l'utilisateur n'a pas bougé sa souris, afficher un popup disant: "*Ce bouton sert à quitter le document*".

Un Thread peut répéter une action régulièrement:

- Tous les 5 secondes, redessine la barre de progression.

Pour ces cas simples, pas besoin de faire des Threads compliqués: on utilise mieux un `Timer`.

- Déclencher une action tout en continuant,
- Déclencher une action après n millisecondes,
- Déclencher des actions toutes les p millisecondes.

Les Timers (2)

Considérons le cas d'une répétition d'une certaine tâche à intervalles réguliers.

Une première technique pour ce faire est d'utiliser la méthode `sleep` pour endormir le thread durant un certain temps. On pourrait par exemple réaliser un `timer` ainsi :

```
import java.util.Date;
public class SimpleTimer {
    public static void main (String[] args) {
        try {
                boolean finished = false;
                while (! finished) {
                // Exécution de la tâche
                  System.out.printf ("%tR\n", new Date()); // format 16:43
                Thread.sleep (2000); // En pause pour deux secondes
                }
            }
        catch (InterruptedException exception){}
    }
}
```

Les classes `Timer` et `TimerTask (1)`

Il y a une classe prévue pour représenter des *timers*, dans la librairie standard Java.

La classe `java.util.Timer` permet de commander l'exécution de tâches qui sont représentées par la classe `java.util.TimerTask`.

Programmer une tâche:

1- Définir la tâche à exécuter. Pour cela, il suffit de définir une nouvelle classe qui étend la classe `TimerTask` et qui redéfinit la méthode public `void run()` dans laquelle on va définir le comportement de la tâche.

2- Créer une instance de la classe `Timer` et d'y attacher la tâche à exécuter en utilisant la méthode `schedule` (*planing*) qui existe sous plusieurs formes.

Les classes `Timer` et `TimerTask` (2)

Exemple: `TimerExemple.java`

```
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class TimerExemple {
    public static void main (String[] args) {
        Timer timer = new Timer();
        timer.schedule (new TimerTask() {
               public void run() {
                                System.out.printf ("%tR\n", new Date());
                               }
        }, 0, 1000);
    }
}
```

La méthode `schedule` (*calendrier*) utilisée prend trois paramètres.

1- Le premier est la tâche à exécuter (instance de la classe `TimerTask`),

2- le second est le temps qu'il faut attendre avant de démarrer le `timer`,

3- Enfin le dernier indique l'intervalle de temps de répétition de la tâche.

Les classes `Timer` et `TimerTask (3)`

La méthode `schedule` est surchargée et existe sous 4 formes dont voici les signatures :

```
public void schedule (TimerTask task, Date time);
public void schedule (TimerTask task, Date firstTime, long period);
public void schedule (TimerTask task, long delay);
public void schedule (TimerTask task, long delay, long period);
```

- Le paramètre `delay` représente un nombre de millisecondes qui sera attendu avant que le timer ne se mette en route.
- Le paramètre `period` indique l'intervalle de temps, en millisecondes, entre deux exécutions de la tâche.
- Enfin, le paramètres `time` (et `firstTime`) indique l'instant auquel la première exécution aura lieu.

Avec la méthode `schedule`, on pourra donc soit programmer l'exécution d'une tâche une seule fois, ou des exécutions répétées à intervalle régulier.