JAVA
PROGRAMMATION OBJET

Centre BTS Al KENDI

2DSI

8. Les Threads

DÉFINITIONS

❑ Un programme est multitâche quand il lance (ou peut lancer) l'exécution de plusieurs parties de son code en même temps

- A un moment donné, il comporte plusieurs points d'exécution liés aux différentes parties qui s'exécutent en parallèle
- Tous les systèmes d'exploitation modernes sont multitâches et ils permettent l'exécution de programmes multitâches
- Si le système est préemptif, il peut à tout moment prendre la main à un programme pour la donner à un autre
- Sinon, un programme garde la main jusqu'à ce qu'il la cède à un autre

UTILITÉ DU MULTITÂCHE

Sur une machine multiprocesseurs

- il permet d'améliorer les performances en répartissant les différentes tâches sur différents processeurs

  - ✓ Par exemple, le calcul du produit de 2 matrices peut être réparti en n tâches parallèles (ou k tâches si le nombre k de processeurs est inférieur à n)

  - ✓ La répartition des tâches sur les processeurs est le plus souvent faite automatiquement par le système qui offre le multitâche

UTILITÉ DU MULTITÂCHE

Sur une machine monoprocesseur

➢ L'exécution en parallèle est simulée, il est intéressant d'utiliser le multitâche pour

- ✓ modéliser plus simplement (simulation par exemple)
- ✓ profiter des temps de pose d'une tâche (attente d'entrées-sorties ou d'une action de l'utilisateur) pour exécuter d'autres tâches
- ✓ réagir plus vite aux actions de l'utilisateur en rejetant une tâche longue et non-interactive dans un autre *thread* (par exemple, chargement d'une image ou lecture de données qui proviennent d'un réseau)

PROCESSUS, THREAD ET MULTITHREADING

Processus

- est un programme qui s'exécute et qui possède son propre espace mémoire : ses registres, ses piles, ses variables et son propre processeur virtuel

Thread

- Un processus peut lui-même fonctionner comme le S.E. en lançant des sous-tâches internes au processus (multi-programmation). Ces sous-tâches sont nommées "flux d'exécution" ou Threads

Multithreading

- Les S.E. (Windows, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, on désigne cette fonctionnalité sous le nom de Multithreading.

DIFFERENCE ENTRE THREAD ET PROCESSUS

- Communication entre threads est plus rapide que la communication entre processus
- Les threads partagent un même espace de mémoire (de travail) entre eux
- Les processus ont chacun un espace mémoire personnel
- Un processus peut lancer plusieurs *threads* qui se partagent le même espace mémoire et peuvent donc se partager des variables
- Un thread prend moins de ressources système qu'un processus

JAVA ET LES THREADS

Java supporte l'utilisation des *threads* :

- A l'inverse de la plupart des autres langages, le programmeur n'a pas à utiliser des librairies natives du système pour écrire des programmes multitâches
- Une thread est une unité d'exécution au sein d'un même processus (ce n'est pas un autre processus)
- Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée"
- La gestion des threads est dépendante de la JVM (problème pour la portabilité)
- Les threads peuvent être préemptifs ou coopératifs

EXEMPLE DES THREADS

- L'interface graphique avec l'utilisateur lance un *thread* pour charger une image pour continuer à traiter les événements générés par les actions de l'utilisateur

- Le serveur réseau qui attend les demandes de connexions venant des autres machines lance un thread pour traiter chacune des demandes

- La multiplication de 2 matrices (*m*, *p*) et (*p*, *n*) peut être effectuée en parallèle par *m*×*n* threads

CRÉATION D'UNE THREAD

- Une classe est une thread si elle remplit une des deux conditions :
  - Elle étend la classe *java.lang.Thread* :
  - Elle implémente l'*interface java.lang.Runnable*

- Le corps du thread est contenu dans une méthode :

  public void run ()

- Un thread est lancé par appel d'une méthode

  start ()

PREMIER EXEMPLE

```
class DeuxThreads {
     public static void main(String args[]) {
          new UneThread("la thread 1").start();
          new UneThread("la seconde thread").start();
    }
}

class UneThread extends Thread {
    public UneThread(String str) {
    super(str);
    }

   public void run() {
       for (int i=0; i<10; i++) {
              System.out.println(i+" "+getName());
                 try {sleep((int)(Math.random()*10));}
                      catch (InterruptedException e){}
        }
     System.out.println(getName()+" est finie");
   }
}
```

PREMIER EXEMPLE

- une exécution

```
0 la thread 1
0 la seconde thread
1 la thread 1
2 la thread 1
1 la seconde thread
3 la thread 1
4 la thread 1
5 la thread 1
6 la thread 1
7 la thread 1
2 la seconde thread
3 la seconde thread
4 la seconde thread
8 la thread 1
5 la seconde thread
9 la thread 1
6 la seconde thread
la thread 1 est finie
7 la seconde thread
8 la seconde thread
9 la seconde thread
la seconde thread est finie
```

UNE CLASSE "THREADÉE"

 C'est une classe qui implémente une thread.

Syntaxiquement on la construit :

- 1ère façon comme classe dérivée de la classe thread :
  - ✓ créer une instance d'une classe fille de la classe thread;

    la classe fille doit redéfinir la méthode run()

- 2ème façon comme implémentation de l'interface Runnable :
  - ✓ utiliser le constructeur Thread(Runnable) de la classe Thread :
    1. créer un Runnable
    2. le passer au constructeur de thread

UNE CLASSE "THREADÉE"

- 1ère façon comme classe dérivée de la classe Thread

```
class MaClasseThread extends Thread {
        . . .
        public void run() {
        // Code qui sera exécuté par le thread
        . . .
        }
}
```

- Quand on écrit :

```
MaClasseThread t1 = new MaClasseThread();
```

- Le code lancé par t1.start(); est le code run() de la classe threadée MaClasseThread

UNE CLASSE "THREADÉE"

Ecrire un programme qui va lancer trois threads simples, chacun d'entre eux se contentant d'afficher un certain nombre de fois un texte donné, à savoir :

- 10 fois "bonjour" pour le premier thread,
- 12 fois "bonsoir" pour le deuxième thread,
- 5 fois un changement de ligne pour le troisième thread.

```
public class TstThr1 {
public static void main (String args[]) {
   Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
   Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
   Ecrit e3 = new Ecrit ("\n", 5, 15) ;
   e1.start() ;
   e2.start() ;
   e3.start() ; }
}
```

UNE CLASSE "THREADÉE"

```
class Ecrit extends Thread {
    private String texte ;
    private int nb ;
    private long attente ;
    public Ecrit (String texte, int nb, long attente) {
    this.texte = texte ; this.nb = nb ; this.attente = attente ; }
    public void run () {
     try  { for (int i=0 ; i<nb ; i++) { System.out.print (texte) ;
                                         sleep (attente) ; }
          }
     catch (InterruptedException e) {} // imposé par sleep
    }
}
```

```
bonjour bonsoir bonjour
bonsoir bonjour bonjour bonsoir bonjour
bonjour bonsoir bonjour
bonjour bonsoir bonjour
bonjour bonsoir bonsoir
bonsoir bonsoir bonsoir bonsoir bonsoir
```

UNE CLASSE "THREADÉE"

 2ème façon comme implémentation de l'interface Runnable :

```
class MonRunnable implements Runnable {
        . . .
        public void run() {
        // Code qui sera exécuté par le thread
        . . .
        }
}
```

- Quand on écrit :

```
MonRunnable monRunnable = new MonRunnable(…);
Thread t1 = new Thread(monRunnable) ;
```

- Le code lancé par `t1.start();` est le code `run()` de l'objet référencé par `monRunnable`

UNE CLASSE "THREADÉE"

- Si on veut hériter d'une autre classe pour la classe qui contient la méthode run(), on est obligé de choisir la 2ème façon (Thread(Runnable))

- Il est aussi plus simple d'utiliser la 2ème façon pour partager des données entre plusieurs threads

- Sinon, l'écriture du code est (légèrement) plus simple en utilisant la 1ère façon

UNE CLASSE "THREADÉE"

```
public class TstThr3 {
public static void main (String args[]) {
   Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
   Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
   Ecrit e3 = new Ecrit ("\n", 5, 15) ;
   Thread t1 = new Thread (e1) ; t1.start() ;
   Thread t2 = new Thread (e2) ; t2.start() ;
   Thread t3 = new Thread (e3) ; t3.start() ; }
}
```

```
bonjour bonsoir
bonjour bonsoir
bonjour bonjour bonsoir bonjour
bonjour bonsoir bonjour bonjour bonsoir
bonjour bonjour bonsoir
bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir
```

```
class Ecrit implements Runnable {
  private String texte ;
  private int nb ;
  private long attente ;
 public Ecrit (String texte, int nb, long attente) {
       this.texte = texte ; this.nb = nb ; this.attente = attente ; }
public void run () {
  try { for (int i=0 ; i<nb ; i++) { System.out.print (texte) ;
                                     Thread.sleep (attente) ; } }  // attention Thread.sleep
 catch (InterruptedException e) {}   }   // impose par sleep
}
```

UNE CLASSE "THREADÉE"

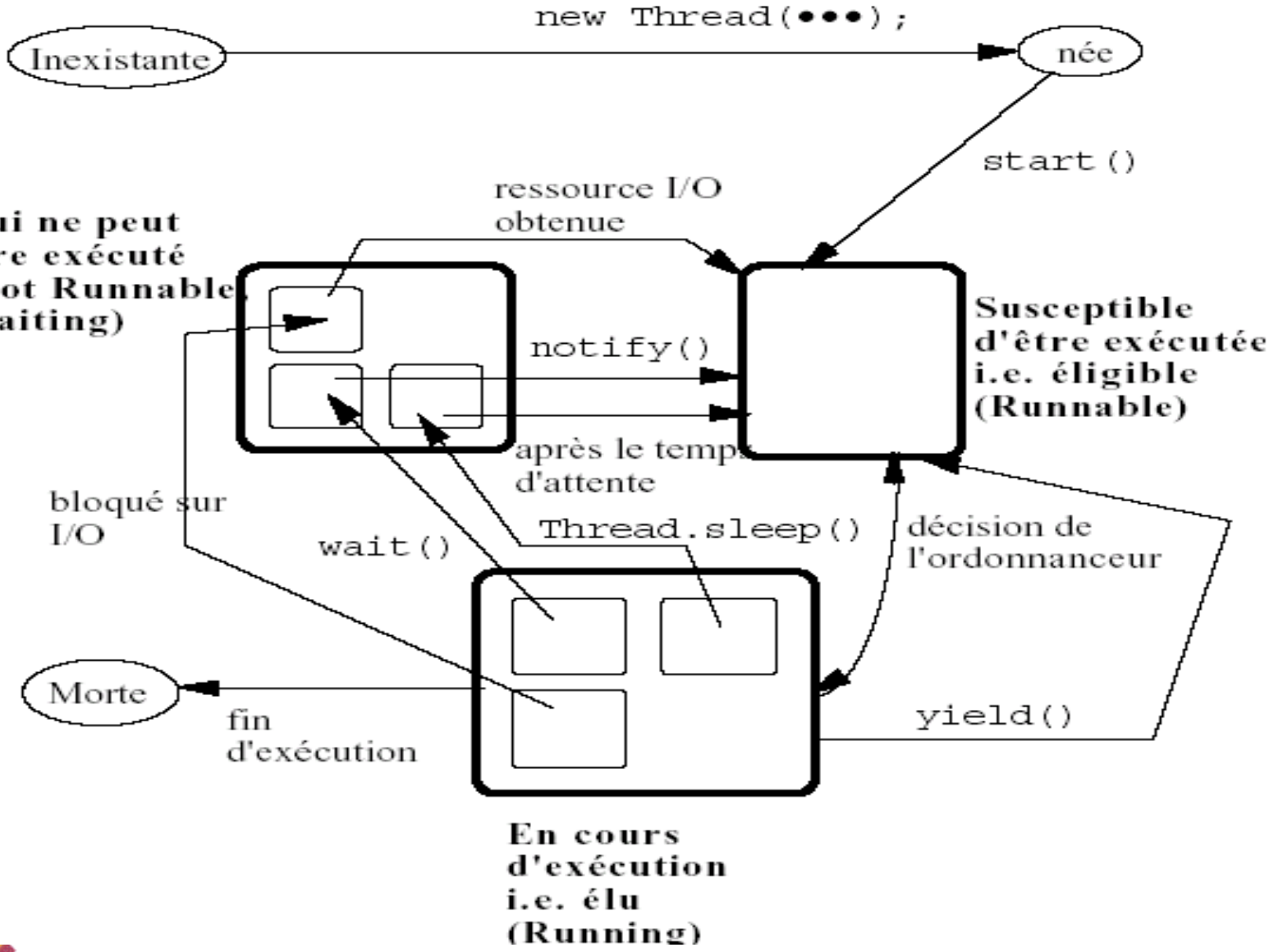
On peut doter la classe *Ecrit* d'une méthode nommée *start* jouant un double rôle :

- création de l'objet de type *Thread*
- et lancement du thread.

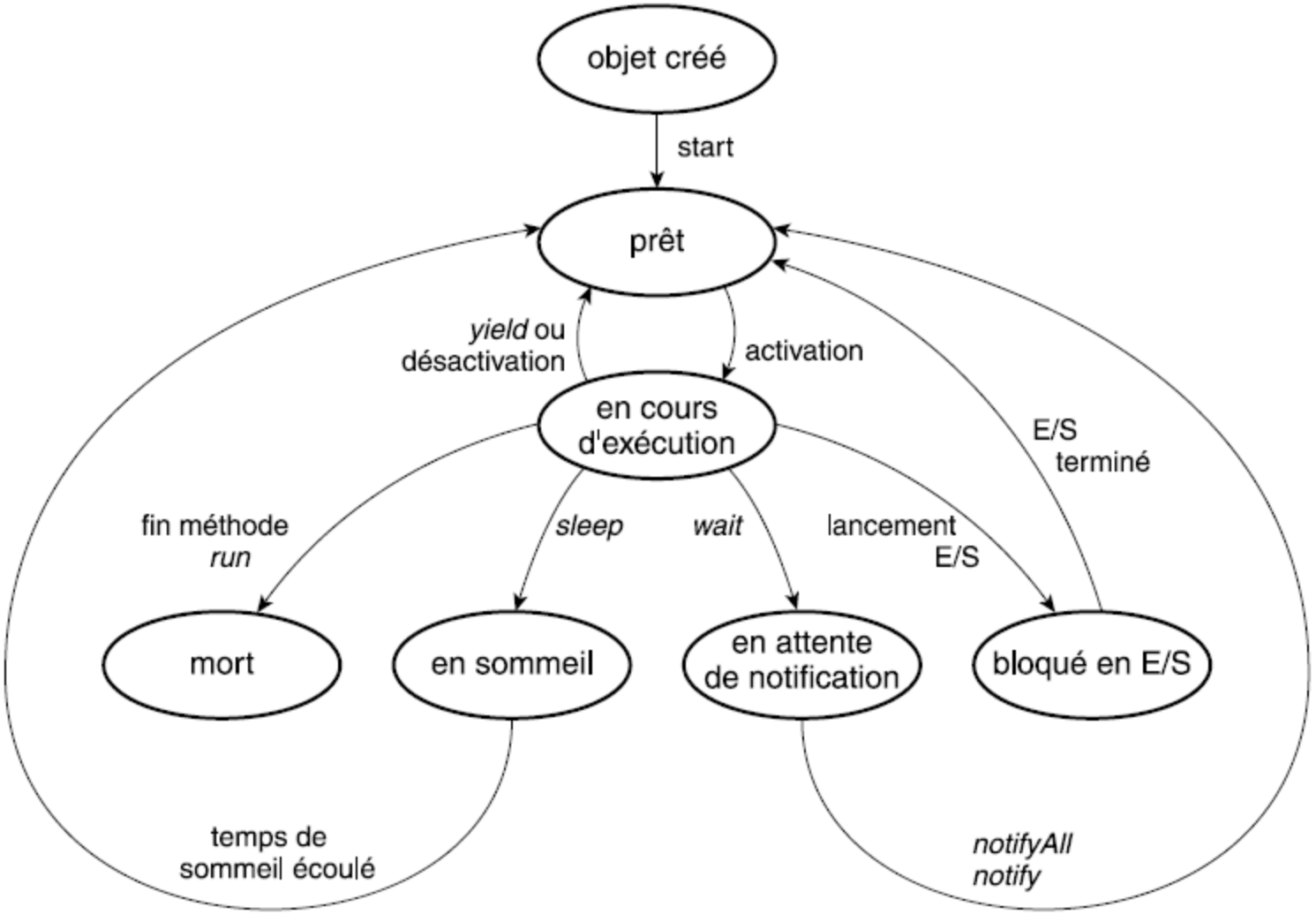
```
public class TstThr3 {
public static void main (String args[]) {
   Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
   Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
   Ecrit e3 = new Ecrit ("\n", 5, 15) ;
   e1.start() ;
   e2.start() ;
   e3.start() ; }
}
```

```
class Ecrit implements Runnable {
   .....
  public Ecrit (String texte, int nb, long attente) {
   // constructeur inchangé }
  public void start () {
 Thread t = new Thread (this) ;
  t.start() ; }
  public void run () {
  // méhode run inchangée }
}
```

LES ÉTATS D'UN THREAD



LES ÉTATS D'UN THREAD



Les différents états d'un thread

LES ÉTATS D'UN THREAD

- À l'état "née", un thread n'est qu'un objet vide et aucune ressource système ne lui a été alloué

- On passe de l'état "née" à l'état "Runnable" en lançant la méthode start()
  - Le système lui alloue des ressources, l'indique à l'ordonnanceur qui lancera l'exécution de la méthode run()

- A tout instant c'est un thread éligible de plus haute priorité qui est en cours d'exécution

LES ÉTATS D'UN THREAD

- On entre dans l'état "Not Runnable" suivant 3 cas :
  - quelqu'un a lancé la méthode sleep()
  - le thread a lancé la méthode wait() en attente qu'une condition se réalise
  - le thread est en attente d'entrées/sorties

- Pour chacune de ces conditions on revient dans l'état "Runnable" par une action spécifique :
  - on revient de sleep() lorsque le temps d'attente est écoulé
  - on revient de wait() par notify() ou notifyAll().
  - bloqué sur une E/S on revient a "Runnable" lorsque l'opération d'E/S est réalisé

- On arrive dans l'état "Morte" lorsque l'exécution de run() est terminée

ORDONNANCEMENT DES THREADS

- A tout thread est associée une priorité
  - Priorité de 1 (Thread.MIN_PRIORITY) à 10 (Thread.MAX_PRIORITY), par défaut 5
  - Un thread adopte la priorité de son thread créateur
  - setPriority(int p) permet de changer le niveau de priorité

- L'ordonnancement est dépendant des plateformes :
  - Sous Windows, et MacOS, un tourniquet pour les threads de même priorité est installé i.e. un thread peut être préempté par l'ordonnanceur par un autre thread de même priorité.
  - Sous Solaris, si on utilise l'ordonnanceur de threads de l'interpréteur ("green threads"), il n'y a pas de tourniquet (sur les threads de même priorité) : thread "égoïste".

ORDONNANCEMENT DES THREADS

```
class ThreadExtends extends Thread {
        public void run() {
                while(true) {
                System.out.println("dans ThreadExtends.run");
                }
        }
}

public class Thread1 {
        public static void main(String args[]) {
                ThreadExtends t0 = new ThreadExtends();
                t0.start();
                while(true){
                System.out.println("dans Thread1.main");
                }
        }
}
```

ORDONNANCEMENT DES THREADS

Sous Windows : ordonnanceur de type Tourniquet

```
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
dans ThreadExtends.run
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
dans ThreadExtends.run
dans ThreadExtends.run
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
```

Sous Solaris : monopolisation du processeur thread égoïste

```
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
...
```

SYNCHRONISATION ENTRE THREADS

- L'utilisation des *threads* peut entraîner des problèmes liés aux accès simultanés aux variables

- On appelle section critique une partie du code qui ne peut être exécutée en même temps par plusieurs threads sans risquer de provoquer des anomalies de fonctionnement

- Exemple de problème : Si x=2, le code x=x+1; exécuté par 2 threads, peut donner en fin d'exécution 3 ou 4

  suivant l'ordre d'exécution :

  1. T1 : lit la valeur de x (2)
  2. T2 : lit la valeur de x (2)
  3. T1 : calcul de x+1 (3)
  4. T2 : calcul de x+1 (3)
  5. T1 : range la valeur calculée dans x (3)
  6. T2 : range la valeur calculée dans x (3)

  x contient 3 au lieu de 4!

SYNCHRONISATION

- Il faut donc éviter l'exécution simultanée de sections de code critiques par plusieurs threads

- Si plusieurs threads veulent exécuter en même temps une section critique, on devra les synchroniser pour qu'ils exécutent cette section les uns après les autres

- En Java les sections critiques sont représentées par du code synchronisé sur un objet

Synchronized POUR UN BLOC

- Dans synchronized (expression) { ··· }

  - où expression repère un objet O, le système Java pose un verrou sur cet objet O pendant l'exécution du bloc

  - Aucun autre bloc synchronized sur cet objet APPARTENANT A UN AUTRE THREAD ne peut être exécuté

  - Un tel autre bloc ne peut être lancé qu'après avoir obtenu ce verrou

  - Exemple :

```
public void trie(int[] tableau) {
        synchronized (tableau) {
                // votre algorithme de tri préféré
        }
}
```

Synchronized : MODIFIEUR DE MÉTHODE D'INSTANCE

 synchronized est souvent utilisé comme modifieur de méthode

```
void synchronized meth() {…}
        signifie
void meth() { synchronized (this) {…}
            }
```

- Lorsqu'un thread veut lancer une méthode d'instance synchronized, le système Java pose un verrou sur l'instance
- un autre thread invoquant une méthode synchronized sur cet objet sera bloquée jusqu'à ce que le verrou soit levé

Synchronized : EXEMPLE 1

- On lance 3 threads du type suivant :

```
public class synchro {
    public static void main(String[] args) {
        Thread t1 = new ThreadExtends("t1"); t1.start();
        Thread t2 = new ThreadExtends("t2"); t2.start();
        Thread t3 = new ThreadExtends("t3"); t3.start();
    }
}



class Compte {
    private static double solde ;
    static void deposer(double somme) { solde = solde + somme;}
    public static double getSolde() {return solde;}
}
```

Synchronized : EXEMPLE 1

```
class ThreadExtends extends Thread {
        public ThreadExtends(String str) {
        super(str);
        }
        public void run() {
              for (int i=0; i<10; i++) {
                    System.out.println(i + "  " + getName());
                    Compte.deposer(1000);
                    try {sleep((int)(Math.random()*10));}
                    catch (InterruptedException e){}
               }
              System.out.println("solde    =    " + Compte.getSolde());
      }
}
```

Synchronized : EXEMPLE1

```
0  t1
0  t3
0  t2
1  t3
1  t1
2  t3
1  t2
3  t3
……..
8  t2
9  t1
solde   =    28000.0
solde   =    28000.0
9  t2
solde   =    29000.0
```

- A la fin de l'exécution, on n'obtient pas nécessairement 30.000

- Il faut rendre deposer (et getSolde) synchronisée :

```
public synchronized void deposer(double somme)
```

Synchronized : EXEMPLE 2

Voici un programme qui gère une "réserve". Il comporte :

- deux threads qui ajoutent chacun une quantité donnée,
- un thread qui puise une quantité donnée.

Un thread ne peut puiser dans la réserve que si elle contient une quantité suffisante.

La réserve est représentée par un objet *r*, de type *Reserve*

La classe *Reserve* dispose de deux méthodes synchronisées *puise* et *ajoute*.

Lorsque la méthode *puise* s'aperçoit que la réserve est insuffisante, il appelle *wait* pour mettre le thread correspondant en attente. Parallèlement, la méthode *ajoute* appelle *notifyAll* après chaque ajout.

Les trois threads sont lancés par *main* et interrompus lorsque l'utilisateur le souhaite (en frappant un texte quelconque).

Synchronized : EXEMPLE 2

```
public class Synchro {
public static void main (String args[]) {
Reserve r = new Reserve () ;
ThrAjout ta1 = new ThrAjout (r, 100, 15) ;
ThrAjout ta2 = new ThrAjout (r, 50, 20) ;
ThrPuise tp = new ThrPuise (r, 300, 10) ;
System.out.println ("Suivi de stock --- faire entrée pour arrêter ") ;
ta1.start () ; ta2.start () ; tp.start () ;
Clavier.saisie_string() ;
ta1.interrupt () ; ta2.interrupt () ; tp.interrupt () ;
}
}
```

```
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
++ on ajoute 100 et il y a maintenant 150
** stock de 150 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 200
** stock de 200 insuffisant pour puiser 300
++ on ajoute 100 et il y a maintenant 300
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
```

Synchronized : EXEMPLE 2

```
class Reserve extends Thread {

private int stock = 500 ; // stock initial = 500

public synchronized void puise (int v) throws InterruptedException {
if (v <= stock) { System.out.print ("-- on puise " + v) ;
                  stock -= v ;
                  System.out.println (" et il reste " + stock ) ; }
else { System.out.println ("** stock de " + stock + " insuffisant pour puiser " + v ) ;
       wait() ;}
 }

public synchronized void ajoute (int v) {
stock += v ;
System.out.println ("++ on ajoute " + v + " et il y a maintenant " + stock) ;
notifyAll() ; }
}
```

Synchronized : EXEMPLE 2

```
class ThrAjout extends Thread {
private int vol ;

private Reserve r ;

private int delai ;

public ThrAjout (Reserve r, int vol, int delai) {
    this.vol = vol ;
    this.r = r ;
    this.delai = delai ;
}

public void run () {
   try { while (!interrupted()) {
                                        r.ajoute (vol) ;
                                        sleep (delai) ;
                                       }
       }
   catch (InterruptedException e) {}
}
}
```

Synchronized : EXEMPLE 2

```
class ThrPuise extends Thread {
private int vol ;

private Reserve r ;

private int delai ;

public ThrPuise (Reserve r, int vol, int delai) {
        this.vol = vol ;
        this.r = r ;
        this.delai = delai ;
}

public void run () {
    try { while (!interrupted()) {
                                        r.puise (vol) ;
                                        sleep (delai) ;
                                        }
          }
   catch (InterruptedException e) {}
 }
}
```

EXERCICE 1

Pour remplir deux barils de 200 Litres avec l'eau à partir d'un puit, on fait appel à deux joueurs A et B disposant chacun un seau vide de 10 Litres. La quantité d'eau portée dans le seau par un joueur au cours d'une opération de remplissage est donc variable entre 1 à 10 Litres. Le jeu de remplissage s'arrête dès qu'un joueur arrive à remplir son baril.

Ecrire une application JAVA qui permet de simuler le jeu de remplissage des barils. L'application doit afficher :

- les différentes étapes de remplissage de chaque joueur ;
- le nombre d'opérations de remplissage et la quantité d'eau mise dans le baril du deux joueurs ;
- le nom du gagnant.

```
import java.io.* ;
public class Eau {
   int volume;
   /* Création d'une instance Eau */
   public Eau(int vol) { volume=vol ; }
}
```

EXERCICE 1

```
public class Baril {
public static final int maximum = 200;
public Eau contenu = new Eau(0);
public int getcontenu() {return contenu.volume; }
public void ajouter (Eau quantite) {
   contenu.volume +=quantite.volume;
   if (contenu.volume>=maximum) contenu.volume=maximum; }
public boolean estplein() {
if (contenu.volume >= maximum) return true ; else return false; }
}
```

```
public class remplissage {
      public static void main(String[] args) {
      Joueur Joueur1= new Joueur("A");
      Joueur Joueur2= new Joueur("B");
      Joueur1.start();
      Joueur2.start();
      }
}
```

EXERCICE 1

```
class Joueur extends Thread {
public String nom ;
public Baril Baril1;
public int ope=0;
public static boolean plein=false;
public Joueur(String A) {nom=A; Baril1=new Baril();}
public void run() {
   int q=0;
   while (plein==false) {
   try {  q=(int)(Math.random()*10);
          Eau e= new Eau(q);
          Baril1.ajouter(e);
          System.out.println(" Joueur : " + nom + " q : " + e.volume + " contenu Baril : "
                                                          +Baril1.contenu.volume);
          ope+=1;
         if (Baril1.estplein()) {
         plein=true;
         System.out.println(" Le Baril du Joueur " + nom + " est plein ");
         System.out.println("Le nombre d'opérs effectuées par : " + nom + " est " + ope);
         System.out.println(" je suis le Joueur : " + nom + " gagnant  ");
         }
        sleep(q);
      }   catch (InterruptedException e) { }
} } }
```

EXERCICE 2

Un producteur est un thread qui dépose des jetons numérotés dans un chapeau qui ne peut contenir qu'un seul jeton. Un consommateur prend ce jeton qui doit être présent dans le chapeau. Donc :

- le producteur doit s'arrêter de déposer des jetons lorsqu'il y en a déjà un et doit être informer qu'un jeton a été retiré.
- le consommateur ne peut pas prendre de jeton s'il n'y en a pas (arrêt du consommateur) et doit être informé lorqu'un jeton a été déposé.

```
public class ProducteurConsommateurTest {
        public static void main(String[] args) {
        Chapeau chap = new Chapeau();
        Producteur p = new Producteur(chap);
        Consommateur c = new Consommateur (chap);
        p.start();
        c.start();
        }
}
```

EXERCICE 2

```
public class Producteur extends Thread {
        private Chapeau chapeau;

        public Producteur(Chapeau c) { chapeau= c;}

        publi c void run() {
                for (int i = 0; i < 10; i++) chapeau.put(i);

  }
```

```
public class Consommateur extends Thread {
        private Chapeau chapeau ;

        public Consommateur(Chapeau c) { chapeau = c;  }

        public void run() {
                for (int i = 0; i < 10; i++)  chapeau.get();
        }
    }
```

EXERCICE 2

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

        public synchronized void get() {
            if (disponible== false) {
                try { wait(); } catch (InterruptedException e) { }
            }
            disponible = false;
            System.out.println("Consommateur #" + " got : " + contenu);
            notify();
        }

        public synchronized void put(int value) {
            if (disponible == true) {
                try {  wait(); }  catch (InterruptedException e) { }
            }
            contenu = value; disponible = true;
            System.out.println("Producteur #" +" put: " + contenu);
            notify();
        }
}
```