Applications Concurrentes :

Conception,

Outils de Validation

ACCOV_B

Chapitre 5

CONCURRENCE EN JAVA

CONCURRENCE EN JAVA

MÉTHODES PUBLIQUES DE LA CLASSE Thread

final void suspend(); --*
final void resume(); --*
static native void yield();
final native boolean isAlive();
final void setName(String Nom);
final String getName();
public void run();
*** méthodes désapprouvées en Java2 car elles peuvent conduire à interblocage**

EXCLUSION MUTUELLE EN JAVA

On utilise le mot clé réservé synchronized
comme modifieur de méthode de classe ou de méthode d'instance ou de bloc;
Il garantit que cette méthode ou ce bloc est une section critique
(donc exécuté en exclusion mutuelle entre processus)
A chaque objet est associé un verrou. Un thread qui invoque une méthode synchronized doit obtenir le verrou avant d'exécuter la méthode

SYNCHRONISATION EN JAVA PAR MONITEUR

La synchronisation entre "thread" utilise des méthodes de la classe Object
A l'intérieur d'un objet, un thread utilise :

```
wait();          // met le thread en attente et relache l'accès à l'objet
notify();        // réveille un des processus qui attendent l'objet par wait()
                 // le choix est arbitraire et n'est pas obligatoirement FIFO
notifyAll();     // réveille tous les processus attendant l'objet par wait()
                 // ils seront exécuté dans un ordre quelconque
```

Ces méthodes doivent être lancées dans des méthodes synchronized
En java chaque moniteur n'a qu'une variable condition anonyme. Toutes les opérations wait, notify, notifyAll, concernent cette condition anonyme
Le moniteur en Java utilise la sémantique de réveil "signal and continue", le processus qui exécute notify ou notifyAll garde le moniteur et continue à s'exécuter dans l'objet.
Un processus qui invoque le moniteur peut y avoir accès avant un processus réveillé par une opération notify

IMPLANTATION DES SÉMAPHORES EN JAVA

```
public class semaphore {

   public Semaphore() {compteur = 0; }          // constructeur

   public Semaphore(int v) {compteur = v; }    // autre constructeur

   public synchronized void P() {
      while (compteur <= 0) {
         try { wait() ; }
         catch (InterruptedException e) {}
      }
      compteur--;          // compteur = compteur - 1
   }

   public synchronized void V() {
      ++ compteur;        // compteur = compteur + 1
      notify();           // ne réveille qu'un processus en attente
   }

   private int compteur;
}
```

EXCLUSION MUTUELLE MODULAIRE EN JAVA

```
public class Compte {      // on encapsule la donnée critique
        // et le contrôle d'exclusion mutuelle est implicite

    private int COMPTE_CLIENT = 0
        // variable commune pour les objets de la classe
    public synchronized void Section_critique(){

            int X = COMPTE_CLIENT;      // PQ1
            X := X + 1;                 // PQ2
            COMPTE_CLIENT := X;         // PQ3
    }   // fin de Section_critique ;

}      // fin de la classe Compte

public class PQ extends thread{

    private Compte compte;
    PQ(Compte compte){this.compte = compte;}

    public void run(){
        for (int I=1; I <16; I++){
            actions_hors _section_critique;
            Compte.Section_critique;
    }

}     // fin de PQ

public class Exclusion_Mutuelle_Modulaire_En_Java {
                // on y crée les processus
    static Compte c = new Compte();

        public static void main(String[] args) {    // programme principal
        PQ p = new PQ(c); PQ q = new PQ(c); PQ r = new PQ(c);
                    // on doit passer en paramètre les objets partagés
        p.start(); q.start(); r.start();           // lance les trois processus
    }                           // fin du programme principal main

}                       // fin de Exclusion_Mutuelle_Modulaire_En_Java
```

```
// GESTION D'HORLOGE EN EXCLUSION MUTUELLE

public class ClockReaderWriter{

   class Clock{
      private int minutes=0, seconds=0;          // accessible dans la classe seult

      synchronized void tick(){                  // en exclusion mutuelle
         minutes=minutes+((seconds+1)/60);
         seconds=(seconds+1)%60;
      }                                                          // fin de tick

      synchronized Clock read() {                // en exclusion mutuelle
         Clock result=new Clock();
         result.minutes=this.minutes;
         result.seconds=this.seconds;
         return result;
      }                                                          // fin de read

      // fonctions de selection pour acces a l'heure
      // car il n'y a pas de structure en Java
      int seconds() {return this.seconds;}
      int minutes() {return this.minutes;}

   }                                                          // fin de Clock
```

```
   class ClockReader extends Thread{                    // processus lecteur
      private Clock clock;                    // accessible dans la classe seulement
      ClockReader (Clock clock) {this.clock = clock;}
      public void run() {                  // accessible partout comme ClockReader
         while (true) {
            Clock now = clock.read;
            System.out.println("minutes:  " + now.minutes() );
            System.out.println("seconds:  " + now.seconds() );
            try {Thread.sleep(1000);}
            catch(Exception e){}
         }                                                          // fin de boucle
      }                                                               // fin de run
   }                                                            // fin de ClockReader

   class ClockWriter extends Thread{                    // processus redacteur
      private Clock clock;                    // accessible dans la classe seulement
      ClockWriter (Clock clock) {this.clock = clock;}
      public void run() {                  // accessible partout comme ClockWriter
         for (int i=1; i <= Integer.MAX_VALUE; i++){
            try {Thread.sleep(1000);}
            catch(Exception e){}
            clock.tick();
            System.out.println( "Tick  " + i);
         }                                                          // fin de boucle
      }                                                               // fin de run
   }                                                           // fin de ClockWriter
   public ClockReaderWriter(){
      Clock c = new Clock();      // on doit passer les objets partagés
      ClockWriter wr = new ClockWriter (c);
      ClockReader r = new ClockReader(c);
      wr.start() ; r.start(); // on lance les deux processus
   }                                              // fin du code de ClockReaderWriter
   public static void main (String[] args) {
      ClockReaderWriterRW = new ClockReaderWriter();
   }
}                                                 // fin de la classe ClockReaderWriter
// exercice : faire un essai en enlevant les synchronized dans Clock
```

```
        // PRODUCTEUR CONSOMMATEUR//  ProducerConsumer.java

class Buffer{
   protected final int max;
         //"protected" : accessible dans la classe et sous-classes
         // "final" : variable non mutable
   protected final Object[] data;                              // tableau d'Objet
   protected int nextIn=0, nextOut=0, count=0;

public Buffer(int max){
   this.max = max;
   this.data = new Object[max];
}
public synchronized void put(Object item)
        while (count == max)
           try {wait(); }
           catch (InterruptedException e){}
   data[nextIn] = item ;
   nextIn = (nextIn+1) % max;
   count++;
   notify();                // reveille un processus en attente s'il y en a
   }
}
public synchronized Object get()
        while (count == 0) {
           try {wait(); }
           catch (InterruptedException e){}

   Object result = data(nextOut))) ;
   nextOut = (nextOut+1) % max;
   count--;
   notify();                // reveille un processus en attente s'il y en a
   return result;
   }
}                                                              // fin de Buffer
// noter l'emploi de notify car il n'y a qu'un blocage à la fois
// valable même si on a plusieurs producteurs ou consommateurs
```

```
class Producer implements Runnable{                    // type processus producteur
   protected final Buffer buffer;
   Producer(Buffer buffer){ this.buffer=buffer; }
   public void run(){
      try{
         for (int j=1; j<=100; j++){
            Thread.sleep(500);
            buffer.put( new Integer(j) ); // crée un objet de classe Integer
         }                                          // fin du bloc où est levée l'exception
      } catch (InterruptedException e) {return}//en réponse à l'exception
   }
}                                                                  // fin de Producer

class Consumer extends Thread {                    // type processus consommateur
   protected final Buffer buffer;
   public Consumer(Buffer buffer){ this.buffer=buffer; }
   public void run(){
      try{
         for (int j=1; j<=100; j++){
            Integer p = (Integer) (buffer.get());
            int k = p.intValue();                          // conversion vers le type int
            Thread.sleep(1000);
         }                                          // fin du bloc où est levée l'exception
      } catch (InterruptedException e) {return}//en réponse à l'exception
   }
}                                                                  // fin de Consumer

public class ProducerConsumer{                     // objet programme principal
   static Buffer buffer = new Buffer(20);                        // tampon commun
   public static void main (String[] args) {
      Producer p = new Producer(buffer);
      Thread pt = new Thread(p);                  // car le producteur est Runnable
      Consumer c = new Consumer(buffer);          // on passe l'objet à partager
      pt.start() ;
      c.start();
   }                                                          // fin de la méthode main
}                                                 // fin de la classe ProducerConsumer
```

//LECTEURS ET RÉDACTEURS//

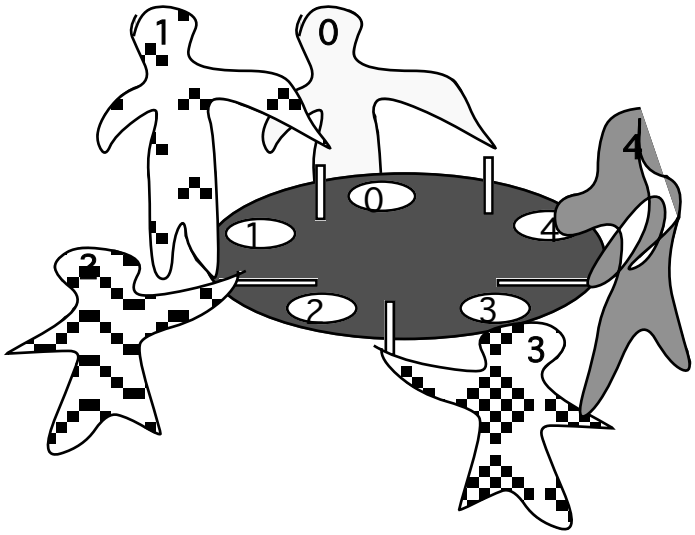
```
public class Database{
   public Database(){
      readerCount = 0 ; Reading = false; Writing = false;
   }

   public synchronized int startRead() {
      while (Writing == true) {
         try{ wait(); }
         catch(InterruptedException e) {}
      }
      ++readerCount;    // readerCount = readerCount + 1
      // le premier lecteur indique que la base est en accès en lecture
      if (readerCount == 1) Reading = true;
      return readerCount;
   }

   public synchronized int endRead() {
      --readerCount                                  // readerCount = readerCount -1
      // le dernier lecteur indique que la base n'est plus en lecture
      if (readerCount == 0) Reading = false;
      notifyAll();          //réveille tous les processus en attente
      return readerCount;
   }

   public synchronized int startWrite() {
      while (Reading == true || Writing == true) {
         try{ wait(); }
         catch(InterruptedException e) {}
      }
      // tout rédacteur indique que la base est en accès en écriture
      Writing = true;
   }

   public synchronized int endWrite() {
      Writing = false;
      notifyAll();          //réveille tous les processus en attente
   }
```

// suite des LECTEURS RÉDACTEURS//

```
   public static final int NAP_TIME = 5;

   private int ReaderCount ;
   private boolean Reading;                    // indique que la base est en lecture
   private boolean Writing ;                   // indique que la base est en écriture

   // les lecteurs et rédacteurs appellent cette méthode pour dormir
   public static void napping(){
      int sleeptime= (int) (NAP_TIME * Math.random());
      try{Thread.sleep(sleeptime * 1000);}
      catch(InterruptedException e) {}
   }
}

public class Reader extends Tread {
   public Reader(Database db) {server = db); }
   public void run() {
      int c ;
      while (true) {
         Database.napping();
         c = server startRead();
         Database.napping();
         c = server endRead();
      }
   }
   private Database server;
}
public class Writer extends Tread {
   public Writer (Database db) {server = db); }
   public void run() {
      while (true) {
         Database.napping();
         erver startWrite();
         Database.napping();
         server endWrite();
      }
   }
   private Database server;
}
```

```
// DINER DES PHILOSOPHES//        Philo.java
                    // Programme général //

public class Philo extends Thread{                    // type processus Philo

   private Server chops ;
   private int x ;

   Philo(int x ; Server : chops) {
      this.x = x ;
      this.chops = chops ;
      new Thread(this).start();
   }
   private void thinking(){
      System.out.println(x + " is thinking");
      yield();
   }
   private void eating(){
      System.out.println(x + " is eating");
      yield();
   }

   public void run() {   // code exécutable de Philo
      while (true) {
         thinking ;
         chops.request(x) ;
         eating ;
         chops.release(x) ;
      }
   }
}

public class Main {
   public static void main (String[] args) {
      final int N = 5;              //   5 philosophes 5 baguettes
      Server chops = new Server(N) ;                  // objet partagé de type Server
      for (int i = 0 ; i < N ; i++) {new Philo(i, chops) ; }
   }
}
```

DINER DES PHILOSOPHES

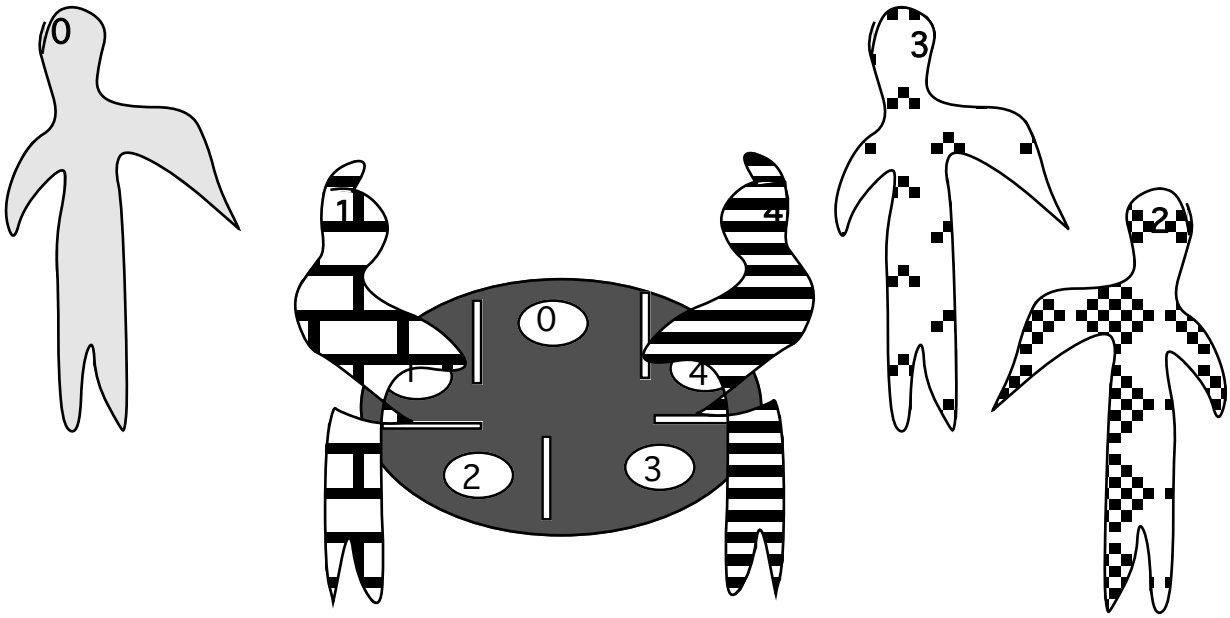


Première implantation de Server : allocation une baguette après l'autre

interblocage

```
// Server.java version 1
//allocation une baguette après l'autre avec risque d'interblocage

public final class Server {

   private int N ;
   private boolean available [ ] ;

   Server (int N) {
      this.N = N ;
      this.available = new boolean[N] ;
      for (int i =0 ; i < N ; i++) {
         available[i] = true ;
      }
   }

   public synchronized void request (int me) {
      while
      ( available [me] == false )
      {
         try { wait() ; }
         catch (InterruptedException e) {}
      }
      available [me] = false ;

      while
      ( available [(me + 1)% N] == false )
      {
         try { wait() ; }
         catch (InterruptedException e) {}
      }
      available [(me + 1)% N]  = false ;
   }

   public synchronized void release (int me) {
      available [me] = true ; available [(me + 1)% N]  = true ;
      notifyAll ;
   }
}
```

DINER DES PHILOSOPHES



Deuxième implantation de Server :
allocation globale et réveil de tous les demandeurs bloqués
(style Java)

available (x) and available(x + 1)
famine

mieux encore
available (x) and available(x + 1) and not requestor(x - 1)
ni famine ni interblocage
(Courtois, Georges, On starvation prevention, RAIRO vol 11, 1977)

(à vérifier si cela reste vrai avec la sémantique du moniteur Java)

```
// Server.java, version 2
// available (x) and available(x + 1) avec possibilité de famine

public final class Server {

   private int N ;
   private boolean available [];

   Server (int N) {
      this.N = N ;
      this.available = new boolean[N] ;
      for (int i =0 ; i < N ; i++) {
         available[i] = true ;
      }
   }

   public synchronized void request (int me) {
      while
      ( available [me] == false || available [(me + 1)% N] == false )
      {
         try { wait() ; }
         catch (InterruptedException e) {}
      }
      available [me] = false ; available [(me + 1)% N]  = false ;
   }

   public synchronized void release (int me) {
      available [me] = true ; available [(me + 1)% N] = true ;
      notifyAll ;            // on réveille tous le processus bloqués
   }                                 // car on n'a qu'une variable condition (implicite)
}
```

COMPORTEMENT DES PROCESSUS JAVA (OU "THREADS")

RÈGLES DE SYNCHRONISATION AVEC SYNCHRONIZED

1. Un processus qui possède le verrou d'un objet peut entrer dans une autre méthode (ou bloc) synchronized du même objet. L'exclusion mutuelle par verrou porte sur l'objet. Il n'y a qu'un verrou par objet.
2. Un processus peut imbriquer les invocations de méthode synchronized de différents objets. Ainsi un processus peut simultanément posséder les verrous de plusieurs objets différents. Cet emboitement de sections critiques peut conduire à interblocage.
3. Si une méthode n'est pas déclarée synchronized , elle peut alors être invoquée quelle que soit la possession du verrou, même si une autre méthode synchronized du même objet est en cours d'exécution.
4. Si la file des processus en attente d'un objet est vide, alors un appel à notify() ou notifyAll() n'a pas d'effet (et n'est pas mémorisé).

VERSION JAVA DES MONITEURS

Le verrou associé à un objet Java se comporte comme un moniteur. Un processus peut acquérir le moniteur d'un objet soit en entrant dans une méthode synchronized , soit en se bloquant.
Cependant Java ne fournit pas de support pour les variables condition nommées. Avec les moniteurs de Hoare, les opérations Wait et Signal peuvent être appliquées à une variable condition nommée, ce qui permet à un processus d'attendre une condition particulière ou d'être notifié lorsqu'une condition spécifique est satisfaite.
Chaque moniteur Java ne possède qu'une seule variable condition qui lui est associée et elle n'est pas nommée. Les opérations wait(), notify() et notifyAll() ne s'appliquent qu'à cette unique variable condition. Lorsqu'un processus Java est réveillé par notify() ou notifyAll(), il ne reçoit pas d'information sur l'origine de ce réveil. C'est au processus réactivé de vérifier lui-même si la condition qu'il attendait est satisfaite ou non. Les moniteurs Java utilisent l'approche "signal and continue" (voir au chapitre 3 du cours ACCOV_B) : lorsqu'un processus reçoit un signal par la méthode notify(), il peut prendre le verrou pour le moniteur (il peut entrer dans le moniteur) uniquement lorsque le processus notifiant sort de la méthode (ou du bloc) synchronized

// VERSION JAVA ANIMÉE DU REPAS DES PHILOSOPHES.

```
import java.applet.*;
import java.util.*;
import java.awt.*;

class Philosophe extends Thread {

    private int Id;
    private DistributeurDeFourchettes df;
    private Fourchette fd, fg;
    public boolean EnTrainDeManger = false;

    public void Debut( int Id, DistributeurDeFourchettes df, Fourchette fd,Fourchette fg ) {
        this.Id = Id;
        this.df = df;
        this.fd = fd;
        this.fg = fg;
    }

    public void run() {
        Fourchette f1, f2;
        while (true) {
            Penser();
            df.Prendre2Fourchettes( Id, fd, fg );
            Manger();
            df.Poser2Fourchettes( Id, fd, fg );
        }
    }

    private void Manger() {
        EnTrainDeManger = true;
        try {
            sleep( (int)(Math.random() * 15 ) );
        } catch (InterruptedException e) {}
        EnTrainDeManger = false;
    }

    private void Penser() {
        try {
            sleep( (int)(Math.random() * 15 ) );
        } catch (InterruptedException e) {}
    }
}


class Fourchette {

    private boolean Occupee = false;

    public void Prendre() { Occupee = true; }

    public void Poser() {Occupee = false; }

    public boolean Libre() {return Occupee == false; }
}
```

```
class DistributeurDeFourchettes extends Thread {

    public synchronized void Prendre2Fourchettes(
            int Philosophe,
            Fourchette f1,
            Fourchette f2 ) {
        while ( f1.Libre()==false || f2.Libre()==false ) {
            try {
                wait();
            } catch(InterruptedException e) {}
        }
        f1.Prendre();
        f2.Prendre();
    }

    public synchronized void Poser2Fourchettes(
            int Philosophe,
            Fourchette f1,
            Fourchette f2 ) {
        f1.Poser();
        f2.Poser();
        notify();
    }
}

public class PhilosopheSpaghetti extends Applet implements Runnable {

    Thread timer = null;
    private Philosophe p[]          = new Philosophe[5];
    private int IdPhilosophe[]      = new int[5];
    private Fourchette f[]          = new Fourchette[5];
    private DistributeurDeFourchettes df        = new DistributeurDeFourchettes();
    private boolean EtatPhilosophe[] = new boolean[5];

    public void init() {
        resize(300,300);            // Set window size
        for ( int i=0 ; i<5 ; i++ ) {
            p[i] = new Philosophe();
            f[i] = new Fourchette();
            EtatPhilosophe[i] = false;
        }
        IdPhilosophe[0] = 0;
        IdPhilosophe[1] = 1;
        IdPhilosophe[2] = 2;
        IdPhilosophe[3] = 3;
        IdPhilosophe[4] = 4;
        p[0].Debut( 0, df, f[0], f[4] );
        p[1].Debut( 1, df, f[1], f[0] );
        p[2].Debut( 2, df, f[2], f[1] );
        p[3].Debut( 3, df, f[3], f[2] );
        p[4].Debut( 4, df, f[4], f[3] );
        p[0].start();
        p[1].start();
        p[2].start();
        p[3].start();
        p[4].start();
    }
```

```
    public void paint(Graphics g) {
        for ( int i=0 ; i<5 ; i++ ) {

            // on efface l'ancien etat
            g.setColor(Color.lightGray);
            if ( EtatPhilosophe[i] )
                g.drawString("p"+i+" mange",20,20+20*i);
            else
                g.drawString("p"+i+" pense",20,20+20*i);
            EtatPhilosophe[i] = p[i].EnTrainDeManger;

            // on affiche le nouvel etat
            g.setColor(Color.darkGray);
            if ( EtatPhilosophe[i] )
                g.drawString("p"+i+" mange",20,20+20*i);
            else
                g.drawString("p"+i+" pense",20,20+20*i);
        }
        cpt++;
        if ( cpt==200 ) {
            g.drawString("Fini",100,60);
            stop();
        }
    }

    private int cpt = 0;

    public void start() {
        if(timer == null) {
            timer = new Thread(this);
            timer.start();
        }
    }

    public void stop() {
        timer = null;
        p[0].stop();
        p[1].stop();
        p[2].stop();
        p[3].stop();
        p[4].stop();
    }

    public void run() {
        while (timer != null) {
            try {Thread.sleep(100);}
            catch (InterruptedException e){}
            repaint();
        }
        timer = null;
    }

    public void update(Graphics g) {
        paint(g);
    }
}
```