PROGRAMACION CONCURRENTE Y DISTRIBUIDA

III.3 Concurrencia con Java: Sincronización

J.M. Drake
L.Barros

Notas:

Recursos de Java para sincronizar threads

- Todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized.
- La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.
- Mecanismos de bloqueo:
  - Métodos synchronized (exclusión mutua).
  - Bloques synchronized (regiones críticas).
- Mecanismos de comunicación de los threads (variables de condición):
  - Wait(), notify(),notifyAll()...
- Cualquier otro mecanismo:
  - Dado que con monitores se pueden implementar los restantes mecanismos de sincronización (Semáforos, Comunicación síncrona, Invocación de procedimientos remotos, etc.) se pueden encapsular estos elementos en clases y basar la concurrencia en ellos.

Lock asociado a los componentes Java.

- Cada **objeto** derivado de la clase object (esto es, prácticamente todos) tienen asociado un elemento de sincronización o lock intrínseco, que afecta a la ejecución de los métodos definidos como synchronized en el objeto:
  - Cuando un objeto ejecuta un método synchronized, toma el lock, y cuando termina de ejecutarlo lo libera.
  - Cuando un thread tiene tomado el lock, ningún otro thread puede ejecutar ningún otro método synchronized del mismo objeto.
  - El thread que ejecuta un método synchronized de un objeto cuyo lock se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.
- Cada **clase Java** derivada de Object, tiene también un mecanismo lock asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados synchronized.

Bloques synchronized.

- Es el mecanismo mediante el que se implementan en Java las **regiones críticas**.
- Un bloque de código puede ser definido como synchronized respecto de un objeto. En ese caso solo se ejecuta si se obtiene el lock asociado al objeto

```
synchronized (object){
    Bloque de estamentos
}
```

- Se suele utilizar cuando se necesita utilizar en un entorno concurrente, un objeto diseñado para un entorno secuencial.

Este mecanismo presenta el inconveniente de las Regiones críticas, y es que los diferentes bloques que interaccionan a través de una región crítica resultan dispersos por múltiples módulos de la aplicación, y ello hace que su mantenimiento sea muy complejo y delicado.

Observese que en Java no hay Regiones Críticas Condicionales, por lo que no se pueden resolver con este mecanismo todos los problemas.

Ejemplo de utilización de bloque synchronized

```
    // Hace que todos los elementos del array sean no negativos
public static void abs(int[] valores){
    synchronized (valores){
        //  Sección crítica
        for (int i=0; i<valores.length; i++){
            if (valores[i]<0) valores[i]= -valores[i];
        }
    }
}
```

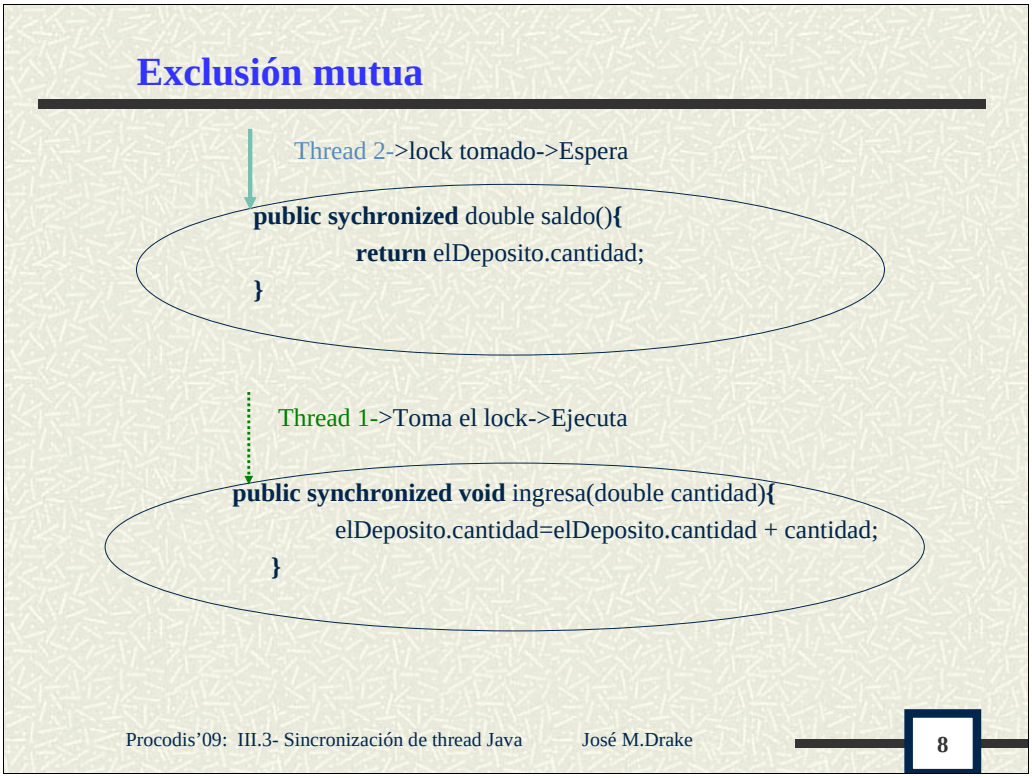
Implementación de Monitores: Métodos synchronized

- Los métodos de una clase Java se pueden declarar como synchronized. Esto significa que se garantiza que se ejecutará con régimen de exclusión mutua respecto de otro método del mismo objeto que también sea synchronized.
- Cuando un thread invoca un método synchronized, trata de tomar el lock del objeto a que pertenezca. Si está libre, lo toma y se ejecuta. Si el lock está tomado por otro thread, se suspende el que invoca hasta que aquel finalice y libere el lock.
- Si el método synchronized es estático (static), el lock al que hace referencia es de clase y no de objeto, por lo que se hace en exclusión mútua con cualquier otro método estático synchronized de la misma clase.

Ejemplo de método synchronized.

```
class CuentaBancaria{
    private class Deposito{
            protected double cantidad;
            protected String moneda =“Euro”
    }
    Deposito elDeposito;
    public CuentaBancaria(double initialDeposito,String moneda){
            elDeposito.cantidad= initialDeposito;
            elDeposito.moneda=moneda; }
    public sychronized double saldo(){return elDeposito.cantidad;}
    public synchronized void ingresa(double cantidad){
            elDeposito.cantidad=elDeposito.cantidad + cantidad;
     }
}
```

Exclusión mutua



Notas:

Consideraciones sobre métodos synchronized.

- El lock es tomado por el thread, por lo que mientras un thread tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto.
- El lock es por cada instancia del objeto.
- Los métodos de clase (static) también pueden ser synchronized. Por cada clase hay un lock y es relativo a todos los métodos synchronized de la clase. Este lock no afecta a los accesos a los métodos synchronized de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobreescribe, este se puede definir como synchronized o nó, con independencia de cómo era y como sigue siendo el método de la clase madre.

Métodos de Object para sincronización.

Todos son métodos de la clase object. **Solo se pueden invocar por el thread propietario del lock (p.e. dentro de métodos synchronized)**. En caso contrario lanzan la excepción **IllegalMonitorStateException**

public final void **wait**() throws InteruptedException

Espera indefinida hasta que reciba una notificación.

public final void **wait**(long timeout) throws InteruptedException

El thread que ejecuta el método se suspende hasta que, o bien recibe una notificación, o bien,transcurre el timeout establecido en el argumento. wait(0) representa una espera indefinida hasta que llegue la notificación.

public final **wait**(long timeout, int nanos)

Wait en el que el tiempo de timeout es 1000000*timeout+ nanos nanosegundos

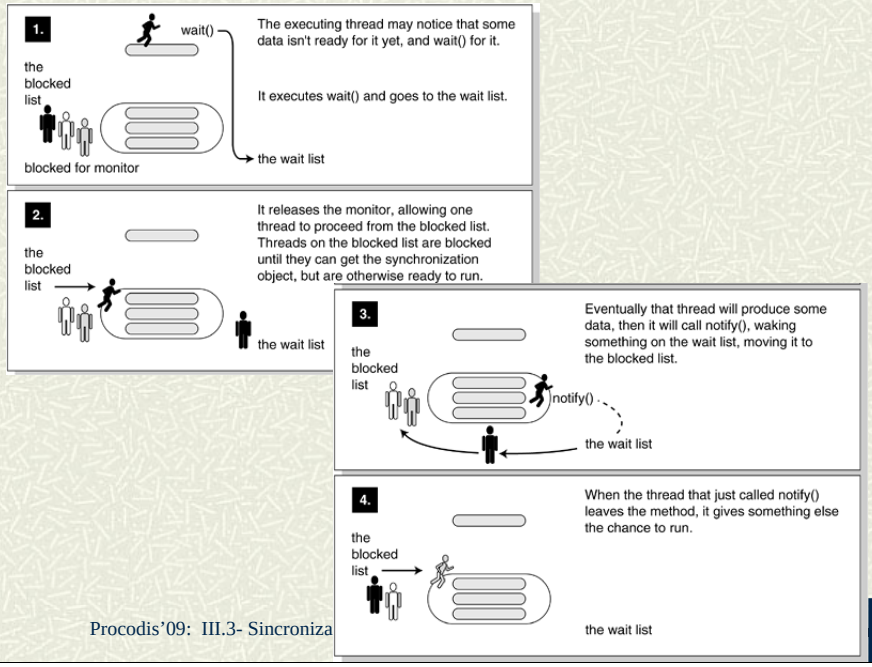
public final void **notify**()

Notifica al objeto un cambio de estado, esta notificación es transferida a solo uno de los threads que esperan (han ejecutado un wait) sobre el objeto. No se puede especificar a cual de los objetos que esperan en el objeto será despertado.

public final void **notifyAll**()

Notifica a todos los threads que esperan (han ejecutado un wait) sobre el objeto.

Protocolo Wait/Notify



Notas:

It's not enough just to say, "Don't run while I am running." We need the threads to be able to say, "OK, I have some data ready for you," and to suspend themselves if there isn't data ready.

Uso de los métodos wait().

- Debe utilizarse siempre dentro de un método synchronized y habitualmente dentro de un ciclo indefinido que verifica la condición:

```
synchronized void HazSiCondicion(){
    while (!Condicion)  wait();
    ..... // Hace lo que haya que hacer si la condición es cierta.
}
```

  - Cuando se suspende el thread en el wait, libera el lock que poseía sobre el objeto. La suspensión del thread y la liberación del lock son atómicos (nada puede ocurrir entre ellos).
  - Cuando el thread es despertado como consecuencia de una notificación, la activación del thread y la toma del lock del objeto son también atómicos (Nada puede ocurrir entre ellos).

Uso de los métodos notify().

- Debe utilizarse siempre dentro de un método synchronized:

```
synchronized void changeCondition(){
    ..... // Se cambia algo que puede hacer que la condición se satisfaga.
    notifyAll();
}
```

- Muchos thread pueden estar esperando sobre el objeto:
  - Si se utiliza notify() solo un thread (no se sabe cual) es despertado.
  - Si se utiliza notifyAll() todos los thread son despertados y cada uno decide si la notificación le afecta, o si no, vuelve a ejecutar el wait() (dentro del while).
- El proceso suspendido debe esperar hasta que el procedimiento que invoca notify() o notifyAll ha liberado el lock del objeto.

Ejemplo de sincronización: Establecimiento de una variable.

Thread 1

```
public synchronized guardedJoy() {
   //This guard only loops once for each
   //special event, which may not
   //be the event we're waiting for.

   while(!joy) {
      try {
         wait();
      } catch (InterruptedException e) {}
   }
   System.out.println("Joy and
     efficiency have been achieved!");
}
```

Thread 2

```
public synchronized notifyJoy() {
      joy = true; notifyAll();
}
```

Ejemplo Monitor: Buffer de capacidad finita

```
public interface Buffer {
  public void put(Object obj)
                  throws InterruptedException;
  public Object get()
                  throws InterruptedException;
}

class FixedBuffer implements Buffer{
  Object[] buf;
  int in = 0;
  int out = 0;
  int count= 0;
  int size;
  public FixedBuffer(int size){
    this.size= size;
    buf = new Object[size];
   }
```

```
public synchronized void put(Object obj)
                  throws InterruptedException{
  while (count == size) wait();
  buf[in]=obj;
  count= count +1;
  in=(in+1)%size;
  notifyAll();
  }
public synchronized Object get()
                  throws InterruptedException{
  while (count == 0) wait();
  Object obj= buf[out];
  buf[out]=null;
  count= count -1;
  out= (out+1) % size;
  notifyAll();
  return (obj);
  }
}
```

Semáforo en Java

```
public  class Semaphore{
  int value;

  public Semaphore(int initialValue){
     value = initialValue;
  }
   synchronized public void signal(){
      value= value+1;
      notify();
   }
   synchronized public void await() throw InterruptedException{
     while (value == 0) wait();
      value = value - 1;
   }
}
```

Buffer de capacidad finita basado en semáforos.

```
public interface Buffer {
  public void put(Object obj)
                        throws InterruptedException;
  public Object get() throws
     InterruptedException;
}
class FixedSemaphoreBuffer implements Buffer{
  protected Object[] buf;
  protected int in = 0;
  protected int out = 0;
  protected int count= 0;
  protected int size;

  Semaphore full = new Semaphore(0);
  Semaphore empty;
  FixedSemaphoreBuffer(int size){
    this.size= size;
    buf = new Object[size];
    empty = new Semaphore(size);
   }
```

```
 public void put(Object obj)
                    throws InterruptedException{
   empty.Wait();
   synchronized (this) {
      buf[in]=obj; count= count +1;
     in=(in+1)%size;
    }
    full.Signal()
  }
  public Object get() throws InterruptedException{
     full.Wait();
     synchonized (this) {
        Object obj= buf[out]; buf[out]=null;
        count= count -1;
        out= (out+1) % size;
     }
     empty.Signal();
     return (obj);
   }
}
```

Ejemplo de monitor: Lectores y escritores.

```
public class ReadWriteController {
  Recurso elRecurso;

  int writerWaiting= 0;
  int readerInside= 0;
  int writerInside= 0;

public Recurso get(){
   goInReader();//obtiene acceso
   return (elRecurso);
   goOutReader;//libera recurso
  }

public void write(Recurso newValue){
   goInWriter();//obtiene acceso
   elRecurso=newValue;
   goOutWriter();//libera recurso
}
```

```
public synchronized void goInReader(){
  try{
     while ((writerWaiting+writerInside) != 0) wait();
   }catch (InterruptedException e){};
  readerInside=readerInside+1;}
public synchronized void goOutReader(){
   readerInside=readerInside-1;
   notifyAll();}
public synchronized void goInWriter(){
    writerWaiting=writerWaiting+1;
    try{
      while((writerInside+readerInside)!=0) wait();
    }catch (InterruptedException e){};
    writerWaiting=writerWaiting-1;
    writerInside=writerInside+1; }
public synchronized void goOutWriter(){
    writerInside=writerInside-1;
    notifyAll();}
}
```


Problema de Lectores y Escritores:

Se desea diseñar un monitor que permita el acceso a un recurso compartido en dos modos, como lector y como escritor.

Se puede permitir el acceso concurrente al recurso como lector, pero se ha de evitar que cuando acceda un escritor, pueda concurrir con él, otro escritor a lector.

Se utiliza el protocolo que establece que cuando un escritor espera para acceder, no se permita el acceso de un nuevo cliente hasta que la sala esté vacia, esto es hayan salido todos los lectores (si los hubiera).

Ejercicio I: ¿Qué ocurrirá si lo ejecutamos?

```
public class WaNot{
int i=0;
public static void main(String argv[]){
      WaNot w = new WaNot();
      w.amethod();
      }

      public void amethod(){
      while(true){
            try{
                  wait();
               }catch (InterruptedException e) {}
            i++;
         }//End of while

      }//End of amethod
}//End of class
```


Notas:

Exception in thread "main" java.lang.IllegalMonitorStateException

Ejercicio II:¿Cómo comunicar dos threads?

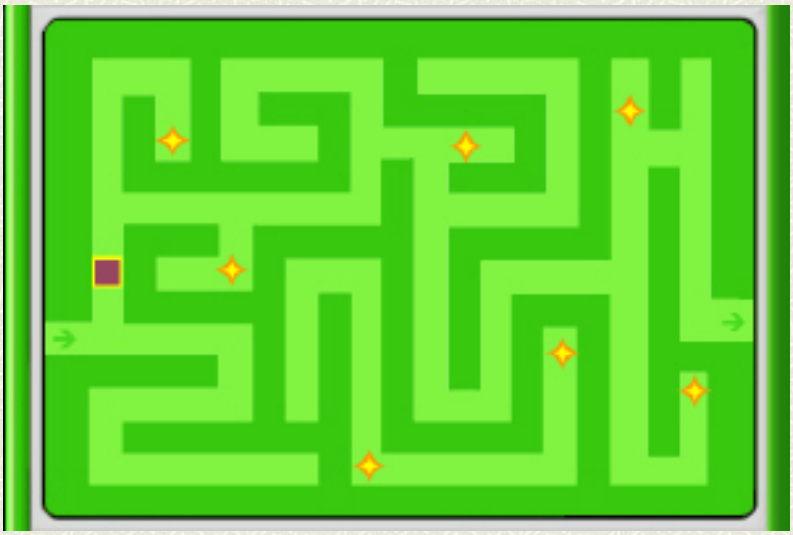
Thread A está esperando que el Thread B le envíe un mensaje:

```
// Thread A
public void waitForMessage() {
      while (hasMessage == false) {
            Thread.sleep(100);
     }
}

// Thread B
public void setMessage(String message) {
      ...
     hasMessage = true;
 }
```

```
// Thread A
public synchronized void waitForMessage() {
     try {
           wait();
     } catch (InterruptedException ex) { }
}

// Thread B
public synchronized void setMessage(String message) {
     ...
     notify();
}
```


Notas:

Maze game I



Notas:

Maze game II. Código sin sincronización

```
public class Maze {
   private int playerX;
   private int playerY;
   public boolean isAtExit() {
      return (playerX == 0 &&
    playerY == 0);
   }
   public void setPosition(int x, int y)
    {
      playerX = x;
      playerY = y;
   }
}
```

Player moves from (1,0) to (0,1):

- 1. Starting off, the object's variables are playerX = 1 and playerY = 0.
- 2. Thread A calls setPosition(0,1).
- 3. The line playerX = x; is executed. Now playerX = 0.
- 4. Thread A is pre-empted by Thread B.
- 5. Thread B calls isAtExit().
- 6.Currently, playerX = 0 and playerY = 0, so isAtExit() returns true!

Notas:

Most of the time, this code works fine. But keep in mind that threads can be pre-empted at any time. Imagine this scenario, in which the player moves from (1,0) to (0,1):

1. Starting off, the object's variables are playerX = 1 and playerY = 0.

2. Thread A calls setPosition(0,1).

3. The line playerX = x; is executed. Now playerX = 0.

4. Thread A is pre-empted by Thread B.

5. Thread B calls isAtExit().

6.Currently, playerX = 0 and playerY = 0, so isAtExit() returns true!

In this scenario, the player is reported as solving the maze when it's not the case. To fix this, you need to make sure the setPosition() and isAtExit() methods can't execute at the same time.

Maze game III. Soluciones de sincronización

```
public class Maze {
   private int playerX;
   private int playerY;
   public synchronized boolean
    isAtExit() {
      return (playerX == 0 &&
    playerY == 0);
   }
   public synchronized void
    setPosition(int x, int y) {
      playerX = x;
      playerY = y;
   }
}
```

==

```
public void setPosition(int x, int y) {
   synchronized(this) {
      playerX = x;
      playerY = y;
   }
}
```


Notas:

The only exception is that the second example has some extra bytecode instructions.

Object synchronization is useful when you need more than one lock, when you need to acquire a lock on something other than this, or when you don't need to synchronize an entire method.

A lock can be any object, even arrays—basically, anything except primitive types. If you need to roll your own lock, just create a plain Object:

```
Object myLock = new Object();
...
synchronized (myLock) {
   ...
}
```

Cuando sí/no sincronizar

- **Cuándo sincronizar**

  En cualquier momento en el que dos o más threads acceden al mismo objeto o campo.

- **Cuándo no sincronizar**

  - **"Sobresincronizar" causa retrasos innecesarios cuando dos o más threads tratan de ejecutar el mismo bloque de código .Por ejemplo, no sincronizar el método entero, cuando sólo una parte necesita ser sincronizada. Poner sólo un bloque sincronizado en esa parte del código:**

```
public void myMethod() {
   synchronized(this) {
      // code that needs to be synchronized
   }
   // code that is already thread-safe
}
```

  - **No sincronizar un método que usa variables locales. Las variables locales son almacenadas en el stack, y cada thread tiene su propio stack, así que, no habrá problemas de concurrencia:**

```
public int square(int n) {
   int s = n * n;
   return s;
}
```


Notas: