BOLETÍN DE ACTIVIDADES – UNIDAD 5

ACTIVIDAD 1 . - Enumere los distintos inconvenientes que ofrecen las primitivas básicas de Java (i.e. monitores) para la sincronización de tareas.

Limitaciones de las primitivas básicas de Java relacionadas con la **exclusión mutua**
Limitaciones de las primitivas básicas de Java relacionadas con la **sincronización condicional**

ACTIVIDAD 2. - A fin de mostrar las principales características de los Locks, que lo diferencian de los monitores clásicos de Java, indique brevemente qué realiza cada uno de los siguientes métodos de la clase **ReentrantLock** (puede consultar la documentación de la librería *java.util.concurrent*)

`ReentrantLock(boolean fair)`	¿Para qué sirve el parámetro "fair"?
`tryLock()`	¿Podemos usar este método para romper la condición de "retención y espera"? ¿Cómo?
`tryLock(long timeout, TimeUnit unit)`	¿Para qué sirve el *timeout*?
`newCondition()`	¿Cuántas veces podríamos utilizar este método dentro de un lock?

ACTIVIDAD 3. OBJETIVO: Ilustrar la utilización de *Locks* y *Conditions*. Compararlos con los monitores clásicos.

ENUNCIADO: Para cada una de las siguientes afirmaciones, indique si se corresponden con características de los Locks de la biblioteca *java.util.concurrent*, de los monitores básicos de Java, o bien de ambos.

	ReentrantLock	**Monitores básicos de Java**
Se ofrecen distintos tipos, con semántica diferente (orientados a exclusión mutua, orientados a resolver el problema de lectores-escritores).		
Dispone de un método que no suspende al invocador si el "lock" ya ha sido cerrado por otro hilo.		
Ofrece un método que sí suspende al invocador si el "lock" ya ha sido cerrado por otro hilo.		
Se puede establecer un plazo máximo de espera al solicitar la entrada al monitor.		
Se puede preguntar por el estado del monitor antes de solicitar acceso al mismo.		
Se puede utilizar más de una variable condición asociada al monitor.		
Se puede cerrar el monitor utilizando un método de la una clase A, y abrirlo en un método de la clase B.		
Se puede cerrar el monitor utilizando un método de la una clase A, y abrirlo con otro método de la clase A.		
Todos los métodos de la clase monitor deben llevar la etiqueta *synchronized.*		
Se pueden interrumpir las esperas de los hilos que desean adquirir el lock.		
El programador no debe preocuparse del cierre y apertura de los locks. La gestión es implícita.		
Cuando se produce una excepción, se debe controlar que en el código asociado a la excepción se abra el lock.		
Todos los hilos que se suspenden en alguna condición van a parar a una única cola (la misma para todos).		

ACTIVIDAD 4. - Dado el siguiente ejemplo de utilización de las clases Locks y Conditions.

```
class BufferOk implements Buffer {
   private int elems, cabeza, cola, N;
   private int[] datos;
   Condition noLleno, noVacio;
   ReentrantLock lock;
   public BufferOk(int N) {
      datos= new int[N];
      this.N=N;
      cabeza = cola = elems = 0;
      lock= new ReentrantLock();
      noLleno=lock.newCondition();
      noVacio=lock.newCondition();
   }
```

```
public int get() {
   int x;
   try {
      lock.lock();
      while (elems==0) {
         System.out.println("consumidor esperando ..");
         try {noVacio.await();}
          catch(InterruptedException e) {}
      }
      x=datos[cabeza];  cabeza= (cabeza+1)%N;
      elems--;
      noLleno.signal();
   } finally {lock.unlock();}
   return x;
}
public void put(int x) {
    try{
      lock.lock();
      while (elems==N) {
         System.out.println("productor esperando ..");
         try {noLleno.await();}
          catch(InterruptedException e) {}
      }
      datos[cola]=x; cola= (cola+1)%N;  elems++;
      noVacio.signal();
   } finally {lock.unlock();}
}
```

a) Explique para qué sirve la clase *BufferOK*. ¿Quién podrá hacer uso de dicha clase?

b) Explique para qué sirven las condiciones *noLLeno, noVacio.*

c) ¿Podrían producirse condiciones de carrera? ¿Por qué?

ACTIVIDAD 5. OBJETIVOS: Ilustrar la utilización de *Locks* y *Conditions*. Compararlos con los monitores clásicos.

ENUNCIADO: En el siguiente código se desea implementar una solución en Java al monitor para el *Crosswalk* visto en las actividades de la Unidad 3. En dicha solución se desea hacer uso de las construcciones *Lock* y *Condition* que se ofrecen en la biblioteca *java.util.concurrent*.

Complete dicho código con las instrucciones necesarias que faltan (atendiendo a los comentarios que se indican), para permitir la utilización adecuada de dichas construcciones.

```
public class Crosswalk {
    private int c, c_waiting, p, p_waiting;
    private Condition OKcars, OKpedestrians;
    private ReentrantLock lock;

    public Crosswalk() {
      c = c_waiting = p = p_waiting = 0;
      lock=new ReentrantLock();
      OKcars=lock.newCondition();
      OKpedestrians=lock.newCondition();
    }

    public void enterC() {
    //cerrar lock



       c_waiting++;
       while (p >0)
//esperar a que los coches puedan pasar


       c_waiting--;
       c++;
  //notificar que los coches podrían pasar


 //abrir lock


   }

public void leaveC() {
//cerrar lock


       c--;
  //notificar que los coches ya pueden pasar


//notificar que los peatones ya pueden pasar

   }
//abrir lock


}
```

```
 public void enterP() {
//cerrar lock


        p_waiting++;
        while ((c>0)||(c_waiting >0))
  //esperar a que los peatones puedan pasar

        p_waiting--;
        p++;

//notificar que los peatones ya pueden pasar


//abrir lock

}


public void leaveP() {
//cerrar lock


        p--;

 //notificar que los coches ya pueden pasar


//notificar que los peatones ya pueden pasar


   //abrir lock



 }

}
```

ACTIVIDAD 6. OBJETIVO: Interpretar el uso de colecciones concurrentes thread-safe (ejemplo: BlockingQueue).

ENUNCIADO: Dado el siguiente ejemplo de utilización de la clase BlockingQueue...

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

a) ¿Qué problema se pretende resolver en este ejemplo? ¿Cuántos hilos hay? ¿Qué representan? ¿Qué información/recurso comparten esos hilos?

b) Si un hilo quiere extraer un ítem de una cola que está vacía, ¿tendrá que esperar? ¿Dónde se controla esto?

c) Si un hilo quiere insertar un ítem en una cola que está llena, ¿tendrá que esperar? ¿Dónde se controla esto?

d) ¿Podrían producirse condiciones de carrera? ¿Por qué?

e) ¿Podemos decir que la cola *BlockingQueue* es Thread-Safe? ¿Por qué?

ACTIVIDAD 7. OBJETIVOS: Describir el funcionamiento de las clases atómicas. Comparar las variables atómicas (ej. AtomicLong) con el uso de monitores.

ENUNCIADO: A continuación se muestra un ejemplo de uso de Variables Atómicas y se compara con la creación de nuestras propias clases. En concreto:

OPCIÓN A: Usando nuestra propia clase

```
class ID {
  private static long nextID = 0;
  public static synchronized long getNext() {
       return nextID++;
  }
}
public class EjCounter extends Thread{
  ID counter;
  public EjCounter(ID c) {counter=c;}
  public void run() {
    System.out.println("counter value: "+
      (counter.getNext()));
  }
  public static void main(String[] args) {
    ID counter= new ID();
    new EjCounter(counter).start();
    new EjCounter(counter).start();
    new EjCounter(counter).start();
  }
}
```

OPCIÓN B: Usando variables atómicas

```
public class EjCounter extends Thread{
  AtomicLong counter;
  public EjCounter(AtomicLong c) {counter=c;}
  public void run() {
    System.out.println("counter value: "+
      (counter.getAndIncrement()));
  }
  public static void main(String[] args) {
    AtomicLong counter=
         new AtomicLong(0);
    new EjCounter(counter).start();
    new EjCounter(counter).start();
    new EjCounter(counter).start();
  }
}
```

a) Analice las dos opciones. ¿Qué es lo que hacen? ¿Cuál será el valor de la variable *counter* en ambos casos? ¿Se pueden producir condiciones de carrera en algún caso?

b) ¿Para qué cree que sirven los siguientes métodos de la clase AtomicLong? Indique cuál sería su instrucción equivalente, haciendo uso de la clase ID definida con la opción A. Para ello, añada nuevos métodos que permitan implementar esa misma funcionalidad.

Método	*Funcionamiento*	*Método equivalente en nuestra clase ID*
counter.addAndGet(5);		
counter.getAndDecrement();		
counter.incrementAndGet();		

ACTIVIDAD 8. OBJETIVOS: Ilustrar el uso de semáforos (*Semaphore*) para la sincronización.

ENUNCIADO: A continuación se muestra un código Java que trata de resolver el problema "Productor-Consumidor con buffer acotado":

```
class Buffer {
   private int head, tail, elems,size;
   private int[] data;
   private Semaphore item;
   private Semaphore slot;
   private Semaphore mutex;

   public Buffer(int s) {
      head=tail=elems=0;  size=s;
      data=new int[size];
      item=new Semaphore(0,true);
      slot=new Semaphore(size,true);
      mutex=new Semaphore(1,true);
   }
```

```
public int get() {
    try {item.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    int x=data[head]; head= (head+1)%size;  elems--;
    mutex.release();
    slot.release();
    return x;
 }
public void put(int x) {
    try {slot.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    data[tail]=x; tail= (tail+1)%size;  elems++;
    mutex.release();
    item.release();
  }
}
```

Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS, justificando su respuesta.

	V/F
El código es incorrecto, pues cada objeto Buffer debería tener un *ReentrantLock* como atributo interno para poder generar semáforos dentro de él. *Justificación:*	
El semáforo slot proporciona sincronización condicional, suspendiendo al hilo productor cuando no haya huecos libres en el buffer. *Justificación:*	
El semáforo item proporciona sincronización condicional, suspendiendo al hilo consumidor cuando no haya elementos en el buffer. *Justificación:*	
El código es incorrecto, pues los métodos put() y get() deberían estar calificados como "synchronized" para poder utilizar los semáforos. *Justificación:*	

ACTIVIDAD 9. OBJETIVOS: Ilustrar el funcionamiento de las barreras, distinguiendo entre *CyclicBarrier* y *CountDownLatch*.

ENUNCIADO: Complete la siguiente tabla de comparación entre estas dos clases. Indique también qué método o métodos ofrecen la funcionalidad solicitada, en cada caso. Para ello, puede emplear la documentación de *java.util.concurrent* para completar su respuesta.

	CyclicBarrier	*CountDownLatch*
¿Lleva un contador (explícito o implícito)?		
¿Permite inicializar el contador en la creación?		
¿Se puede incrementar el contador? ¿Cómo?		
¿Se puede decrementar el contador? ¿Cómo?		
Método que se emplea para que el hilo se quede esperando		
¿Permite ejecutar alguna acción cuando el último hilo llega al punto de sincronización?		
¿Se puede reutilizar?		

ACTIVIDAD 10. OBJETIVOS: Hacer uso de las herramientas proporcionadas por la biblioteca *java.util.concurrent*.

ENUNCIADO: Dado el siguiente programa en Java, en el que la columna de la izquierda de cada línea de código proporciona su número de línea, que puede ser utilizado para refenciarla.

```
1.1  public class Buffer {
1.2    private int store = 0;
1.3    private boolean full = false;
1.4
1.5    public int get()   {
1.6        int value = store;
1.7        store = 0;
1.8        full = false;
1.9        return value;
1.10   }
1.11
1.12
1.13   public void put(int value)    {
1.14       full = true;
1.15       store = value;
1.16   }
       }
2.1
2.2    public class Main {
2.3      public static void main(String[] args)  {
2.4        Buffer c = new Buffer();
2.5        Consumer c1 = new Consumer(c, 1);
2.6        Producer p1 = new Producer(c, 2);
2.7        c1.start();
2.8        p1.start();
2.9        System.out.println("Producer and " +
2.10         "Consumer have terminated.");
2.11     }
       }
```

```
3.1  public class Consumer extends Thread {
3.2    private Buffer b;
3.3    private int number;
3.4    public Consumer(Buffer ca, int id)  {
3.5      b = ca;
3.6      number = id;
3.7    }
3.8    public void run()  {
3.9      int value = 0;
3.10     for (int i = 1; i < 101; i++)
3.11     {
3.12       value = b.get();
3.13       System.out.println("Consumer #" + number
3.14 +
3.15         " gets: " + value);
3.16     }
3.17   }
       }
4.1
4.2  public class Producer extends Thread {
4.3    private Buffer b;
4.4    private int number;
4.5    public Producer(Buffer ca, int id) {
4.6      b = ca;
4.7      number = id;
4.8    }
4.9    public void run()  {
4.10     for (int i = 1; i < 101; i++)
4.11     {
4.12       b.put(i);
4.13       System.out.println("Producer #" + number
4.14 +
4.15         " puts: " + i);
4.16     }
       }
       }
```

Indique qué código debe ser añadido al programa y dónde, haciendo uso de las herramientas de sincronización proporcionadas por la biblioteca *java.util.concurrent*, de modo que:

A. La clase buffer se utiliza en modo *thread-safe*. Importante: haga uso de las clases *ReentrantLock* y *Condition*.

B. El hilo principal debe escribir su último mensaje una vez que los otros hilos hayan acabado. Proporcione aquí al menos **2 soluciones diferentes**, por ejemplo utilizando las clases: *Semaphore*, *CyclicBarrier*, *CountDownLatch*, etc.

ACTIVIDAD 11. OBJETIVOS: Describir el funcionamiento de las variables atómicas y compararlas con los monitores. Ilustrar la utilización de los locks y las condiciones. Ilustrar el uso de barreras.

ENUNCIADO: Se ha definido una clase *Counter* que permite realizar incrementos y decrementos sobre un contador. Además, en la clase *RaceCondition* se dispone del método principal *main*() en el que se lanzan dos hilos (1 incrementador y 1 decrementador), que actúan sobre la misma variable contador. Al realizar estos hilos el mismo número de iteraciones, se espera que el resultado final del contador sea igual a su valor inicial (es decir, el valor 0).

A continuación se muestra el código de las clases *Counter*, *Incrementer*, *Decrementer* y *RaceCondition*.

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

```
public class RaceCondition
{
     public static void main(String[] args)
     {
       Counter c = new Counter();
       int loops=1000;
       System.out.println("Loops "+ loops );

       Incrementer inc = new Incrementer(c, 1, loops);
       Decrementer dec = new Decrementer(c, 2, loops);

       inc.start();
       dec.start();

       System.out.println("Main Thread obtains: "+c.value() );
     }
}
```

```
public class Incrementer extends Thread
{
     private Counter c;
     private int myname;
     private int cycles;
     public Incrementer(Counter count,
         int name, int quantity)
     {
         c = count;
         myname = name;
         cycles=quantity;
     }

     public void run()
     {
       for (int i = 0; i < cycles; i++)
       {
         c.increment();
         try
         {
            sleep((int)(Math.random() *
                   10));      }
         catch (InterruptedException e) { }
       }
       System.out.println("Thread #" +
         myname + " has done "+cycles+
         " increments.");
    }
}
```

```
public class Decrementer extends Thread
{
     private Counter c;
     private int myname;
     private int cycles;
     public Decrementer(Counter count,
        int name, int quantity)
     {
         c = count;
         myname = name;
         cycles=quantity;
     }

     public void run()
     {
       for (int i = 0; i < cycles; i++)
       {
         c.decrement();
         try
         {
                sleep((int)(Math.random() *
                       20));}
         catch (InterruptedException e) { }
       }
       System.out.println("Thread #" +
         myname + " has done "+cycles+
         " decrements.");
     }
}
```

Se desean realizar las modificaciones necesarias en el código para que:

- El hilo principal se espere a la terminación de los otros dos hilos (incrementador y decrementador)
- No se produzcan condiciones de carrera.

A.- Proporcione a continuación dos soluciones distintas para evitar los problemas de condiciones de carrera que puedan producirse, de modo que:

a1) En una solución se empleen métodos sincronizados (y monitores si fuera necesario).

a2) En la otra solución se utilicen variables atómicas.

B.- Modifique las clases que sean necesarias para conseguir que el hilo principal se espere a que terminen los otros hilos, haciendo uso de los siguientes métodos de sincronización (proporcione diferentes soluciones):

b1) Seleccione entre variables condición y el uso del método *join*() de la clase Thread.

b2) Utilizando *CountDownLatch*.

ACTIVIDAD 12. OBJETIVOS: Ilustrar el uso de semáforos (*Semaphore*) para la sincronización. Ilustrar la utilización de los locks y las condiciones. Ilustrar el uso de barreras.

ENUNCIADO: Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS, justificando su respuesta.

	V/F
El objeto *CountDownLatch* es una barrera que una vez abierta ya no puede ser utilizada de nuevo. *Justificación:*	
Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *CountDownLatch* inicializado a 5, sabiendo que los hilos A ejecutan *c.await*() y los hilos B ejecutan *c.countDown*(), todos los hilos A quedarán suspendidos. *Justificación:*	
Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *CyclicBarrier* inicializado a 4, sabiendo que los hilos A ejecutan *c.await*() y los hilos B también ejecutan *c.await*(), algún hilo de la clase A podrá quedarse suspendido indefinidamente. *Justificación:*	
Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *Semaphore* inicializado a 3, sabiendo que los hilos A ejecutan *c.acquire()* y los hilos B ejecutan *c.release*(), todos los hilos B quedarán suspendidos, pues un hilo no puede realizar el método "*release*()" sobre un semáforo si previamente no ha adquirido un permiso de dicho semáforo. *Justificación:*	
Un objeto "c" de la clase *CountDownLatch* se puede utilizar para que M hilos de clase B esperen a que otro hilo A les avise. Para ello, inicializamos "c" a 1, los hilos B usarán *c.await*() y el hilo A llamará a *c.countDown*() una sola vez. *Justificación:*	
Un objeto "c" de la clase *CountDownLatch* se puede utilizar para garantizar exclusión mutua, inicializándolo a 1 en su constructor, y protegiendo la sección crítica entre un *c.await()* a su entrada y un *c.countDown()* a su salida. *Justificación:*	
Para que un hilo A espere hasta que otros N hilos de una misma clase (H1..Hn) hayan ejecutado una sentencia B dentro de su código se puede utilizar un *Semaphore* S inicializado a 0; A invoca *S.release()*, mientras que H1..Hn invocan *S.acquire()* tras la sentencia B. *Justificación:*	