ACTIVITIES UNIT 5

ACTIVITY 1.- List the different disadvantages or limitations of Java basic primitives for task synchronization (i.e. Java monitors).

Limitations of basic Java primitives related with **mutual exclusion:**
Limitations of basic Java primitives related with **conditional synchronization:**

ACTIVITY 2.- Usage of Locks and Conditions. To show the main features of Looks that distinguish them from classic Java monitors, please briefly indicate what each of the following methods of **ReentrantLock** class makes (consult the documentation of *java.util.concurrent* library)

`ReentrantLock(boolean fair)` What does the "*fair*" parameter do?

`tryLock()`
`tryLock(long timeout, TimeUnit unit)`
Can you use this method to break "hold and wait" condition? How? What is the *timeout* for?

`newCondition()` What does this method create? How many times can you use it inside a lock?

ACTIVITY 3. Usage of Locks and Conditions. For each of the following sentences, mark whether it corresponds with Locks features (i.e. features of ReentrantLock) or Java basic monitors (or both).

	ReentrantLock	**Java basic monitors**
It offers a method that does not suspend the caller if the "lock" has already been closed by another thread.		
You can define a maximum waiting time when requesting access to the monitor.		
You can ask for the state of the monitor before requesting access to this monitor.		
You can only use one condition variable linked to the monitor.		
All methods of the monitor must have the *synchronized* label.		
You can interrupt threads that are waiting to acquire the lock.		
The programmer does not need to manage the opening and closing of locks. This management is implicit.		
When there is an exception, you must open the lock in the code related to the exception.		
All threads that need to be suspended for conditional synchronization wait in the same queue, independently of their waiting condition.		
It offers a method that suspends the caller if the "lock" has already been closed by another thread.		

ACTIVITY 4. Usage of Locks and Conditions. Given the following example:

```
class BufferOk implements Buffer {
   private int elems, head, tail, N;
   private int[] data;
   Condition notFull, notEmpty;
   ReentrantLock lock;
   public BufferOk(int N) {
      data= new int[N];
      this.N=N;
      head = tail = elems = 0;
      lock= new ReentrantLock();
      notFull=lock.newCondition();
      notEmpty=lock.newCondition();
   }
```

```
public int get() {
    int x;
    try {
       lock.lock();
       while (elems==0) {
          System.out.println("consumer waiting ..");
          try {notEmpty.await();}
           catch(InterruptedException e) {}
       }
       x=data[head];  head= (head+1)%N;
       elems--;
       notFull.signal();
    } finally {lock.unlock();}
    return x;
}
public void put(int x) {
     try{
       lock.lock();
       while (elems==N) {
          System.out.println("producer waiting ..");
          try {notFull.await();}
           catch(InterruptedException e) {}
       }
       data[tail]=x; tail= (tail+1)%N;  elems++;
       notEmpty.signal();
    } finally {lock.unlock();}
}
```

Indicate whether the following statements are True (T) or False (F). Justify your answers.

1.	In order that the *BufferOk* class can be used by several threads in a secure way (*thread-safe*) we must label its methods with "*synchronized*".	
2.	The *BufferOk* class controls the access to the buffer, so that both the readings and the writings on this buffer are done in mutual exclusion.	
3.	The *notFull* condition serves to control if the buffer is empty, so that if a thread executes the *get*() method and the buffer is empty, it will be suspended in that condition.	
4.	In this code, race conditions can be produced, for example in the use of the variable int x, which is declared inside the *get*() method.	
5.	In order for the *BufferOk* class to be considered as a Java monitor, only a single condition variable could have been declared.	

ACTIVITY 5. Usage of Locks and Conditions. Using the following code, we would like to implement a solution in Java to the monitor for the Crosswalk, using *Lock* and *Condition* constructions provided by the *java.util.concurrency* library. Complete the following code with the needed instructions (according to the comments that are provided) to allow the right use of these constructions.

```
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;


    }

    public void enterC() {


       while (p >0){
       //cannot cross


       }
      //can cross
       c++;
     //notify that cars can now cross


   }

public void leaveC() {



       c--;
  //notify that cars can now cross



  //if needed, notify that pedestrians
  //can now cross


   }



}
```

```
 public void enterP() {



        while ((c>0)||(c_waiting >0)){
         //wait


        }
        p++;

    // notify that pedestrians can now cross



  }


public void leaveP() {


        p--;

//if needed, notify that cars can now cross



//if needed, notify that pedestrians
//can now cross




 }

}
```

ACTIVITY 6. Use of thread-safe concurrent collections. Given the following example of usage of BlockingQueue class:

```
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) What is the problem to be solved? How many threads are there? What do they represent? Which information or resource is shared by threads?

b) If a thread wants to consume an item from an empty queue, will it have to wait? Where is this controlled?

c) If a thread wants to insert an item into a full queue, will it have to wait? Where is this controlled?

d) Can there be race conditions? Why?

ACTIVITY 7 Atomic variables. Given the following code, that offers two different options for solving the same problem:

OPTION A: Using our own class

```
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();
  }
}
```

OPTION B: Using atomic variables

```
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) Discuss the two options. What do they do? Which is the final value of the *counter* variable in both cases? Can they cause *race conditions* in any case?

b) What do you think the following methods of **AtomicLong** class are for? Which is their functionality? Explain which would be its equivalent statement making use of the ID class defined in option A, adding new methods if needed.

Explain what this method does	*Implement an equivalent method in the "ID class"*
counter.addAndGet(5);	
counter.getAndDecrement();	
counter. incrementAndGet();	

ACTIVITY 8. Usage of Semaphores. Given the following sentences, indicate whether they are True or False.

```
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();
  }
 }
```

	T/F
This code is wrong, because each Buffer object should have a *ReentrantLock* in order to create semaphores inside it.	
The *slot* semaphore provides conditional synchronization, suspending the producer thread when there is no free space in the buffer.	
The *item* semaphore provides conditional synchronization, suspending the consumer thread when it wants to consume and there are no elements in the buffer.	
The code is wrong, because methods *put*() and *get*() must be qualified as "synchronized" to use semaphores in a safe-way.	

ACTIVITY 9. CyclicBarrier and CountDownLatch. Compare these two classes:

	CyclicBarrier	*CountDownLatch*
Does it have a counter? Is it explicit or implicit?		
Does it allow initializing the counter when the class is created?		
Can the counter be incremented? How?		
Can the counter be decremented? How?		
Method used to make the thread wait		
Does it allow executing any action when the last thread reaches the synchronization point?		
Can it be reused? How?		

ACTIVITY 10. ***java.util.concurrent*** **package.** Given the following Java program:

```
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 public void put(int value)    {
1.13       full = true;
1.14       store = value;
1.15   }
1.16 }

2.1  public class Main {
2.2     public static void main(String[] args)  {
2.3        Buffer c = new Buffer();
2.4        Consumer c1 = new Consumer(c, 1);
2.5        Producer p1 = new Producer(c, 2);
2.6        c1.start();
2.7        p1.start();
2.8        System.out.println("Producer and " +
2.9          "Consumer have terminated.");
2.10    }
2.11 }
```

```
3.1  public class Consumer extends Thread {
3.2     private Buffer b;
3.3     private int number;
3.4    private semaphore sem;
3.5     public Consumer(Buffer ca, int id, ...)  {
3.6        b = ca;
3.7        number = id;
3.8     }
3.9     public void run()  {
3.10       int value = 0;
3.11       for (int i = 1; i < 101; i++) {
3.12          value = b.get();
3.13         System.out.println("Consumer #" + number +
3.14         " gets: " + value);
3.15       }
3.16    }
3.17 }

4.1  public class Producer extends Thread {
4.2    private Buffer b;
4.3    private int number;
4.4    public Producer(Buffer ca, int id) {
4.5        b = ca;
4.6        number = id;
4.7    }
4.8    public void run()  {
4.9        for (int i = 1; i < 101; i++)  {
4.10          b.put(i);
4.11          System.out.println("Producer #" + number +
4.12          " puts: " + i);
4.13       }
4.14     }
4.15 }
4.16
```

Explain which code must be added to the program and where, making use of the synchronization tools provided by java.util.concurrent library, so that:

A. Buffer class is used in thread-safe way. Important: make use of *ReentrantLock* and *Condition* classes.

B. The main thread must write its last message once the other threads have finished. Please, provide at least **2 different solutions**, for example using Sempahore, CyclicBarrier, CountdownLatch classes, etc.

ACTIVITY 11. ***java.util.concurrent*** **package**. Given the following code:

```
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 = 1; 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 = 1; i < cycles; i++)
       {
         c.decrement();
         try
         {
                sleep((int)(Math.random() *
                       20));}
         catch (InterruptedException e) { }
       }
       System.out.println("Thread #" +
         myname + " has done "+cycles+
         " decrements ");
     }
}
```

We would like to make the necessary modifications in the code to achieve that:

- The main thread waits for the completion of the other two threads (*incrementer* and *decrementer*).
- There are no race conditions.

A) Provide **<u>two different solutions</u>** to take into account that no race conditions can be produced, so that:

a1) In one solution you must employ **synchronized methods**.

a2) In the other solution, you must employ atomic variables.

B) Modify all needed classes to make that the main thread waits for the completion of the other two threads, **<u>giving two different solutions</u>**:

b1) Using the ***join***() method of the Thread class and/or **condition variables**.

b2) Using a **CountDownLatch.**

ACTIVITY 12. ***java.util.concurrent***. Rewrite the sentences to make all them **true**:

1. For each lock of type *ReentrantLock* you can create as many variable conditions related to this lock as needed, using *CreateCondition*() method of this class.
2. If you use the "ReentrantLock" class to implement a Java monitor, all methods using "Condition" objects must be protected with the "synchronized" label.
3. You can make M threads of class A wait until another thread of class B notifies them, using an object "c" of class *CountDownLatch*. Therefore, you will initialize "c" to M, threads A will call *c.await()* and thread B will call *c.countDown()* a total of M times.
4. You can make a thread A wait until other N threads of class B finish, using a *Semaphore* S initialized to N; thread A invokes *S.acquire()* whereas threads B invoke *S.release()* before finishing.
5. Among other properties, a *CyclicBarrier* differs from a *CountDownLatch* barrier in that you can specify in the constructor of the cyclic barrier a *run*() method that will be executed just when the barrier opens.
6. The *CountDownLatch* objects ensure that the *await*() method will always suspend the invoking thread.
7. You can implement a right solution for the critical section problem using a ReentrantLock "rl", being "*rl.lock*()" the input protocol and "*rl.unlock()*" the output protocol.
8. If several threads make use of the same AtomicLong object, you must protect the methods that this object offers with the "synchronized" label, to avoid race conditions.
9. The threads that decrement the counter of the "CountDownLatch" barrier get suspended until the counter is zero.
10. CountDownLatch barriers are created once and they can be used as many times as needed. When the counter of the barrier gets to zero, automatically its initial value is restored.
11. You can use a *Semaphore S* initialized to 0 to make a thread A wait until other N threads (H1..HN) of same class have executed a sentence B inside their code. Thread A invokes N times *S.acquire(),* whereas H1..HN invoke *S.release()* after sentence B.

ACTIVITY 13. Usage of barriers (CyclicBarrier). Given the following code:

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
   int myRow;
   Worker(int row) { myRow = row; }
   public void run() {
    while (!done()) {
     processRow(myRow);
     try {barrier.await();}
     catch (InterruptedException e) {return;}
     catch (BrokenBarrierException e) {return;}
    }
   }
  }
  public Solver(float[][] matrix) {
   data = matrix;
   N = matrix.length;
   barrier = new CyclicBarrier(N,
     new Runnable() {
     public void run() {
        mergeRows(...);
     }
   });
   for (int i = 0; i < N; ++i)
    new Thread(new Worker(i)).start();
   waitUntilDone();
  }
 }
```

a) Which is the main thread? What does it do?

b) How many secundary threads are created? Where is the code that they execute?

c) How many barriers are? Of which type? Who creates them and how?

d) Who waits at the barrier? Which method is used for waiting at the barrier?

e) When is the barrier opened? Which method is used for opening it?

ACTIVITY 14.- Usage of Semaphores. Given the following code:

```
import java.util.concurrent.Semaphore;

class Process2 extends Thread {
      private int id;
      private Semaphore sem;

      private void busy() {
            try {sleep(new java.util.Random().nextInt(500));}
            catch (InterruptedException e) {}
      }
      private void msg(String s)      {System.out.println("Thread " + id + s);}
      private void noncritical()      {msg(" is NON critical"); busy();}
      private void critical()         {msg(" entering CS"); busy(); msg(" leaving CS");}

      public Process2(int i, Semaphore s) {id = i; sem = s;}
      public void run() {
            for (int i = 0; i < 2; ++i) {
                  noncritical();
                  try {sem.acquire();} catch (InterruptedException e) {}
                  critical();
                  sem.release();
            }
      }
      public static void main(String[] args) {
            Semaphore sem = new Semaphore(1, true); //permisos, fair?
            for (int i = 0; i < 4; i++)
                  new Process2(i, sem).start();
      }
}
```

a) Explain the behaviour of this code. What would happen if the semaphore *sem* is initialized with a counter value equal to 3?. In which cases will it be interesting to have a semaphore counter greater than 1? Think in some examples.

b) Next code shows usage of semaphores for establishing a synchronization point between threads. Discuss this code explaining what it does and/or show a trace of execution.

```
import java.util.concurrent.Semaphore;

class ProdCons extends Thread {
    static final int N=6; // buffer size
    static int head=0, tail=0, elems=0;
    static int[] data= new int[N];
    static Semaphore item= new Semaphore(0,true);
    static Semaphore slot= new Semaphore(N,true);
    static Semaphore mutex= new Semaphore(1,true);

    public static void main(String[] args) {
        new Thread(new Runnable() { // producer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    put(i);
                }
            }
        }).start();
        new Thread(new Runnable() {// consumer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    System.out.println(get());
                }
            }
        }).start();
    }

    private static void busy() {
        try {sleep(new java.util.Random().nextInt(500));}
        catch (InterruptedException e) {}
    }

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

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

ACTIVITY 15. Let us assume there is a "thread-safe" Buffer class with capacity for 100 elements, whose *put*() method allows inserting an "int" element and whose *get*() method extracts an element. Giving the following code:

```
public class Main{
  public static void main(String[] args){
    Buffer d=new Buffer();
    SENTENCE A
    Worker w1=new Worker(c,d,1);
    Worker w2=new Worker(c,d,2);
    Worker w3=new Worker(c,d,3);
    w1.start();
    w2.start();
    w3.start();

    try{
      SENTENCE B
    }catch(InterruptedException e){}
    System.out.println("This is a race!");
 }
}
```

```
public class Worker extends Thread{
  SENTENCE C
  private Buffer buf;
  private int number;

  public Worker (SENTENCE D, Buffer c, int d)
  {  obj=a;
     buf=c;
     number=id;
    }

  public void run(){
    for (int i=1; i<=100; i++)
    {  buf.put(number*100+i);
       if (i==50){
         try{   SENTENCE E
            }catch(InterruptedException e){} };
       System.out.println(buf.get());
    }//for
  } //run
}
```

And given the following possible labels to fill up the code:

ID	LABEL	ID	LABEL
1	Semaphore c=new Semaphore(0);	2	Semaphore c=new Semaphore(4);
3	CyclicBarrier c=newCyclicBarrier(3);	4	CyclicBarrier c=new CyclicBarrier(4);
5	CountDownLatch c=new CountDownLatch(0);	6	CountDownLatch c=new CountDownLatch(3);
7	CountDownLatch c=new CountDownLatch(4);	8	int c=0;
9	c.await();	10	c.acquire();
11	c.release();	12	c.countDown();
13	c.acquire(); c.acquire(); c.acquire();	14	w1.join(); w2.join(); w3.join();
15	private Semaphore obj;	16	private CyclicBarrier obj;
17	private CountDownLatch obj;	18	private int obj;
19	Semaphore a	20	CyclicBarrier a
21	CountDownLatch a	22	int a
23	obj.release();	24	obj.acquire();
25	obj.await();	26	obj.countDown();

If the main thread must write its message when each of the Worker threads shown in the code **have completed at least 50 iterations**:

1. In SENTENCE B you can use the *join* method of Thread class. That is, label 14.	
2. If you use semaphores for coordinating Workers threads and the main thread, you must use the following combination of sentences-labels: A-1, B-13, C-15, D-19, E-11	

3. If you use barriers for coordinating Workers threads and the main thread, you can use the following combination of sentences-labels: A-3, B-9, C-16, D-20, E-26.	
4. If you use barriers for coordinating Workers threads, you can use the following combination of sentences-labels: A-7, B-12, C-17, D-21, E-25.	
5. If you use *CyclicBarriers*, you must initialize them to value 4 and use label 25 for SENTENCE E, among other issues.	

ACTIVITY 16. Given the following code:

```
public class BufferMin{
     private int[] data;
     private int elems, head, tail, N;
     Condition notFull, notMinimum;
     ReentrantLock rl;

 public BufferMin(int N){
    data= new int[N];
    this.N=N;
    head=tail=elems=0;
    rl=new ReentrantLock();
    notFull=rl.newCondition();
    notMinimum=rl.newCondition();
}
```

```
public int getPairs()
{  int x, y, sum;
    try{
      rl.lock();
      while(elems<2){
                try { notMinimum.await();
                } catch(InterruptedException e) {} }
     x=data[head];  head=(head+1)%N;
     y=data[head];  head=(head+1)%N;
     sum=x+y;
     elems=elems-2;
     notFull.signal();
 } finally { rl.unlock();}
  return (sum);
}

public void insert(int x) {
 try{
     rl.lock();
     while (elems==N) {
       try {notFull.await();
       }  catch(InterruptedException e) {} }
    data[tail]=x; tail= (tail+1)%N; elems++;
    if (elems>=2)
      notMinimum.signal();
 } finally {rl.unlock();}
}
```

1. You must label all methods of the *BufferMin* class with "synchronized" to be considered as a *thread-safe* class.	
2. If several threads use *getPairs*() and *insert*() methods of the *BufferMin* class, there can be race conditions when accessing concurrently to the *data* parameter.	
3. *BufferMin* class can only be considered as a monitor in Java if you declare a single variable condition.	
4. Mutual exclusion of *BufferMin* class can be implemented with *Semaphore mutex* object initialized to 1, and conditional synchronization with the condition variables *notFull* and *notMinimum*.	
5. *BufferMin* could be directly implemented with a *BlockingQueue* object, using its *take*() and *put*() methods in a suitable way, since this is a producer-consumer kind problem.	