This exam has a maximum duration of 2 hours.

This exam has a maximum score of **10 points**, equivalent to **3.5** points of the final grade for the course. It contains questions of theoretical units and lab sessions. Indicate, for each of the following **58 statements**, if they are true (T) or false (F). **Each answer is worth: right= 10/58, wrong= -10/58, empty=0.**

Important: **first 3 errors do not penalize,** so they will be equivalent to an empty answer. From the 4th error (inclusive), the decrement for wrong answers will be applied.

THEORY QUESTIONS

Regarding the concept of critical section (CS):

	T/F
1. We denominate Critical Section to any fragment of code that accesses variables shared between threads.	T
2. The use of correct input and output protocols converts a Critical Section into an atomic action.	T
3. The input and output protocols of the Critical Section guarantee the properties of mutual exclusion, no pre-emption, and bounded waiting.	F
4. The use of correct input and output protocols prevents the appearance of race conditions.	T

Regarding the concurrent programming in Java:

5. Methods of an object that are not declared as synchronized can be executed while another thread is using the same object.	T
6. The compiler guarantees that each method with the synchronized label has its own Lock.	F
7. A shared object is any object maintained in the heap and referenced from 2 or more threads.	T
8. The code to be executed by a thread is contained in its `start` method.	F

Regarding the *lock* concept:

9. Only the owner (i.e. the thread currently running the Critical Section protected by the lock) can open the lock.	T
10. When a thread tries to close a lock that has already been closed by another thread, it goes to a suspended state.	T
11. Given a critical section protected by a lock, if the lock is opened at the output protocol, then all threads locked at the input protocol are allowed to enter in this critical section.	F

Regarding the possible states of a thread:

12. The only way that a thread that has executed `sleep(x)` can go to ready-to-run is when `x` milliseconds has passed.	F
13. When we create a thread in Java, this thread goes directly into ready-to-run state.	F
14. The execution of `yield()` causes the corresponding thread to go from running to suspended.	F

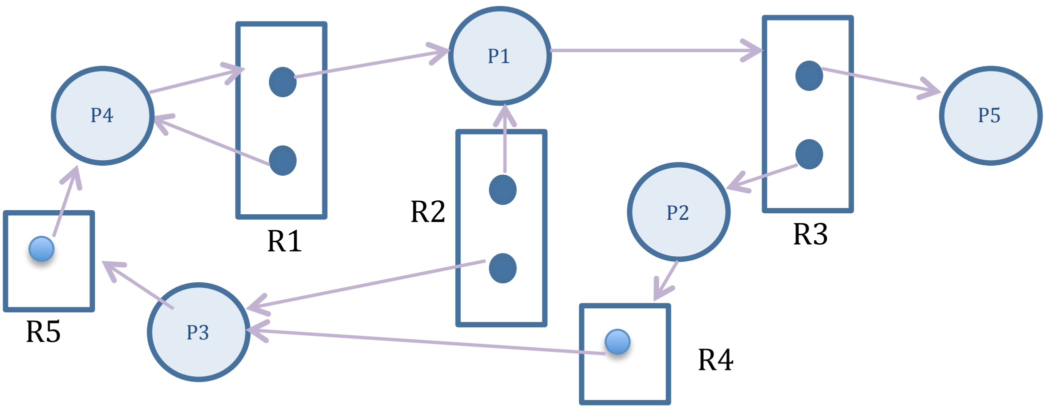
Regarding the definition of concurrent programming:

15. A sequential program may consist of different threads with logical parallelism.	F
16. The problems of communication and synchronization between activities must be resolved so that the activities of an application can cooperate with each other.	T
17. One of the disadvantages of the concurrent programming is the difficulty to adapt to changes on demand (i.e. few scalability).	F

Let's suppose that we have the following example of a server for the sale of tickets, which uses: (i) a shared constant *pvpTicket* with the sale value of the tickets; and (ii) a shared variable *numTickets* that indicates the amount of unsold tickets. In addition, several threads are created that sell tickets while *numTickets*> 0. Each thread consists of a loop in which there is a random wait and then sells a ticket and displays a message on the screen. The loop ends if there are no tickets left. The main thread shows a final message when there are no tickets available for sale.

18. This is not an example of concurrent programming, since the threads do not cooperate with each other.	F
19. The access to *pvpTicket* is a Critical Section (it can generate interferences).	F
20. For the program to be correct, a lock has to be used to protect access to *numTickets*.	T
21. If we want the main thread to wait for the rest of the threads to finish, we can achieve this by using a set of invocations to the `join()` method of the Thread class.	T

Given the following resource allocation graph:



22. In this graph there is a unique safe sequence.	T
23. We can affirm that in the current graph there is no deadlock.	T
24. If process P5 requests an instance of R4, then all processes will be in a deadlock.	T

In a system there are eight processes in execution: P0, P1 ... P7, that want to use the resources R0, R1, ..., R7, each of them with a single instance, where the resources are used in mutual exclusion and nobody can appropriate resources assigned to another.
The execution profile of a Pi process is as follows:

```
while (TRUE){
     Request(R(i+1) % 8));      //Sentence A
     Request(Ri);              //Sentence B
     UsingResources();
     Release(Ri);
     Release(R(i+1) % 8));
};
```

25. Deadlocks cannot occur since there is an even number of processes.	F
26. If all processes execute their "Sentence A" before either of them executes their "Sentence B", a situation is reached in which all the Coffman conditions are met.	T
27. If the execution profile of the even processes is modified, so that the sentences A and B are exchanged, the Coffman condition of "circular wait" is broken.	T
28. If the profile of a single process is modified so that it first performs its "Sentence B" and then its "Sentence A", and the rest of the processes maintain the profile indicated above, a deadlock would never occur.	T

Regarding the usage of *java.util.concurrent* tools:

29. With each lock of type *ReentrantLock*, you can create as many condition variables associated to this lock as required, indicating in the constructor of the *Condition* class the lock to which it belongs.	F
30. If the *ReentrantLock* is used to implement a Java monitor, the methods where the *Condition* objects are used must appear protected with the *synchronized* label.	F
31. For M threads of class A to wait for another thread of class B to notify them, a "c" object of *CountDownLatch* class can be used. To do this, we initialize "c" to 0, threads A will use *c.await()* and thread B will call *c.countDown()* a total of M times.	F
32. For a thread A to wait until other N threads of class B execute a sentence X, we can use a *Semaphore* S initialized to 0; thread A invokes N times *S.acquire()*, while threads B invoke *S.release()* after sentence X.	T
33. Among other properties, a *CountDownLatch* differs from a *CyclicBarrier* barrier in that a *run*() method can be specified in the *CountDownLatch* constructor, which will be executed just when the barrier is opened.	F
34. If several threads use the same *AtomicInteger* object, the methods that this object offers must be protected with the *synchronized* label, in order to avoid race conditions.	F

Given the following code:

```
import java.util.concurrent.BlockingQueue;
public class Producer extends Thread
{   private final BlockingQueue queue;
    private int prodname;
    public Producer(BlockingQueue q, int name)
    {    queue=q;
         prodname = name;
    }
    public void run()
    {    int elem;
         for (int i = 1; i < 11; i++)
         {   elem=i+(prodname*10);
             try{ queue.put(Integer.valueOf(elem));
             }catch (InterruptedException e) { }

             System.out.println("Producer #" + prodname + " puts: " + elem);
             try {   sleep((int)(Math.random() * 100));
              } catch (InterruptedException e) { }
    }  }
}

public class Consumer extends Thread
{    private final BlockingQueue queue;
     private int cname;
     public Consumer(BlockingQueue q, int name)
     {   queue=q;
         cname = name;
     }

     public void run()
     { int value = 0;
       for (int i = 1; i < 11; i++)
       { try{ value = (Integer)queue.take();
            }catch (InterruptedException e) { }
         System.out.println("Consumer #" + cname + " gets: " + value);
         try { sleep((int)(Math.random() * 100));            }
            catch (InterruptedException e) { }
     }   }
}

public class ConsumerProducerProblem
{
     public static void main(String[] args)
     {   BlockingQueue q =new ArrayBlockingQueue(2);
         Consumer c1 = new Consumer(q, 1);
         Producer p1 = new Producer(q, 1);
         Consumer c2 = new Consumer(q, 2);
         Producer p2 = new Producer(q, 2);

         c1.start(); c2.start();
         p1.start(); p2.start();
     }
}
```

35. It is required to use the *synchronized* label in the methods of the *Producer* and *Consumer* classes to avoid race conditions.	F
36. Executing this code will not cause deadlocks.	T
37. The shared object "q" offers its "`put`" and "`take`" methods with access in mutual exclusion and guarantees conditional synchronization.	T

Given the following code that shows a monitor to manage access to a parking lot that has an exit and two gates (North and South). Assume that the threads (cars) invoke the `enterX()` and `exit()` methods, before and after accessing the parking respectively:

```
Monitor manageParkingLot{
  int n = 10;
  int nCars = 0;
  int nWaitNorth = 0;
  int nWaitSouth = 0;
  boolean turnS = true;
  boolean turnN = true;
  Condition freeSpace;

entry void enterSouth()
{
 nWaitSouth++;
 while(nCars == n || !turnS) freeSpace.wait();  freeSpace.notify();
 nWaitSouth--;
 nCars ++;
 if (nCars == n) {turnS = false; turnN = true;}
}

entry void enterNorth()
{
 nWaitNorth++;
 while(nCars == n || !turnN) freeSpace.wait(); freeSpace.notify();
 nWaitNorth--;
 nCars ++;
 if (nCars == n) {turnS = true; turnN = false;}
}

entry void exit(){

 if (nCars < n ||
    (nCars == n  && (nWaitNorth == 0 || nWaitSouth == 0 )))   {
             turnN = true;
             turnS = true;
 }
 nCars --;
 freeSpace.notify();
}
}
```

38. With the Lampson-Redell model, the first car entering the parking lot has used the north gate.	F
39. With the Lampson-Redell model, in the queue of the condition variable *freeSpace* there may be simultaneously waiting cars that want to enter by the north gate and cars that want to enter by the south gate.	T
40. This monitor cannot be implemented with the Hoare model because in both `enterNorth()` and `enterSouth()`, we must use "`if`" instead of "`while`".	F
41. With the Brinch-Hansen variant, more than 10 cars can enter the parking lot.	T

In a banking system there are current accounts with several associated holders. Analyze the following proposed program to manage a current account that can be accessed by several holders simultaneously:

```
public class CurrentAccount {
 private AtomicInteger balance = new AtomicInteger(0);
 public void deposit(int  n) {
   balance.addAndGet(n);
 }
 public void refund(int  n) {
   balance.addAndGet(-n);
 }
 public int getBalance() {
  return balance.get();
 }
}
```

42.It is necessary to use the `synchronized` label at `deposit` and `refund` methods to avoid race conditions.	F
43.It is necessary to use the `synchronized` label in all methods to avoid race conditions.	F
44.The `balance` attribute may be left in an inconsistent value when the operations are performed.	F

We want to implement a Java monitor that provides the basic functionality of the *CountDownLatch*, with the *countDown* and *await* methods.

```
public class MyCountDownLatch {
 private int n;
 public MyCountDownLatch(int n0) {
  n=n0;
 }
 public synchronized void countDown() {
   n--;
   notifyAll();
 }
 public void await() throws InterruptedException {
   while (n > 0) wait();
 }
}
```

45.This solution is not correct because the `notifyAll`() must be performed only when the counter reaches zero.	F
46.For this solution to be correct, simply add the *`synchronized`* label in the `await` method.	T
47.After opening the barrier, the value of the initial counter must be reset to "n0".	F

Regarding monitors:

48. A monitor that follows the Lampson-Redell model suspends the thread that has invoked `c.notify()`, being "c" a condition variable, and activates one of the threads that called `c.wait()` before.	F
49. A monitor that follows the Brinch-Hansen model requires any `notify()` invocation to be the last statement in the monitor method in which such invocation appears.	T
50. A monitor that follows the Hoare model suspends (and leaves in a special queue) the thread that invoked `c.notify()`, being "c" a condition variable, and activates one of the threads that called `c.wait()` before.	T

LAB. PRACTICES (8 STATEMENTS)

Regarding practice 1 "Shared use of a pool", where we have the following cases:

Pool0	Free access to the pool (no rules)
Pool1	Kids cannot swim alone (instructor must be with them in the pool)
Pool2	There is a maximum of kids per instructor
Pool3	There is a maximum pool capacity
Pool4	If there are instructors waiting to exit the pool, kids cannot enter into the pool

1. In Pool4, when a child leaves the pool invoking `kidRests()` it would only be necessary to make a `notifyAll()` if there is an instructor waiting to leave.	F
2. In Pool2, when a child leaves the pool invoking `kidRests()` it would only be necessary to make a `notifyAll()` if there is an instructor waiting to leave.	F
3. In Pool3, if we add a new method that allows us to consult the current state of the pool, this method requires the `synchronized` label.	T

Regarding practice 2 "Dining philosophers", where we have the following versions:

Version 1	Asymmetry (all but last)
Version 2	Asymmetry (even/odd)
Version 3	Both or none
Version 4	Capacity of the table

4. Deadlocks may occur in the solution for Version 2 if the delay (N) for picking up the forks is extremely high.	F
5. Any deadlock solution based on asymmetry requires that there are the same number of philosophers of both `Philo` and `LefthandedPhilo` classes.	F
6. All solutions developed in this practice break some Coffman's condition.	T

Regarding practice 3 "The ants problem", where we have the following versions:

Activity 1	ReentrantLock with one condition variable related to the territory
Activity 2	ReentrantLock with a condition variable for each cell of the territory

7.	If the ants could move diagonally across the territory, there would be no risk of deadlock.	F
8.	In activity 2, it is necessary to perform a `signalAll` on the condition variable of the cell that you are leaving and another one on the condition variable of the cell to which you are going.	F