# ACTIVITIES UNIT 1

**QUESTION 1.** Given the following sentences, modify them as needed to make them all **TRUE**.
*Note*: there can be sentences initially true.

| |
|---|
| 1. A concurrent program is formed by a collection of activities (threads) that are executed in parallel, independently and without any type of communication among them. |
| 2. A process with only one executing thread, but that is executed in a processor with four cores, is an example of a concurrent program. |
| 3. A concurrent program needs to be executed on machines with more than one processor (or with a processor with several cores), to allow its tasks to be concurrent. |
| 4. One of the advantages of concurrent programming is that it allows improving the debugging of applications, with respect to sequential programming, offering an easy, simple debugging. |
| 5. Concurrent programming allows programming in a more direct way (i.e. with greater semantic gap) those applications where there are multiple simultaneous activities. |
| 6. In Java you can only create threads if you use the *java.util.concurrent* package. |
| 7. The code to execute for each thread must be contained in its *start*() method. |
| 8. The *t.run*() statement allows you to launch a thread and execute its *run*() method concurrently. |
| 9. To define a thread of execution in Java, we must define some instance of a class that implements the *Thread* interface. |
| 10. To assign a name to a thread, you can pass a string as an argument in its constructor. |
| 11. *Thread.yield*() causes a thread to go from the 'prepared' state to the 'suspended' state. |
| 12. The *setName*() method of the Thread class always associates a name with the thread that is currently running. |

**ACTIVITY 1.** OBJECTIVE: Describe the management of executing threads in Java.

FORMULATION: Given the following program:

```
public class T extends Thread {
       protected int n;
       public T(int n) {this.n = n;}

       public void delay(int ms) {
      // suspends execution for ms milliseconds
            try {  sleep(ms);
             } catch (InterruptedException ie){ie.printStackTrace();}
       }

       public void run() {
          for (int i=0; i<10; i++) {
                 System.out.println("Thread "+n +" iteration "+i);
                 delay((n+1)*1000);
              }
          System.out.println("End of thread "+n);
       }

       public static void main(String[] argv) {
          System.out.println("---  Begin of execution ---- ");
          for (int i=0; i<6; i++)
               new T(i).start();
          System.out.println("--- End of execution ---- ");
       }
 }
```

1) How many threads are created? Are they created using a “class with name” or an “anonymous class”?

6, but 7 if we count the main thread

2) Rewrite the previous code, but this time implementing *Runnable* (instead of extending *Thread*)

3) What will happen if we use `T(i).run()` in the *main* method instead of `T(i).start()`?

No concurrency

4) The message `"End of execution"` is not always the last message to be written on the screen. Why? How can we ensure that it is ALWAYS written at the end?

Because it is run after starting threads which print messages, but we are not waiting for them to finish before we print the end of execution message.

To make sure it is always written at the end, we could store the last thread in a variable `t` and call `t.join()` before the ending message.

**ACTIVITY 2 -** OBJECTIVE: Describe the management of executing threads in Java.

FORMULATION: Given the following Java code:

```
public class T2 extends Thread {
    private int level;
    public T2(int n){
       level = n;
    }
    public void createThread(int i) {
       T2 h = new T2(i);
       if (i>=1)
          h.start();
       System.out.println("Thread of level "+i+" created.");
    }
    public void run() {
       if (level>0)
           createThread(level-1);
       System.out.println("End of thread. Level:" + level);
    }
    public static void main(String[] argv) {
       for (int i=1; i<3; i++)
           new T2(2).start();
    }
}
```

1) How many threads are **created**? How many of them are **executed**?

6 are created (2 of level 2, 2 of level 1, and 2 of level 0).
Only 4 are executed (level 0 threads are not executed).

2) Is there any thread with "level" equal to 0? And with "level" 1? If so, indicate how many threads there are with these levels.

2 of each

3) Show a possible trace of this program, i.e. all messages shown in the screen.

Thread of level 0 created | Thread of level 0 created | End of thread. Level:1 | Thread of level 1 created
End of thread. Level:2 | Thread of level 1 created | End of thread. Level:1 | End of thread. Level:2

4) If we execute several times this program, do we always obtain the same messages? Are these messages always shown in the same order? Why?

No, because we don't know when the scheduler will change threads. First, every "thread created" message could be printed because context switch occurs just after printing the message, and then every "thread ended" message.

5) What will be the result of this program if there is `h.run()` in the "createThread" instead of `h.start()`? Please, show a trace of this new version of the program. If we execute it several times, would we always obtain the same result?

Now, only 2 threads are started (the ones created in the main loop), the rest run sequentially and not in separated threads. Therefore, context switches can still happen between those 2 threads, so the message trace can be different between executions.

**ACTIVITY 3.-** OBJECTIVE: Describe the management of executing threads in Java.

FORMULATION: Given the following Java code:

```
public class ExThread {
      public static void main(String[] args) {
          System.out.println(Thread.currentThread().getName());
          for (int i=0; i<10; i++){
             new Thread("MyThread "+i){
                 public void run() {
                    System.out.println("executed by"+
                        Thread.currentThread().getName());
                 }
             }.start();
          }
       }
  }
```

1) For thread creation, does it use a "class with name" or an "anonymous class"? Is `Thread` class extended or is `Runnable` interface implemented?

2) What does `Thread.currentThread.getName()` mean? What is it used for here?

3) Show a possible trace of the program and what will be shown on the screen.

4) Modify this program to assign the name of the thread with the `setName()` method of `Thread` class.

**ACTIVITY 4.-** OBJECTIVE: Describe the management of executing threads in Java.

FORMULATION: Given the following Java code:

```
public class ThreadName extends Thread {
   public void run() {
         for (int i = 0; i < 3; i++)
               printMsg();
   }
   public void printMsg() {
         System.out.println ("name=" +
          Thread.currentThread().getName());
   }
   public static void main(String[] args) {
         for ( int i = 0; i < 10; i++ ) {
               ThreadName tt= new ThreadName();
               tt.setName("MyThread" + i);
               if (i<5) tt.start()} }
   }}}
```

1) Briefly explain what this code does. Show a trace of this program.

2) How many threads are created? How many threads are executed?

3) Modify this program to use the anonymous creation of threads (i.e. without defining any instance of type `ThreadName tt)`.

**ACTIVITY 5.** OBJECTIVE: Discuss methods `isAlive` vs. `join`.

Given the following code:

```
public class CalculateResults extends Thread{
    private String result = "Not calculated";
    public void run(){
        result = calculate();}
    private  String calculate(){
        // Performs a long-time calculation
        try {Thread.sleep(10000);
        } catch(InterruptedException e){};
        System.out.println("Agent thread finishes its calculation");
        return "Calculation done";
    }
    public  String getResults(){
        return result; }
}


class Example_1 {
    public static void main(String[] args){
        CalculateResults agent =new CalculateResults();
        agent.start();
        // It does something during the calculation process
        System.out.println("Main in execution");

       // Employs the result
        System.out.println(agent.getResults());
    }
}
```

1) What does this code do? What is it shown on the screen?

2) Rewrite the *main*() method to make the main thread wait until the agent thread finishes before obtaining the result calculated by this agent thread. For implementing this wait you have to employ **busy-waiting** in the main thread (i.e. the thread will repeatedly check whether the condition is true or not).

3) Rewrite the *main*() method, but this time you should employ **suspending waiting** (i.e. suspending the thread that must wait).

**ACTIVITY 6.** OBJECTIVE: Understand the `Thread.interrupt` method
Given the following Java code:

```
public class SimpleThreads {
   // Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName =  Thread.currentThread().getName();
        System.out.format("%s:%s%n",threadName, message);
     }
    private static class MessageLoop  implements Runnable {
        public void run() {
            String importantInfo[] = {"One","Two","Three","Four"};
            try {
               for (int i = 0; i < importantInfo.length; i++) {
                  Thread.sleep(4000); // Pause for 4 seconds
                  threadMessage(importantInfo[i]);  // Print a message
               }
           }catch(InterruptedException e) {threadMessage("I wasn't done!");}
    } }

  public static void main(String args[]) throws InterruptedException {
     // Delay, in milliseconds before we interrupt MessageLoop thread
     long patience = 1000 * 60;  // (default one minute)
     threadMessage("Starting MessageLoop thread");
     long startTime = System.currentTimeMillis();
     Thread t = new Thread(new MessageLoop());
     t.start();
     threadMessage("Waiting for MessageLoop thread to finish");
     // loop until MessageLoop thread exits
     while (t.isAlive()) {
       threadMessage("Still waiting...");
       //Wait maximum of 1 second for MessageLoop thread to finish.
       t.join(1000);
       if (((System.currentTimeMillis()-startTime)>patience)&& t.isAlive()) {
           threadMessage("Tired of waiting!");
           t.interrupt();
           // Shouldn't be long now --- wait indefinitely
           t.join();
        } }
     threadMessage("Finally!");
  } }
```

1) What is it shown on the screen? What is the usage of `Thread.isAlive()`, `Thread.interrupt()` and `Thread.join()` methods here?

2) Try different values of "`patience`" to check the usage of the `interrupt` method.

**ACTIVITY 7.** OBJECTIVE: Use of Lambda notation in the creation of threads.

Analyse the following code and answer the questions.

```
1  public class RunnableLambdaEx {
2   public static void main(String[] args) {
3    System.out.println(Thread.currentThread().getName() + ": RunnableTest");
4
5    // --- Parte 1 ----------------------
6    Runnable task1 = new Runnable(){
7      public void run(){
8        System.out.println(Thread.currentThread().getName() + " is running");
9      }
10   };
11   Thread thread1 = new Thread(task1);
12   thread1.setName("hilo1"); thread1.start();
13   new Thread(task1,"hilo1b").start();
14
15   // --- Parte 2 ---------------
16   Thread thread2 = new Thread(new Runnable() {
17     public void run(){
18       System.out.println(Thread.currentThread().getName() + " is running");
19     }
20   });
21   thread2.setName("hilo2"); thread2.start();
22
23   new Thread(new Runnable() {
24     public void run(){
25       System.out.println(Thread.currentThread().getName() + " is running");
26     }
27   },"hilo2b").start();
28
29   // --- Parte 3 --------------------
30   Runnable task3 = () -> {
31       System.out.println(Thread.currentThread().getName() + " is running");
32   };
33   new Thread(task3,"hilo3b").start();
34
35   Thread thread3 = new Thread(() ->
36       System.out.println(Thread.currentThread().getName() + " is running"));
37   thread3.setName("hilo3"); thread3.start();
38
39   new Thread(() -> System.out.println(Thread.currentThread().getName()+ " is
40 running"),"hilo4").start();
41  }
42 }
```

1) Indicate, for each of the parts (Parte 1, Parte 2, Parte 3), how many threads will be created. Are any of the thread creation statements incorrect?.

2) Show in a possible programme trace what will be printed on the screen.

3) Establish the correspondence between the instructions of part 3 with lambda notation, which are equivalent in the way threads are created (except for object names) to those in parts 1 and 2.

- lines 30-32 equivalent to .....

- line 33 equivalent to .....

- lines 35-38 equivalent to .....

- lines 39-40 equivalent to .....

**ACTIVITY 8.** OBJECTIVE: Use of Lambda notation in the creation of threads.

Rewrite the code from activity 3 using the Lambda notation, so that the class ExThread provides the same functionality.

```
public class ExThread {
      public static void main(String[] args) {
          System.out.println(Thread.currentThread().getName());
          for (int i=0; i<10; i++){
              new Thread("MyThread "+i){
                  public void run() {
                     System.out.println("executed by"+
                          Thread.currentThread().getName());
                  }
              }.start();
          }   }   }
```