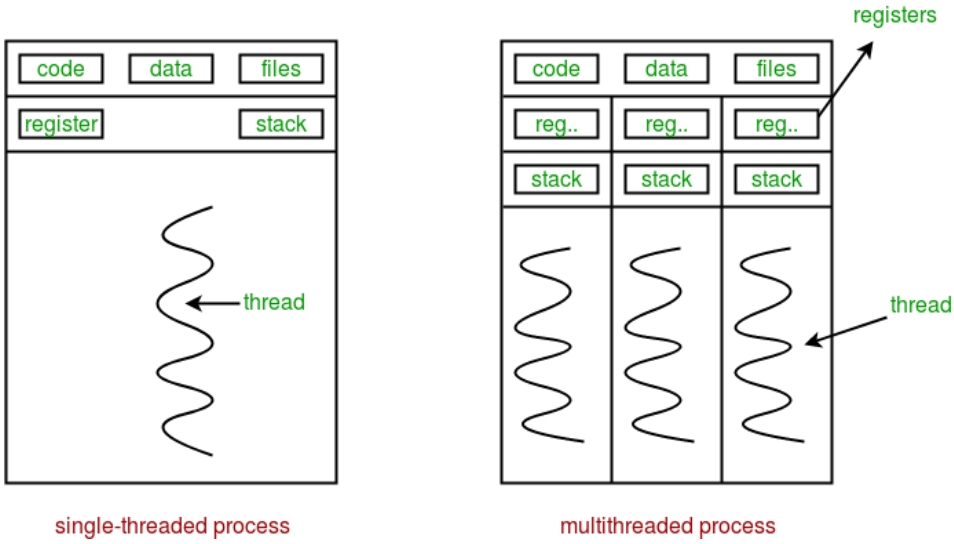
Exercises UD02



1. Training

1. Create two Java classes (threads) that extend the `Thread` class. One of the threads must display the word PING on the screen in an infinite loop and the other the word PONG. Inside the loop use the `sleep()` method so that we have time to see the words that are displayed when we execute it, you will have to add a `try-catch` block (to catch the `InterruptedException` exception). Then create the `main()` function that makes use of the previous threads. Are the PING and PONG texts displayed in order (ie PING PONG PING PONG ...)?
2. Transform the previous exercise but use the `Runnable` interface to declare the thread.
3. Make an application that executes 4 threads so that they are executed in an orderly manner one after the other, waiting for each one to finish the previous one to execute
4. Create a Java class that uses 5 threads to count the number of vowels in a given text. Each thread will take care of counting a different vowel, all threads updating the same common variable that represents the number of total vowels. To avoid race conditions, synchronized methods must be used.
5. Create a Java program that launches five threads, each will increment a counter variable (integer and shared by all) 5000 times and then exit. Check the final result of the variable. Is the correct result obtained?

   Now synchronize access to that variable. Launch threads first via `Thread` and then via the `Runnable` interface. Check the results by making a comparative table between the three sections.
6. Planning stretching exercises
   Create an application that, using the `Timer` and `TimerTask` classes, reminds the user to get up to stretch their legs every 30 minutes.
   Although it is outside the scope of this book, you can cause the warning to be displayed via an operating system notification using the `SystemTray` and `Traylcon` classes from the `java.awt` package. You can find multiple examples on the web.
7. Sleeping threads
   - Create a Java application that builds 5 threads from a single class that inherits from `Thread`.
   - Each thread has a name.
   - Each thread, in its `run` method, has an infinite loop. Inside the loop:
     - Write the text: "I'm the loop *name* and I'm working."
     - Stops execution for a random period of time between 1 and 10 seconds.
8. Implement a program that receives through its arguments a list of text files and counts the number of characters in each file. Modify the program so that a thread is created for each file to be counted. Shows how long it takes to count each file in the first sequential task and then using threads. To calculate the time it takes to execute a process we can use the `System.currentTimeMilis()` method as follows:

```java
long t_start, t_finish;
t_start = System.currentTimeMillis();
Process(); //work
t_finish = System.currentTimeMillis();
long t_total = t_finish - t_start;
System.out.println("The work lasted: " + t_total + " miliseconds");
```

9. Make a Java program that receives through its arguments a list of text files and counts the number of words in each file. A thread must be created for each file to be counted. Shows the number of words in each file and how long it takes to count the words.

10. Resolving concurrency error with `java.util.concurrent` classes

    The following code produces concurrency errors. Solve the problem using the classes from the `java.util.concurrent` package.

```java
package UD02;

import java.util.ArrayList;
import java.util.List;

public class PageManager extends Thread {

    private static List<String> list = new ArrayList<String>();

    @Override
    public void run() {
        while (true) {
            if (list.size() >= 10) {
                list.remove(0);
            } else if (list.size() < 10) {
                list.add("Text");
            }
            for (String s : list) {
                //going through the list
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            list.add("Texto");
        }
        for (int i = 0; i < 100; i++) {
            new PageManager().start();
        }
    }
}
```

11. Use of Semaphores

    Program a multithreaded system composed of two methods. The code of each of the methods can only be executed by two threads simultaneously, so four threads could be in concurrent execution, two in each method. Implement the solution using semaphores.

12. It is about simulating the game to guess a number. Several threads will be created, the threads are the players who have to guess the number. There will be a referee who will generate the number to guess, check the player's move and find out which player has the turn to play. The number has to be between 1 and 10, use the following formula to generate the number: `1 + (int)(10*Math.random());`

    3 classes are defined:

    - `Referee`: Contains the number to guess, the turn and shows the result. The following attributes are defined: the total number of players, the turn, the number to guess and whether the game is over or not. In the constructor, the number of participating players is received and the number to guess and the turn are initialized. It has several methods: one that returns the turn, another that indicates if the game is over or not and the third method that checks the player's move and finds out who it is next, this method will receive the player identifier and the number that has played; should be set to `synchronized`, so when one player is making the move, no other player can interfere. In this method, it will be indicated what the next turn is and if the game has ended because a player has guessed the number correctly.
    - `Player`: Extends `Thread`. Its constructor receives a player identifier and the arbiter, all threads share the arbiter. The player inside the `run` method will check if it is his turn, in which case he will generate a random number between 1 and 10 and create the move using the corresponding umpire method. This process will repeat until the game is over.
    - `Main`: This class initializes the referee indicating the number of players and launches the player threads, assigning an identifier to each thread and sending them the referee object they have to share.

    Example of output when executing the program:

```
NUMBER TO GUESS: 3
Player1 says: 9
     It's Player2 turn
Player2 says: 9
     It's Player3 turn
Player3 says: 10
     It's Player1 turn
Player1 says: 4
     It's Player2 turn
Player2 says: 7
     It's Player3 turn
Player3 says: 7
     It's Player1 turn
Player1 says: 6
     It's Player2 turn
Player2 says: 3
  Player2 wins!
```

2. Activities

1. What is a thread?
   a) A small process that runs in its own memory space.
   b) A small computing unit that runs independently.
   c) A small unit of computation that runs within the context of a process.
   d) A synchronized method that is executed in an exclusive way.

2. Which of the following is not a characteristic of threads?
   a) It can be executed independently of the system processes.
   b) It takes advantage of the processor cores generating a real parallelism.
   c) It cannot be executed independently, it always depends on a process.
   d) They share memory space.

3. What class allows you to create threads in Java?
   a) The `Runnable` class.
   b) The `Thread` class.
   e) The `Exception` class.
   d) The `Process` class.

4. Which method allows to start the execution of a thread in Java?
   a) `run`.
   b) `start`.
   e) `init`.
   d) `go`.

5. What interface allows to create execution threads?
   a) The `Runnable` interface.
   b) The `Thread` interface.
   e) The `Exception` interface.
   d) The `Process` interface.

6. What does Runnable allow that Thread does not?
   a) Create a thread.
   b) Implement the `run` method.
   c) Create multiple threads based on a single object.
   d) Sharing information between threads.

7. Which class in the `java.lang` package represents a scriptable task?
   a) `Thread`.
   b) `Timer`.
   e) `TimerTask`.
   d) `Runnable`.

8. Which description is correct for the `ExecutorService` interface?
   a) Subinterface of `Executor`, allows managing asynchronous tasks.
   b) Allows the scheduling of asynchronous task execution.
   e) Factory of `Executor` and `Callable` objects.
   d) Provides representations of units of time.

9. Which class allows limiting the number of threads accessing a critical section?
   a) `Thread`.
   b) `Synchronized`.
   c) `Semaphore`.
   d) `Phaser`.
10. Which class provides a synchronized data structure equivalent to an ArrayList?
    a) `ConcurrentHashMap`.
    b) `ConcurrentSkipListMap`.
    e) `CopyOnWriteArrayList`.
    d) `CopyOnWriteArraySet`.
11. Which method allows a thread to be stopped until another thread finishes its work?
    a) `run`.
    b) `join`.
    e) `sleep`.
    d) `yield`.
12. Given two code segments whose outputs are assigned to the same variables. What type of dependency would be produced?
    a) Flow dependency.
    b) Anti dependency.
    c) Output dependency.
    d) It does not generate any dependency.
13. What type of concurrency problem occurs when two threads have different values for the same variable?
    a) Sliding condition.
    b) Race condition.
    e) Memory inconsistency.
    d) Deadlock.
14. How can I recover the execution of a stopped thread with the `wait` method?
    a) With the `sleep` method.
    b) With the `start` method.
    c) With the `notifyAll` method.
    d) With the `yield` method.
15. When several threads access shared data, and the execution result depends on the specific order in which the shared data is accessed, it is said that:
    a) There is a critical section.
    b) There is a race condition.
    c) That can't happen, threads can't share data.
    d) None of the above.
16. Any solution to the critical section problem must satisfy the conditions of:
    a) Mutual exclusion, progress and limited waiting.
    (b) Mutual exclusion, limited waiting and holding.
    c) Aging and starvation.
    d) Mutual exclusion, isolation and limited waiting.

17. Indicate which of the following statements about monitors is FALSE:
    a) They allow to solve the problem of the critical section.
    b) They can be binary or counters.
    c) They can be used to guarantee the order of execution between processes.
    d) They are a synchronization mechanism.

18. Given the following piece of code on a particular Object to perform a critical section:

```
notify();
//Critical section
wait();
```

    Which of the following statements is true?
    a) The code shown does not ensure retention.
    b) The code shown does not ensure mutual exclusion.
    c) It is a valid solution to the critical section problem when the code runs on a single processor.
    d) The code shown allows to solve the critical section problem.

19. Indicate which of the following statements about Semaphores is FALSE:
    a) They allow to solve the problem of the critical section.
    b) They can be binary or counters.
    c) They cannot be used to guarantee the order of execution between processes.
    d) They are a synchronization mechanism.

20. Given the following code snippet, and assuming that the `Semaphore S` object has an initial value of 2:

```
S.release(S);
Function();
S.acquire(S);
```

    How many processes can execute the Function task at the same time?
    a) 0
    b) 2
    c) 3
    d) None of the above. It depends on the execution.

21. Given the following piece of code executed by a thread:

```
syncronyzhed(Object){
...
if (<<condition is true>>)
    wait();
    FUNCTION
}
```

    Which of the following statements is true?
    a) The thread will only execute FUNCTION when the condition is not met.
    b) The thread could execute FUNCTION even if it is fulfilled, due to the thread waiting for get monitor.
    c) The thread could execute the FUNCTION even if it is fulfilled, since it is a problem

inherent to the parallel use of threads.
d) The thread will never get to execute FUNCTION.

22. A race condition:
    a) It happens by allowing multiple processes to manipulate shared variables concurrently.
    b) It exists when several processes access the same data in memory and the result of the execution depends on the specific order in which the accesses are made.
    c) It does not need software mechanisms to be prevented, since whenever the code of concurrent processes with the same arguments is executed, they will be executed in the same way.
    d) a and b are correct.
23. Given the following code snippet, and assuming that the `Semaphore sem` object has an initial value of 2:

    ```
    1 sem.acquire();
    ```

    What will happen?
    a) The process executing the operation is blocked until another process executes a `sem.release();`
    b) The process will continue without blocking, and if there were previously blocked processes at because of the semaphore, one of them will be unlocked.
    c) After doing the operation, the process will continue without blocking.
    d) A semaphore can never have the value 2, if its initial value was 0 (zero) and it has been operated correctly.
24. If a synchronized block is used to achieve process synchronization:
    a) Condition variables should always be included, as the block only provides mutual exclusion.
    b) The `wait` and `notify` operations are used within the same object.
    c) The `wait` and `notify` operations are used on separate objects.
    d) The condition variable is always specified with an `if` condition.

3. Application Activities

1. Explain how sequential programming is different from concurrent programming.
2. List and explain the states and transitions a thread goes through.
3. Explain why the same code can lead to successful execution or concurrency errors.
4. Explain what function the `Runnable` interface has in multithreaded programming in Java.
5. Explain the benefits of creating a thread from the `Runnable` interface versus doing it by inheriting from `Thread` from an inheritance point of view.
6. Choose two classes that represent data structures from the `java.util.concurrent` package and explain how they differ from their counterparts in the `java.util` package.
7. Tell what the `sleep` method is for, how it can be called, and what exceptions its call can raise.
8. Explain what interrupts are in computing.
9. List and explain the most important mechanisms for sharing information between threads.
10. Explain what variables declared as `volatile` guarantee.
11. Indicates how the `join` method is used to synchronize threads.
12. Describe the usefulness of semaphores in concurrent programming.

4. Expansion activities

1. Java's `Thread` class has a `stop` method that is marked deprecated. Find out why.
2. Find out what a thread pool consists of. Learn how to create them in Java.
3. Go deeper into the knowledge of interruptions in operating systems. Find out what are the main reasons why they are caused.
4. Find out if semaphores exist as a synchronization tool in the Python language. Even if you don't know how to program in this language, check if the lock reservation and release methods are similar to Java.
5. Deepens the knowledge of computing in video cards and CUDA architecture. Find out in which programming languages you can work with this technology.
6. In computing, there is a classic problem known as "The Philosopher's Dinner." Find out what it is and study some of the solutions available on the internet.
7. In this case study, a multitasking solution to the classic philosopher's dinner problem is going to be developed. At a round table there are N philosophers sitting. In total he has N chopsticks to eat rice, each chopstick being shared by two philosophers, one on the left and one on the right. Like good philosophers, they devote themselves to thinking, although from time to time they get hungry and want to eat. In order to eat, a philosopher needs to use the two chopsticks next to him.

   To implement this problem, you must create a main program that creates N threads executing the same code. Each thread represents a philosopher. Once created, an infinite wait loop is performed. Each of the threads will have to perform the following steps:

   1. Print a message on the screen “Philosopher i thinking”, being i the identifier of the philosopher.
   2. Think for a certain random amount of time.
   3. Print a message on the screen “Philosopher i want to eat”.
   4. Try to pick up the chopsticks you need to eat. Philosopher 0 will need chopsticks 0 and 1, Philosopher 1 chopsticks 1 and 2, and so on.
   5. When you have control of the chopsticks, it will print a message on the screen “Philosopher i eating”.
   6. The philosopher will be eating for a random amount of time.
   7. Once you have finished eating, you will put your chopsticks back.
   8. Go back to step 1.

   However, deadlocks can occur if, for example, all the philosophers want to eat at the same time. If everyone manages to take the toothpick to their left, none can take the one to their right. For this, several solutions are proposed:

   - Allow a maximum of N−1 philosophers sitting at the table.
   - Allow each philosopher to pick up their chopsticks only if both chopsticks are free.
   - Asymmetric solution: an odd philosopher first takes the stick on the left and then the one on the right. An even philosopher takes them in the reverse order.

Implement a solution to the philosophers problem, a solution that does not present a deadlock problem. For simplicity, it is recommended to use the proposed method of asymmetric solution.

5. Suggested exercises

1. Write a class called `Command` that creates two threads and forces the writing of the second to always be before the screen writing of the first.

   Execution example:

   ```
   Hi, I'm thread number 2.
   Hi, I'm thread number 1.
   ```

2. Write a class called `Relays` that simulates a relay race as follows:

   - Create 4 threads, which will wait to receive a signal to start running. Once the threads have been created, the race will be indicated to start, so one of the threads must start running.
   - When a thread finishes running, it puts a message on the screen and waits a couple of seconds, passing the token to another of the threads so that it starts running, and ending its execution (its own).
   - When the last thread finishes running, the parent will display a message indicating that all children have finished.

   Execution example:

   ```
   All threads created.
   I give the output!
   I am thread 1, running . . .
   I finished. I pass the baton to son 2
   I am thread 2, running . . .
   I finished. I pass the baton to son 3
   I am thread 3, running . . .
   I finished. I pass the baton to son 4
   I am thread 4, running . . .
   I finished!
   All threads ended.
   ```

3. Write a class called `SuperMarket` that implements the operation of `N` supermarket checkouts. The `M` customers of the supermarket will spend a random amount of time shopping and will subsequently randomly select which checkout to position themselves in order to be in their corresponding queue. When it is their turn, they will be attended to by proceeding to the corresponding payment and entering the variable Results of the supermarket. As many threads as there are clients must be created, and the `M` and `N` parameters must be passed as arguments to the program. To simplify implementation, each customer's payment value can be randomized at the time of their payment.

4. Write a class called `Parking` that receives the number of parking spaces and the number of cars in the system. You must create as many threads as there are cars. The car park will have a single entrance and a single exit. At the vehicle entrance there will be a control device that allows or prevents their access to the car park, depending on its current status (available parking spaces). The waiting times of the vehicles inside the car park are random. When a vehicle leaves the car park, it notifies the control device of the space number assigned to it and the space it was occupying is released, thus making these available again. A vehicle that has left the parking lot will wait a random time to re-

enter it again. So, the vehicles will be entering and leaving the car park indefinitely. It is important that the program is designed in such a way as to ensure that, sooner or later, a vehicle that remains waiting at the entrance of the car park will enter it (starvation does not occur).

Execution example:

```
IN: Car 1 parks at 0.
Free seats: 5
Parking: [1] [2] [3] [0] [0] [0]
ENTRANCE: Car 2 parks in 1.
Free seats: 4
Parking: [1] [2] [3] [0] [0] [0]
ENTRANCE: Car 3 parks at 2.
Free places: 3
Parking: [1] [2] [3] [0] [0] [0]
ENTRANCE: Car 4 parks at 3.
Free places: 2
Parking: [1] [2] [3] [4] [0] [0]
ENTRANCE: Car 5 parks at 4.
Free places: 1
Parking: [1] [2] [3] [4] [5] [0]
DEPARTURE: Car 2 leaving.
Free places: 2
```

6. Information sources

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos