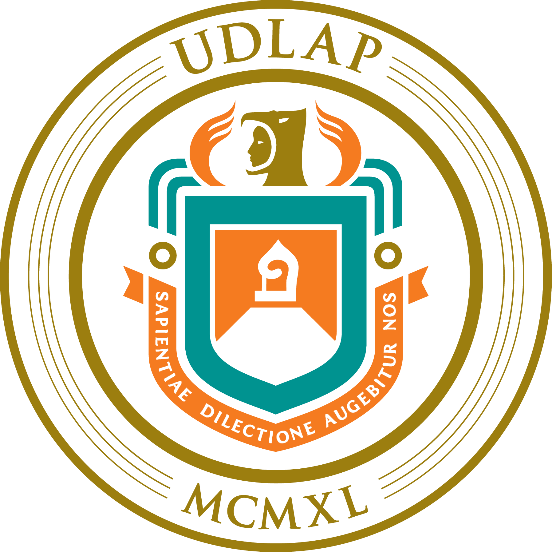
**Universidad de las Américas Puebla**



**Sistemas Distribuidos**

Práctica 4

Dining-Philosophers

**Integrantes del equipo:**

Pablo de la Rosa Michicol ID: 150245

Alina Daniela Fernández Jiménez ID: 150662

Primavera 2017

**Fecha:** 29/03/17

## **Índice:**

Objetivo…………………………………………………………….....……………...……...…....…2

Desarrollo………………………...………………………………..............................................…3

Actividad 1.……………………………...………………...............................................………….3

Actividad 2.……………………...………………………..............................................……….....5

Actividad 3…………….…………….……………………...............................................………...8

Actividad 4.……………………...……………………………...............................................…….8

Conclusión……………….………………..………………….…………………………….......…...9

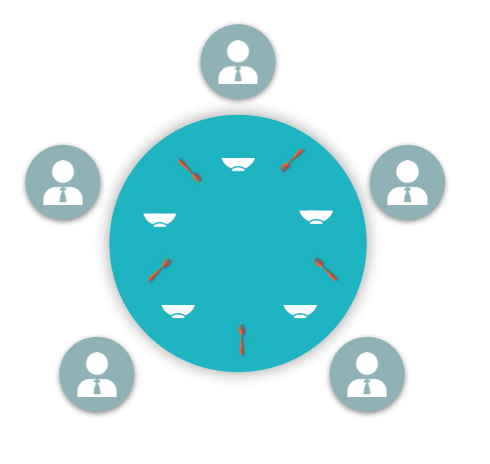
# **Práctica 3**

## **Objetivo:**

Esta práctica consiste en analizar el programa de los filósofos comensales con el motivo de comprender mejor el problema de sincronización de procesos en un sistema distribuido.

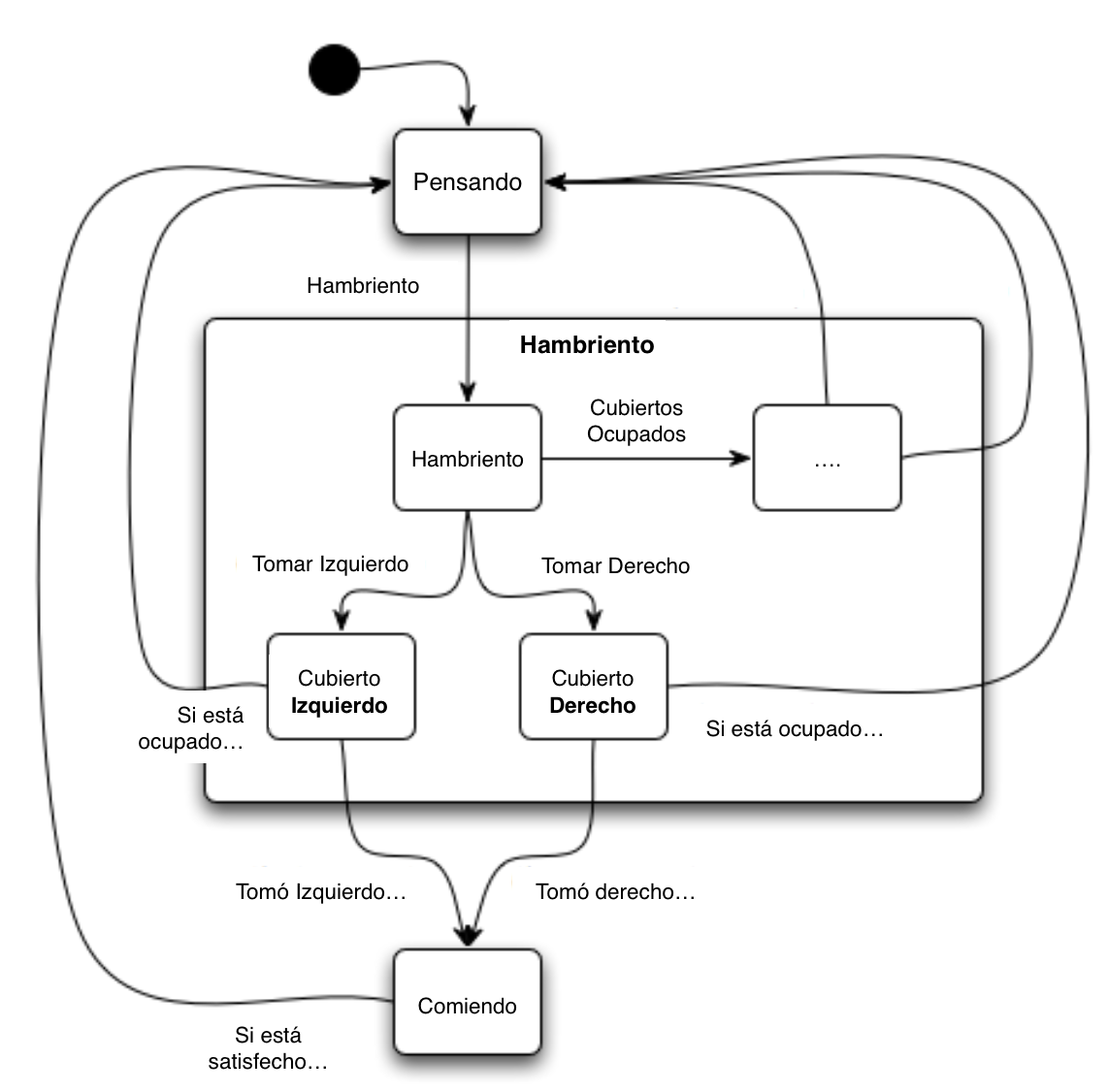
El problema consiste en lo siguiente:

Cinco filósofos se sientan silenciosamente alrededor de una mesa redonda, con platos llenos de spaghetti. Junto a cada filósofo se coloca un tenedor. Cada filósofo debe alternarse entre pensar y comer. Los filósofos puede comer únicamente cuando tienen ambos tenedores, derecho e izquierdo.



Restricciones del problema:

* Cada tenedor puede ser sujetado únicamente por un filósofo a la vez.
* Cuando un filósofo termina de comer pone ambos tenedores sobre la mesa.
  + Estos tenedores ahora están disponibles para los demás.
* Un filósofo puede tomar el tenedor de la derecha o la izquierda.
  + No puede comenzar a comer hasta tener ambos.



## **Desarrollo:**

### Actividad 1: Análisis Dining Philosophers

Esta actividad consistió en comprender el funcionamiento del programa y completar el código, para esto se llevaron a cabo los siguientes pasos:

* Se descargó y abrió el programa de DiningPhilosophers.
* Se analizaron las clases CenaFilosofo.java y Filosofo.java.
* Se analizaron los métodos y la documentación.
* Se completó el código faltante.
* Se observó el comportamiento de la aplicación cuando ocurre un deadlock.
* Creación del lock:

Lock nuevoLock = new ReentrantLock();

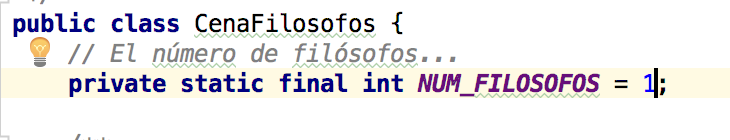
* Bloqueo de recurso:

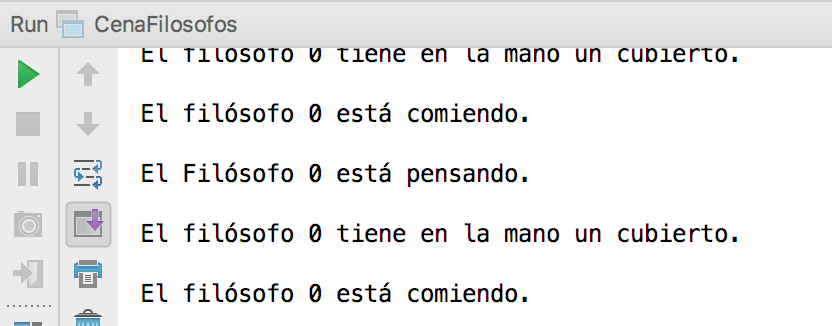
nuevoLock.lock();

* Desbloqueo de recurso:

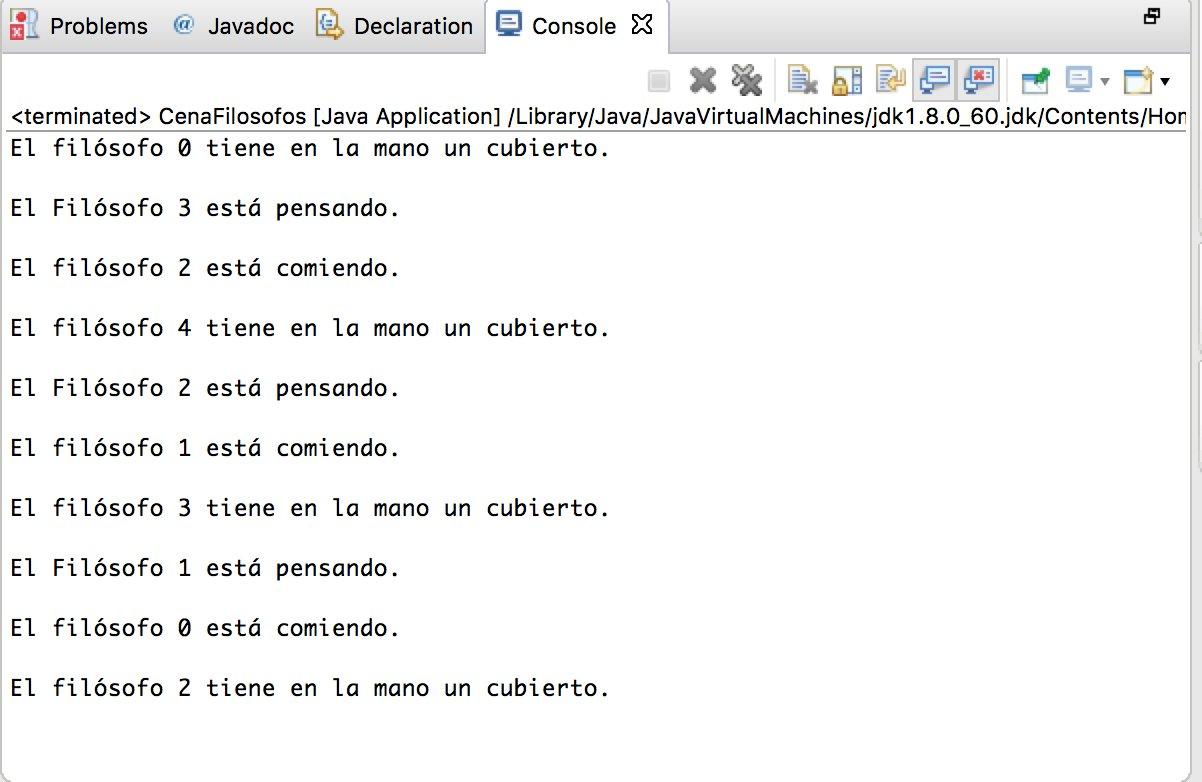
nuevoLock.unlock();

* Ejecución de la aplicación con 1 filósofo:

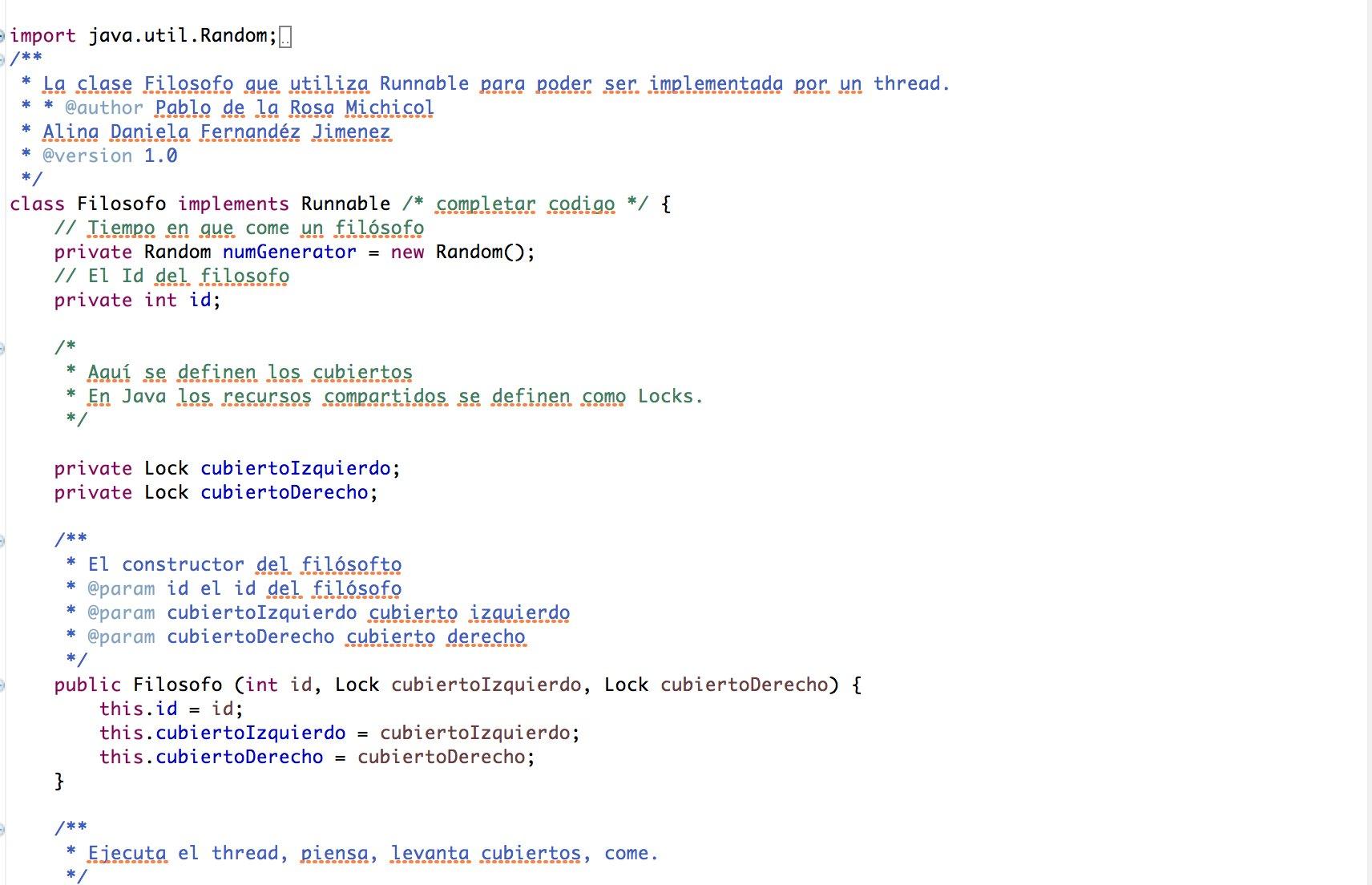




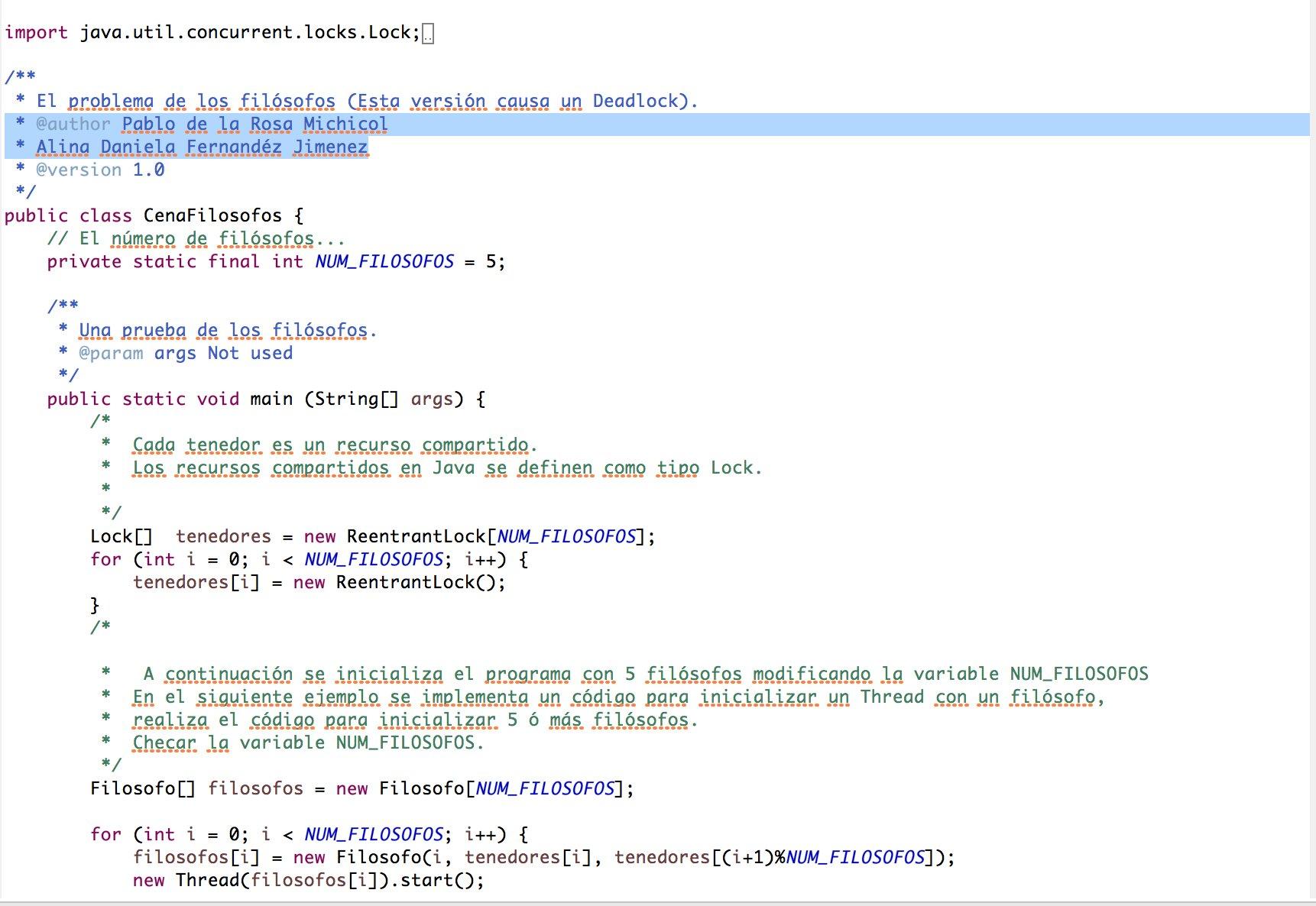
* Ejecución de la aplicación con 2 o más filósofos:



Clase Filosofo.java:



Clase CenaFilosofos.java:

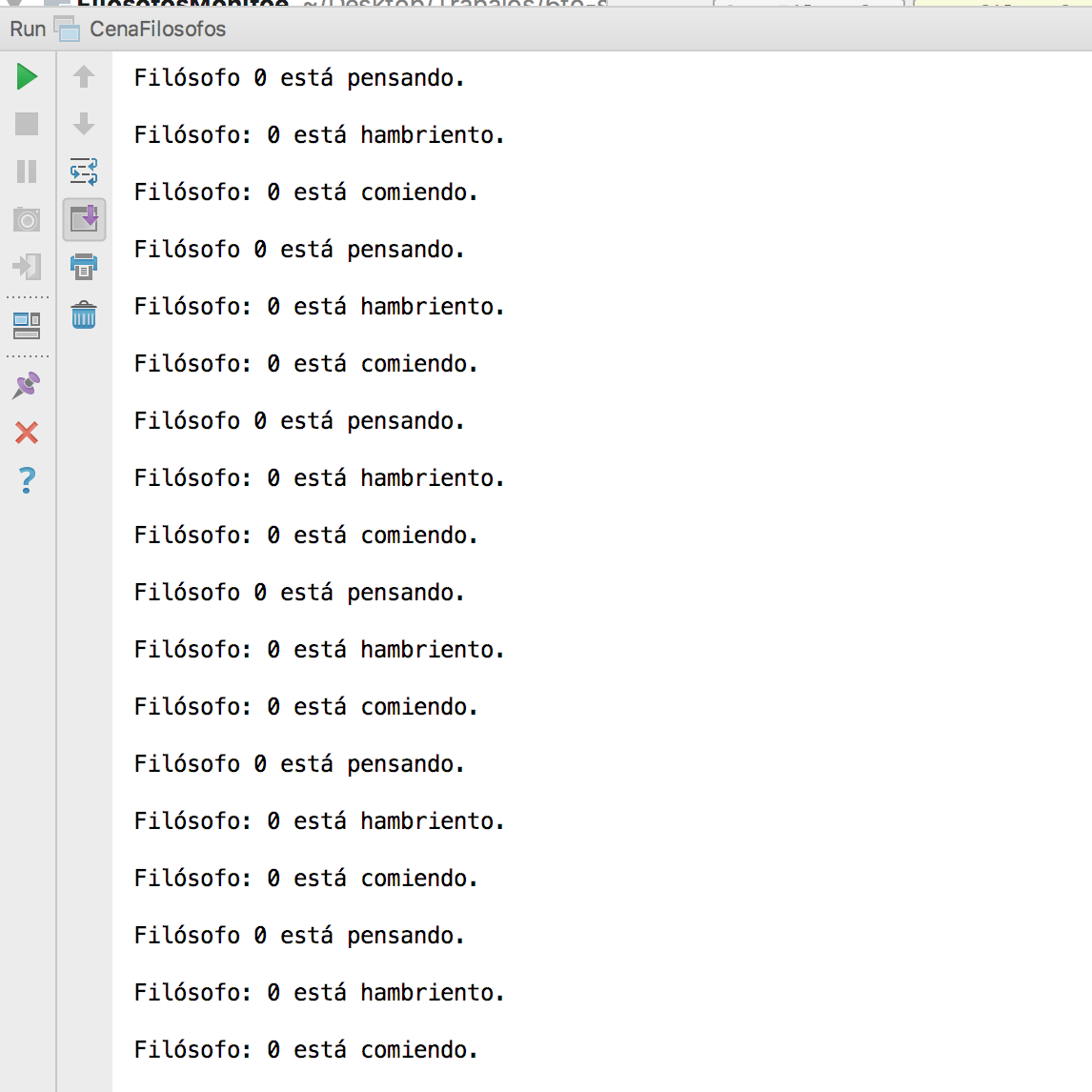


Para representar los tenedores se emplearon locks, los cuales se encargan de bloquear un recurso compartido a múltiples hilos de ejecución y así aseguran el acceso a un sólo hilo a la vez, en caso de que esté ocupado por un hilo se hace una cola de espera para que una vez que esté desocupado se haga un unlock para liberarlo y dejar que otro hilo lo pueda utilizar.

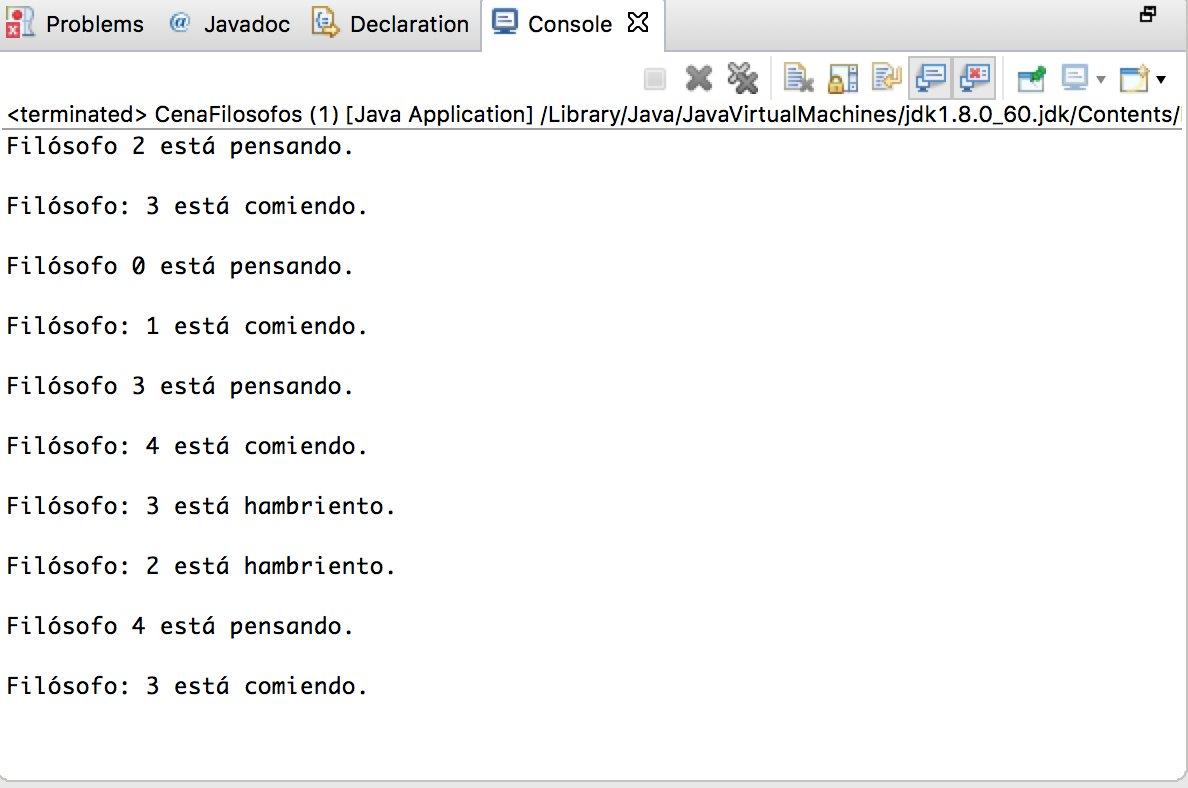
### Actividad 2: Desarrollo Dining Philosophers Monitor

Para esta actividad se introdujo el concepto de un monitor para resolver el problema de los filósofos. Este se encarga de evitar que haya deadlocks, sin embargo aún pueden morir de hambre (Starvation).

* Se abrió el proyecto DiningPhilosophersMonitor.
* Se analizaron las clases CenaFilosofo.java, Monitor.java y Filosofo.java.
* Se analizaron los métodos y la documentación.
* Se completó el código faltante.
* Se ejecutó la aplicación con 1 filósofo.

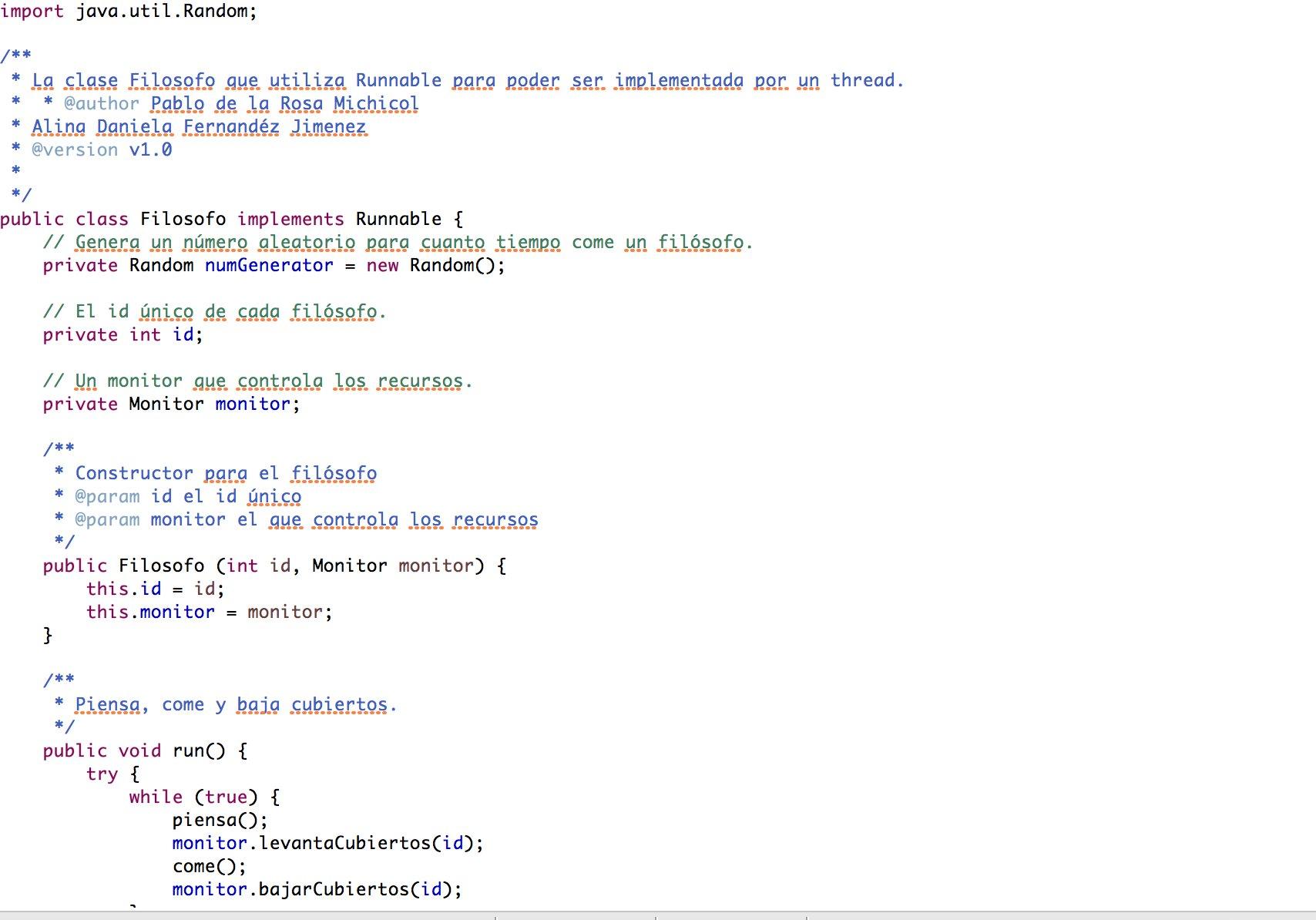


* Se ejecutó la aplicación con 2 o más filósofos.

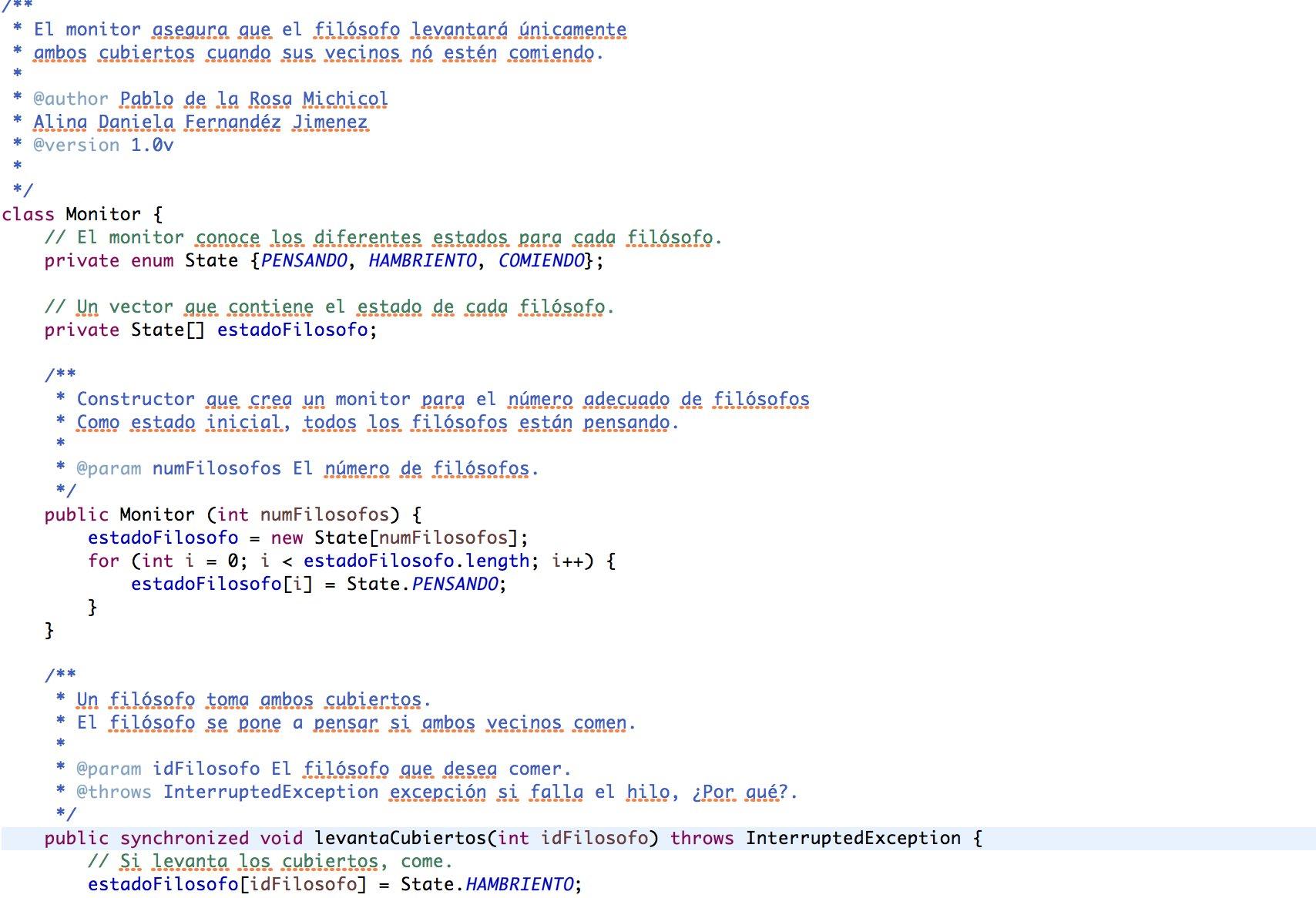


* Nota para la ejecución con 3, 5 ó 7 tenedores: En el caso del uso de un monitor no hay recursos que se pueden cuantificar como lo es en el caso de deadlocks, es decir que el monitor solo verificará las acciones cuando se levanta o bajan los cubiertos más no se puede elegir la cantidad de estos, esto quiere decir que el modelo de usar el monitor es más escalable que usar deadlocks.

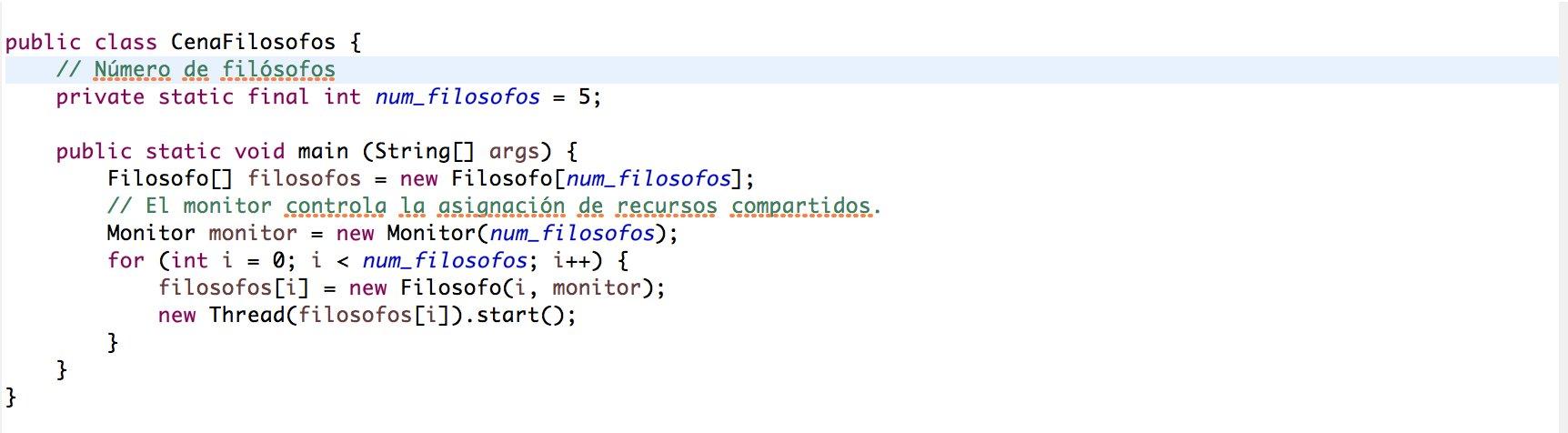
Clase Filosofo.java:



Clase Monitor.java:



Class CenaFilosofos.java:



Un monitor es una estrategia de sincronización que garantiza la exclusión mutua, cosa que un Lock no hace. Los monitores proporcionan un mecanismo para que los hilos esperen por un evento y esto se logra en Java mediante el uso de "Wait()". También notifica a los hilos que se ha desencadenado un evento con "Notify()".

### Actividad 3: Limpieza

En esta actividad se revisó y limpió el código identificando los métodos implementados.

* Se identificaron los métodos implementados para ambos proyectos.
* Se eliminaron los bloques de código que decían "Completar código".
* Se modificó la documentación con nuestros datos personales.

### Actividad 4: Preguntas

* ¿Cuál es la diferencia entre Wait() y Sleep()?

Wait() no es un método para esperar tiempo, es parte de una primitiva para sincronización.

Sleep() es un método para esperar tiempo, no libera ningún monitor.

* ¿Para qué se utiliza Lock?

Lock se utiliza para desbloquear los hilos en orden de llegada, a comparación de synchronized el cual no garantiza el orden.

* ¿Cuál es la diferencia entre Runnable() y Thread()?

Runnable es una interface que sólo implementa el método run(), mientras que Thread es una clase, la cual puede tener subclases para hacer uso de sus métodos.

* ¿Por qué es importante el uso de Synchronized?

Tiene una gran importancia ya que al declarar el uso de Synchronized nos estamos asegurando que si varios hilos se ejecutan en un método, solamente uno se ejecutará a la vez.

* ¿Cuál es la diferencia entre un Semáforo y un Monitor?

Un semáforo informa a los procesos que fueron bloqueados, el administrador de procesos elige quien pasa a ejecución, cuando se usa un monitor está asegurando que solo el productor o el consumidor estarán ejecutándose

* ¿Por qué los filósofos aún podrían morir de hambre (Starvation) en el ejercicio con el monitor?

Sucede la misma situación que usando deadlocks, a diferencia que con deadlocks tenemos recursos contables como son los tenedores, en el caso del monitor no tenemos recursos pero si un método que verifica cuando se suben y bajan los tenedores.

## **Conclusión:**

Gracias a esta práctica analizamos un problema básico de sincronización de procesos como lo es el de los filósofos comensales, de esta manera pudimos comprender mejor por qué ocurre este problema y cómo están elaboradas las soluciones propuestas, es importante saber cómo afrontar este tipo de problemas para encontrar la solución adecuada según cada caso ya que es muy común el problema de sincronización de procesos.

Contribuciones:

* Alina: Comprensión del problema afrontado, colaboración en encontrar solución y añadir código necesario a la primera parte de la práctica, limpieza de código y documentación.
* Pablo: Investigación sobre el problema de los filósofos, afrontar el problema con locks y monitores, realización de pruebas y ejecución, limpieza de código y resolución de preguntas.