IES Camas Antonio Brisquet

Familia Profesional: INFORMÁTICA Y COMUNICACIONES

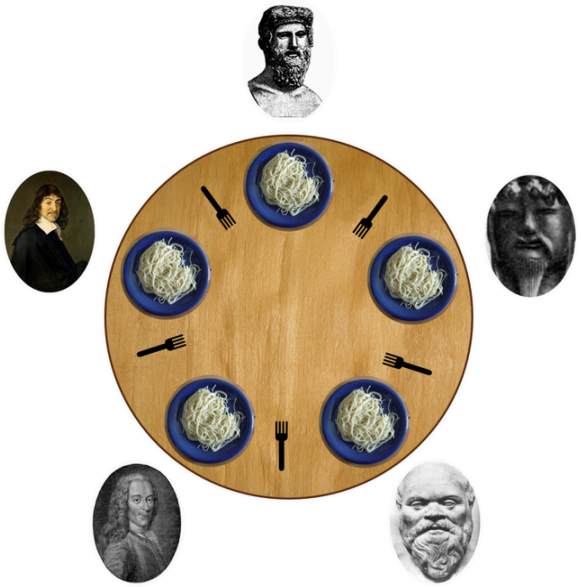
Ciclo Formativo:

DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Módulo:

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

Programación multihilo en Java: La cena de los filósofos



Autor: *Moreno Sánchez, Juan Manuel*

Docente: *Domínguez Mayo, Juan Pablo*

Curso académico: *2025 - 2026*

Índice de contenidos

1. Introducción

El objetivo de esta práctica es implementar una solución al problema clásico de la "Cena de los Filósofos", propuesto originalmente por Edsger Dijkstra. Este problema modela un escenario de concurrencia donde cinco procesos (filósofos) compiten por recursos limitados (tenedores) para realizar su tarea (comer), alternando con periodos de inactividad (pensar).

La implementación busca demostrar el manejo de hilos en Java, garantizando la seguridad en el acceso a recursos compartidos y evitando problemas críticos como el **interbloqueo (deadlock)**, la **inanición (starvation)** y las condiciones de carrera.

2. Diseño de la Solución

La solución se ha diseñado bajo el paradigma de Orientación a Objetos, separando claramente la lógica de negocio, la concurrencia y la ejecución. El proyecto se estructura en los siguientes paquetes:

- **`es.iescamas.multihilo.app`**: Contiene el punto de entrada (`main`). Se instancia el monitor y se lanzan los hilos.
- **`es.iescamas.multihilo.monitor`**: Contiene la lógica de sincronización. Se ha optado por el **Patrón Monitor** (`MonitorFilosofos`), ya que permite encapsular el estado compartido (array de estados de los filósofos) y proteger las secciones críticas mediante métodos `synchronized`.
- **`es.iescamas.multihilo.monitor.hilo`**: Define la tarea del filósofo (`Runnable`), separando la lógica del hilo de la gestión de recursos.

Se ha implementado una solución adicional basada en el algoritmo del "Portero" (Footman), utilizando la clase `MonitorFilosofoConPortero` que decora al monitor original para limitar el acceso a la mesa.

3. Implementación Concurrente

3.1. Uso del Monitor (`MonitorFilosofos`)

La clase `MonitorFilosofos` gestiona un array de enteros que representa el estado de cada filósofo: `PENSANDO (0)`, `HAMBRIENTO (1)` y `COMIENDO (2)`. Para garantizar la exclusión mutua y la coordinación, se utilizan los mecanismos nativos de Java:

- **`synchronized`**: Los métodos `tomarTenedores` y `dejarTenedores` son sincronizados para evitar condiciones de carrera al leer/escribir el array de estados.
- **`wait()`**: Si un filósofo quiere comer pero sus vecinos están comiendo, el hilo se suspende y libera el bloqueo del monitor, pasando al estado `WAITING` para no consumir CPU inútilmente.
- **`notifyAll()`**: Cuando un filósofo termina de comer, notifica a todos los hilos esperando para que verifiquen si sus condiciones para comer se cumplen ahora.

3.2. Prevención del Interbloqueo (Deadlock)

En la solución básica con Monitor, el deadlock se evita imponiendo la restricción lógica de que un filósofo sólo pasa a estado `COMIENDO` si ambos vecinos no están comiendo. Esto rompe la condición de "espera circular" típica donde cada uno tiene un tenedor y espera el siguiente.

3.3. Variante del Portero (MonitorFilosofoConPortero)

Se ha implementado una variante que utiliza un semáforo (`java.util.concurrent.Semaphore`) inicializado con `N-1` permisos (4 permisos para 5 filósofos).

- Antes de intentar coger tenedores, el filósofo debe adquirir un permiso del portero (`acquire()`).
- Al limitar el aforo a 4 comensales, se garantiza matemáticamente que al menos un filósofo podrá comer, eliminando la posibilidad de interbloqueo.

4. Pruebas y Resultados

Se han realizado pruebas de ejecución observando la salida por consola mediante la clase `Utilidad`, verificando visualmente que nunca aparecen dos vecinos comiendo simultáneamente (representado como

4.1. Escenario de Ejecución

Las pruebas se realizaron simulando una cena con 5 filósofos (`N=5`). Se utilizó la clase `Utilidad` para monitorizar visualmente la ejecución, confirmando que en ningún momento aparecen dos tenedores (`[ 🍴 ]`) asignados a filósofos vecinos simultáneamente.

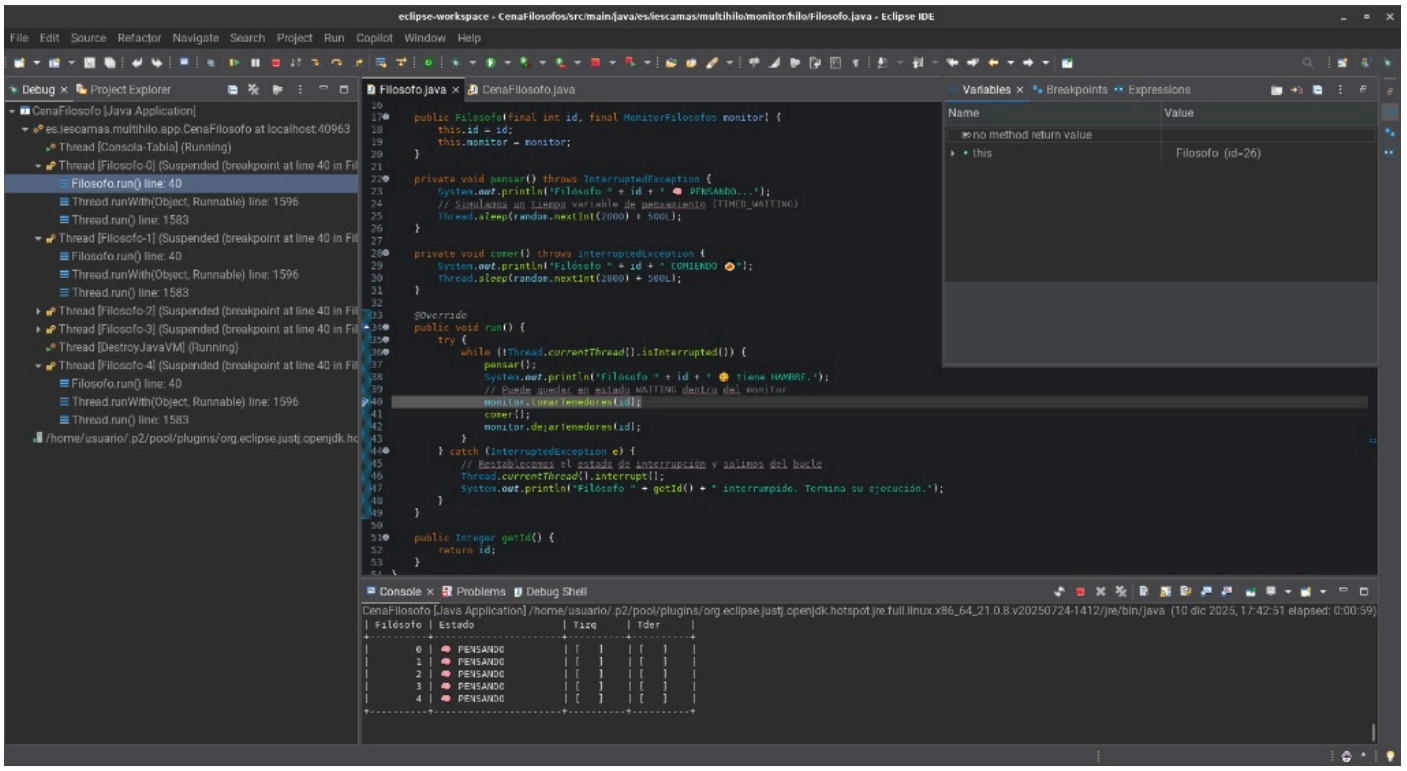
4.2. Pruebas Unitarias (JUnit)

Se han desarrollado pruebas unitarias en `MonitorFilosofosTest` para verificar la robustez del monitor:

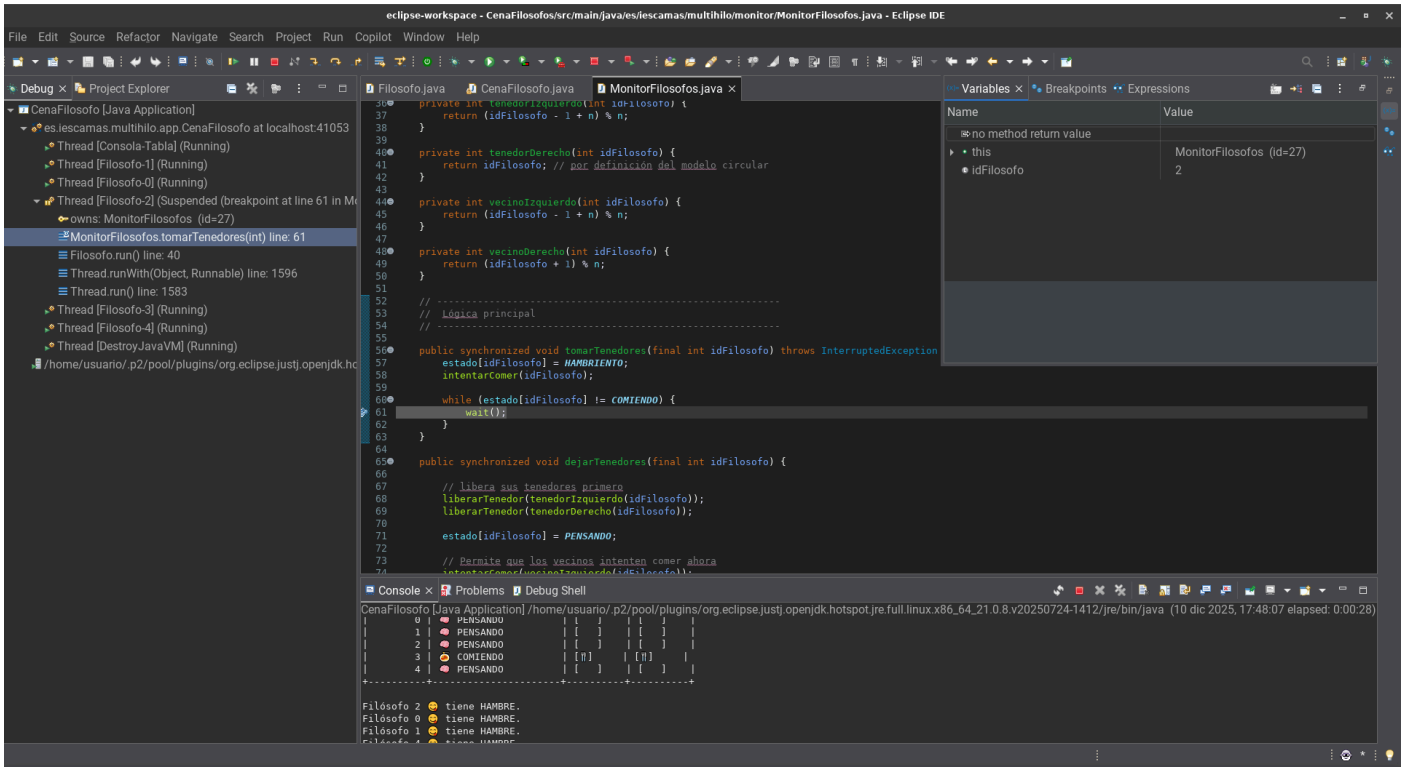
- **Test de Inicialización:** Verifica que todos comienzan pensando.
- **Test de Exclusión Mutua:** Se fuerza a dos hilos a competir por los mismos recursos, asegurando mediante aserciones (`assertFalse`) que el monitor impide el acceso simultáneo.
- **Test de Circularidad:** Verifica que la lógica del último filósofo conecta correctamente con el primero (filósofo 4 con filósofo 0).

4.3. Análisis de Depuración

Mediante el depurador de Eclipse, se ha comprobado el ciclo de vida de los hilos.



En la *Figura 1* se observan los 5 hilos (`Filosofo-0` a `Filosofo-4`) creados correctamente. El hilo `Filosofo-0` se encuentra detenido justo antes de solicitar acceso al monitor en la línea 40 (`monitor.tomarTenedores(id)`), demostrando que los hilos alcanzan la sección crítica.



En la *Figura 2* (ver abajo) se demuestra la gestión eficiente de la espera. Un hilo se encuentra en estado `WAITING` (suspendido en `Object.wait()`) debido a que la condición lógica para comer no se cumplió (sus vecinos tenían los tenedores).

5. Conclusiones

La práctica ha permitido comprender la complejidad de gestionar recursos compartidos. El uso de `wait()` y `notifyAll()` dentro de bloques `synchronized` es potente pero propenso a errores si no se controla bien el flujo. La implementación de variantes como el Portero con `Semaphore` demuestra cómo abstracciones de más alto nivel pueden simplificar la prevención de bloqueos.