

**Ingeniería en Sistemas**

**Computacionales**

**Raúl Tostado Blázquez I.D. 145901**

**Materia: Sistemas Distribuidos**

**Nombre del Trabajo: Práctica Dining-Philosophers**

**Profesora: Ofelia D. Cervantes Villagómez**

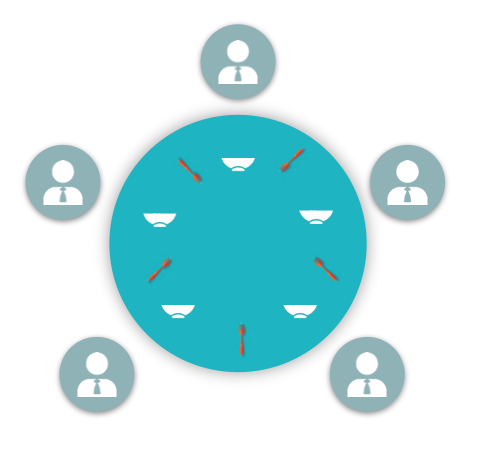
**San Andrés Cholula, Puebla a**

**25 de marzo de 2015**

Reporte práctica Dining-Philosophers

**Introducción a la práctica**

**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.

[](https://camo.githubusercontent.com/e950719a86eff9d12c3302600aec5f445cebc0c6/687474703a2f2f7777772e696e6e6f766134642e6d782f77702d636f6e74656e742f75706c6f6164732f323031352f30332f7068696c6f736f70686572732e706e67)

**Notas:**

* 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.

**Antes de comenzar:**

El primer paso fue dar una lectura detallada a la práctica para saber cuál es el contenido de la misma, posteriormente se descargaron como Zip los programas con los que se trabajó.

# Actividad 1 - Análisis Dining Philosophers

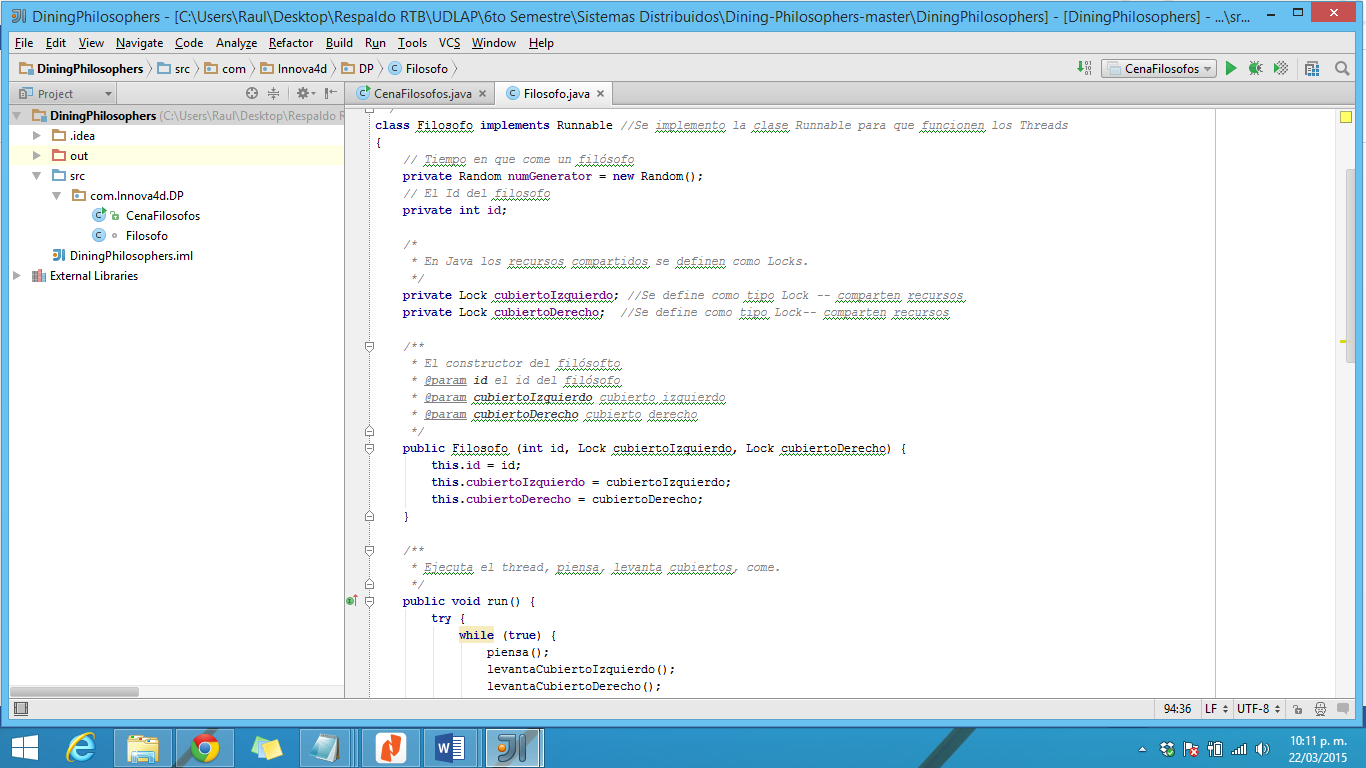
Esta actividad consiste en comprender el funcionamiento del problema de los filósofos. Para esto deberás **completar** el código en la aplicación Java **DiningPhilosophers**, instrucciones:

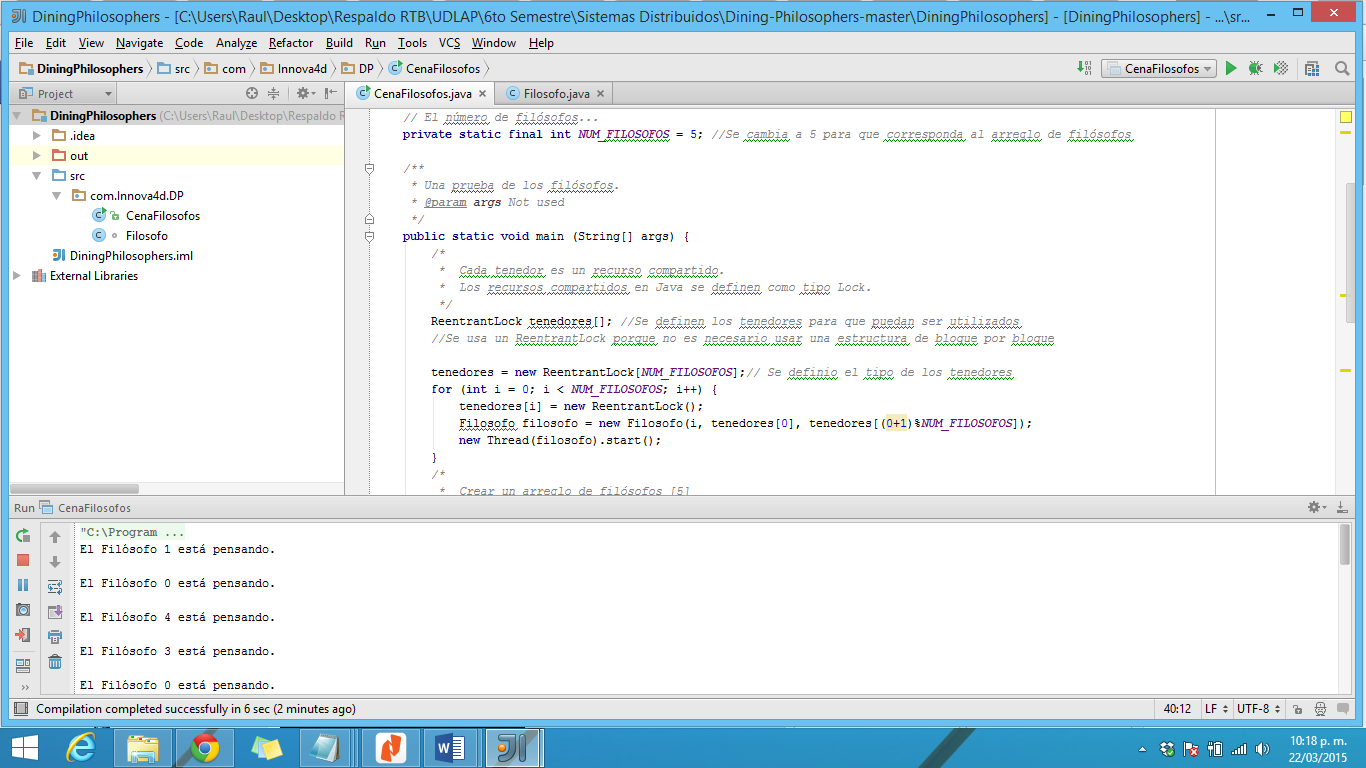
* Abrir el proyecto **DiningPhilosophers**.
* Analizar las clases **CenaFilosofo.java** y **Filosofo.java**.
* Analizar **atentamente** los métodos y la documentación.
* Completar el código faltante.

**Nota importante:** Éste primer ejercicio ilustra el comportamiento de una aplicación que causa un **deadlock**.

**Analiza** el comportamiento si:

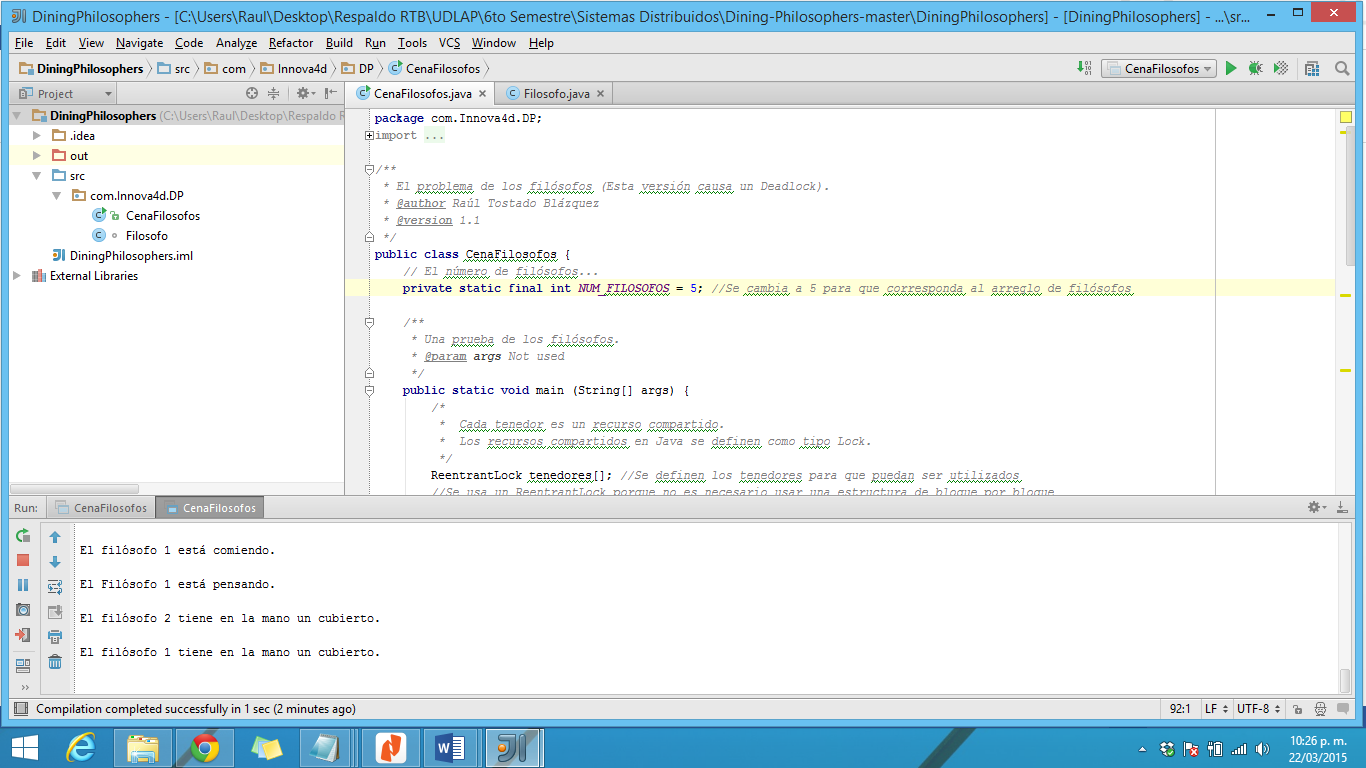
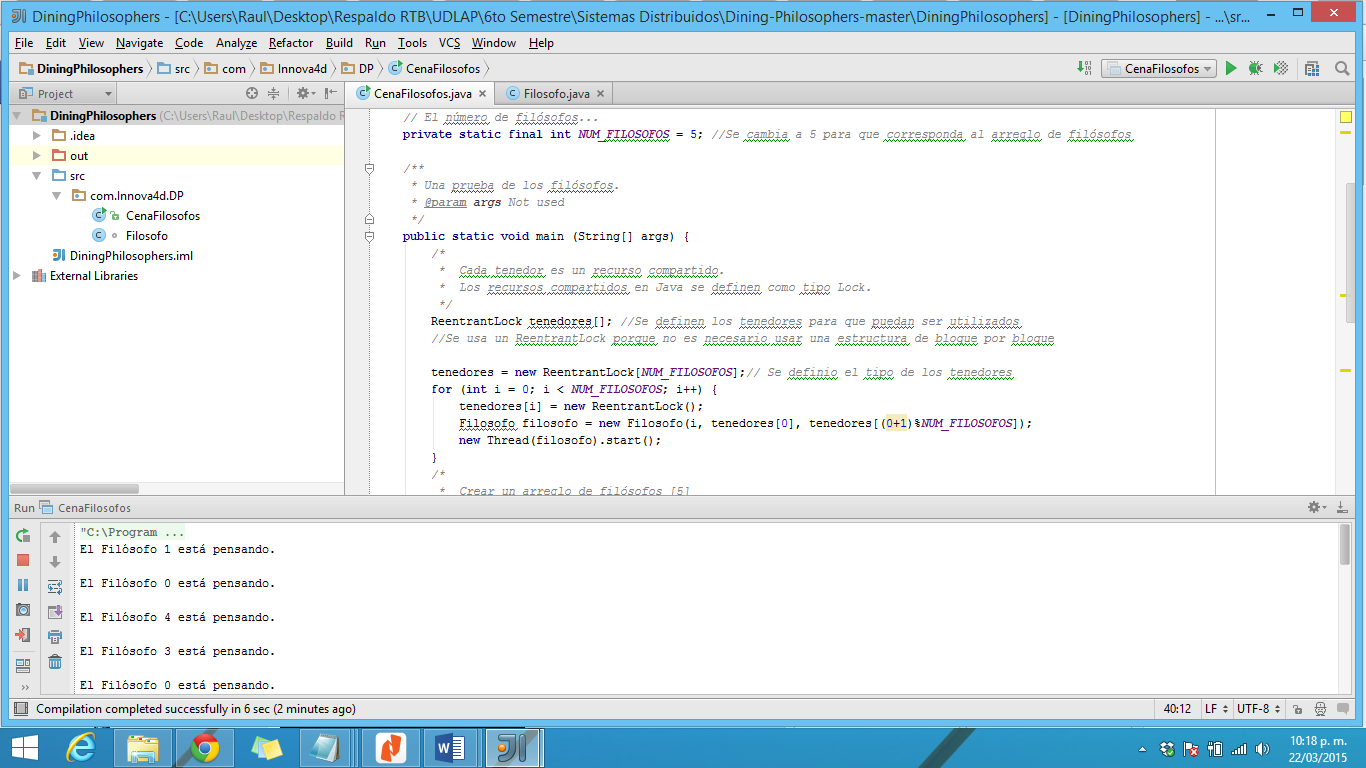
* Ejecutas la aplicación con 1, 2, 3, 4, y 5 filósofos.
* Es **importante** que anotes tus observaciones de ésta actividad en tu **reporte**.





Lo que se hizo fue hacer una lectura cuidadosa de cada una de las clases presentadas en las imágenes anteriores para saber cómo funciona, la parte de completar el código fue relativamente fácil, ya que si se analiza de manera clara todos los métodos podemos inferir que es lo que le hace falta al programa.

Independientemente del número de filósofos que se encuentren en la mesa el resultado siempre es el mismo un DeadLock, debido a que al quedarse con un cubierto en la mano, imposibilitan a los otros filósofos de que puedan comer y finalmente al “no compartir”, terminarán por morir de hambre los filósofos en la mesa.



# Actividad 2 - Análisis Dining Philosophers Monitor

Esta actividad consiste en comprender el funcionamiento de un **monitor** para resolver **el problema de los filósofos**. Para esto deberás **completar** el código en la aplicación Java**DiningPhilosophersMonitor**, instrucciones:

* Abrir el proyecto **DiningPhilosophersMonitor**.
* Analizar las clases **CenaFilosofo.java**, **Monitor.java** y **Filosofo.java**.
* Analizar **atentamente** los métodos y la documentación.
* Completar el código faltante.

**Nota importante:** Éste segundo ejercicio ilustra el comportamiento de una aplicación que utiliza un**monitor** para evitar un **deadlock**, sin embargo los filósofos aún podrían morir de hambre**(Starvation)**.

**Analiza** el comportamiento si:

* Ejecutas la aplicación con 1, 2, 3, 4, y 5 **filósofos**.
* Ejecutas la aplicación con 3, 5 ó 7 **tenedores**.
* En el código se omite el uso de **Synchronized en los métodos**.
* Es **importante** que anotes tus observaciones de ésta actividad en tu **reporte**.

En esta parte es necesario que por lo menos existan dos filósofos, ya que si solo tuviéramos uno, no cumple con las condiciones necesarias para comer, se quedaría solo pensando y moriría de hambre; al tener dos o más filósofos debe de existir el mismo número o más tenedores para que puedan comer, si existen menos tenedores que filósofos, solo algunos podrían comer. Por ejemplo si hay 3 tenedores y 5 filósofos únicamente un filósofo comería mientras que los 4 restantes piensas, además de que dos filósofos no tendrías cubierto a la derecha e izquierda para levantar. Sin los métodos synchronized se genera una excepción al momento de ejecutar y el programa no funciona correctamente.

**Actividad 3 - Limpieza**

La actividad está prácticamente terminada, sin embargo revisa limpia el código identificando los métodos que haz implementado.

* **Identificar** los métodos implementados para ambos proyectos.
* **Eliminar** los bloques de código que digan **"Completar código"**
* **Modificar** la documentación al inicio de cada clase con tus datos personales.
* Recordando que la documentación debe ser siguiendo las guías de [Javadocs](http://en.wikipedia.org/wiki/Javadoc)

**R=** Esta parte va adjunta en el código enviado.

**Actividad 4 - Preguntas**

En el reporte de ésta práctica, además de mostrar y explicar el desarrollo de las actividades deberás responder a las siguientes preguntas:

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

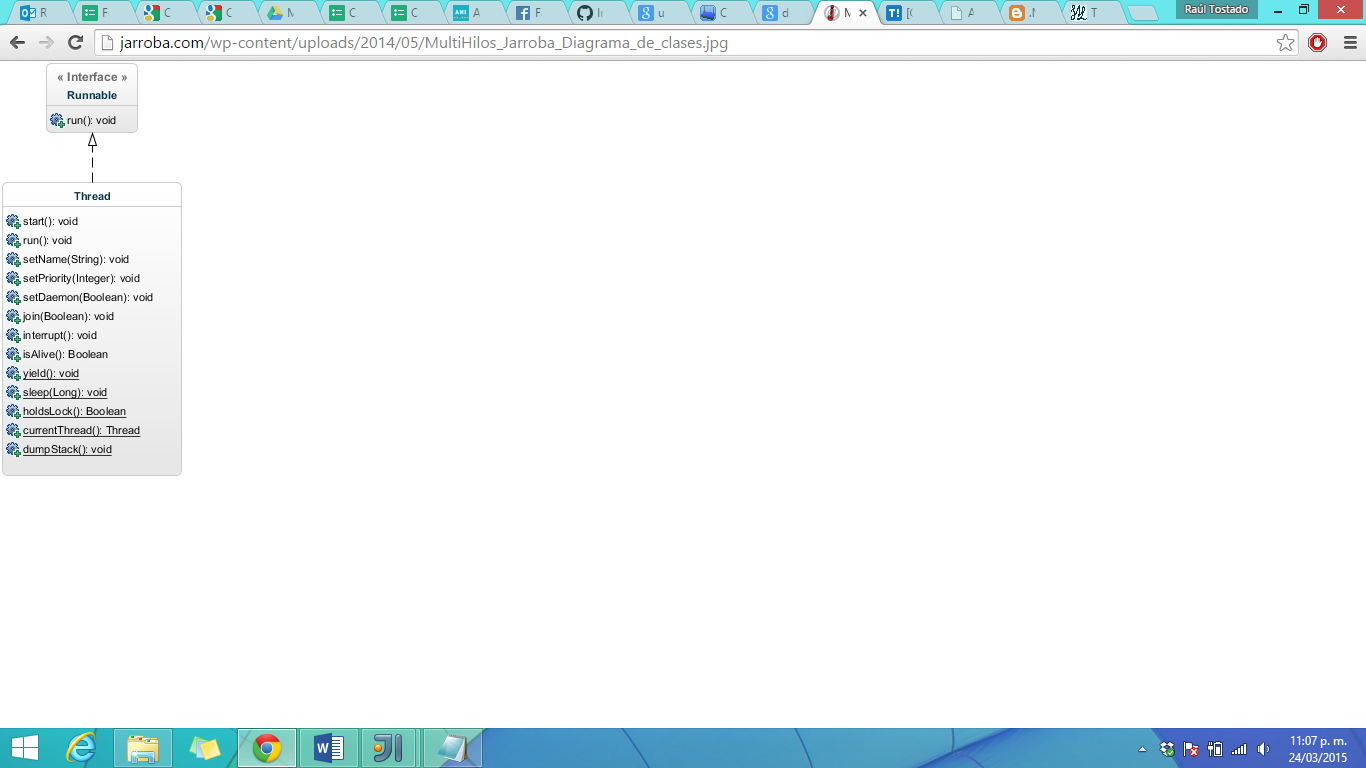
**R=** El método Wait() libera los recursos, en cambio Sleep() conserva los recursos, únicamente pausa su ejecución.

* ¿Para qué se utiliza **Lock**?

**R=** Es un bloque de instrucciones que ejecuta un solo hilo. Quiere decir que el Thread que lo invoca tiene acceso a esa instancia del objeto y cualquier otro objeto que quiera acceder tendrá que esperar a que el anterior acabe su ejecución. Se apropia del objeto (Tenedor).

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

**R=** Thread es una clase de la cual se va a heredar y Runnable es una interface que se va a implementar



* ¿Por qué es importante el uso de **Synchronized**?

**R=** Es importante porque asegura que un solo Thread pueda tomar el control del monitor, si un segundo o más Threads quisieran el control del monitor, tendrá que esperar a que el primero lo termine de usar.

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

**R=** Un monitor constituye la interface de acceso al recurso compartido, ofrece una mayor seguridad, robustez y escalabilidad; mientras que un semáforo es un objeto que es utilizado para controlar el acceso a un recurso compartido, también permite limitar el número de procesadores que acceden a un recurso compartido.

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

**R=** Porque puede ser que se de en el caso que dos filósofos que se sientan juntos estén ambos hambrientos, por lo cual solo uno podría comer mientras que el otro esperar, si es larga la espera morirá de hambre

**Conclusiones**

Considero fue muy importante la realización de esta práctica, ya que un año atrás comenzamos a ver el funcionamiento de un Thread, pero no quedó muy clara la idea de cómo funcionaban y cuál era su utilidad. Con lo aprendido al realizar esta práctica seremos capaces de implementar de una mejor manera el uso de un Thread en algún programa que lo requiera.