Apellidos:	**Nombre:**

La duración máxima del examen será de 2:30 horas.

Ejercicio 1) Sincronización en un programa concurrente (4p)

¿Cómo se puede implementar la exclusión mutua con un semáforo? (0.5p)

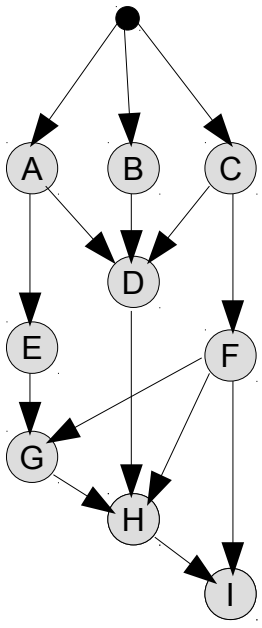
Explica las dos formas que existen en Java de implementar la exclusión mutua sin usar semáforos. Pon ejemplos de código. (1p)

¿Cómo se implementa la sincronización condicional con un semáforo? (0.5p)

¿Qué herramientas de alto nivel existen en Java para implementar la sincronización condicional? Explica el funcionamiento de 3 herramientas distintas que conozcas (2p)

Ejercicio 2) Programa Concurrente (3p)

Implementa un programa Java completo con 3 procesos que ejecute las siguientes tareas cumpliendo con las restricciones indicadas con las flechas. Es decir, la tarea D no puede comenzar hasta que no hayan terminado las tareas A, B y C. Si dos tareas no tienen restricciones entre ellas, deberían ejecutarse en paralelo. Puedes utilizar cualquier herramienta de sincronización que no sea un semáforo. Simula la ejecución de cada tarea como la impresión por pantalla de la letra indicada en el círculo. Implementa el código completamente (Creación de hilos, método main, etc.). No se puede utilizar SimpleConcurrent.



Solución:

Ejercicio 3) Programa Concurrente (3p)

Se dispone de una librería de comunicaciones y se quiere implementar varios tests para verificar que la librería funciona correctamente. La librería ofrece un cliente de WebSockets para conectarse a un servidor. Para realizar una conexión con un servidor se usa un código similar al siguiente:

```
WebSocketClient client = new WebSocketClient("ws://server/path");

client.connect();

client.sendMessage("message");
```

Como una comunicación de websockets es bidireccional, el cliente también puede recibir mensaejes del servidor, para ello registra un manejador de mensajes usando una expresión lambda:

```
WebSocketClient client = new WebSocketClient("ws://server/path");

client.onMessage(message -> System.out.println("Message:"+message));

client.connect();
```

Por último, el cliente permite al desarrollador ejecutar código cuando el cliente se desconecta del servidor (por cualquier motivo). Uno de los motivos por los que se puede cerrar la conexión es porque el cliente la cierra de forma explícita con el método "close". Para utilizar esta funcionalidad se usaría un código similar al siguiente:

```
WebSocketClient client = new WebSocketClient("ws://server/path");

client.onClose(reason -> System.out.println("WebSocket closed for "+reason));

client.connect();

//..

client.close();
```

Para poder ejecutar los manejadores de eventos (onMessage y onClose) el cliente de websockets utiliza hilos creados internamente (gestionados mediante un pool de hilos). Es decir, los manejadores no se ejecutan en el mismo hilo que crea el objeto cliente y llama a los métodos connect, sendMessage o close.

Para verificar que el funcionamiento del cliente de websockets es el adecuado se quieren implementar varios tests:

- **1) Test onClose:** Verificar que cuando se invoca el método close el manejador registrado en onClose es ejecutado antes de 5 segundos. Para este test se puede usar la URL ws://server/path.
- **2) Test onMessage:** Verificar que el manejador onMessage funciona correctamente y recibe los mensajes enviados por el servidor. Para ello, se puede usar la URL ws://server/echo. Esta URL implementa un servidor de eco, que reenvía al cliente cada mensaje que se se le envía. Para verificar que el cliente está implementado correctamente, hay que verificar que el contenido del mensaje enviado es el que se recibe realmente (y no cualquier otro). Además, no sólo hay que comprobar que un mensaje enviado es recibido, sino que hay que comprobar también que si se envían varios mensajes (5) estos se reciben en el orden en que fueron enviados.

Solución: