Ejercicio 4: Contador

Simula el comportamiento de un grupo de personas (10, por ejemplo) que intentan incrementar, una a una, el valor de un único contador. Crea 3 clases:

1. **Main.java:** Instancia todos los objetos e inicia la simulación.
2. **Person.java:** Estos son los hilos o runnables que llamarán al contador.
3. **Counter.java:** Es un objeto compartido entre todas las personas.

Main.java

- Instancia el contador (solo un contador).
- Declara todas las personas que tienen acceso al contador (al menos 10 personas).
- Inicia la simulación.
- Espera a que todos los hilos terminen y luego imprime el resultado final del contador.

People.java

- Esta clase extiende de `Thread` o implementa `Runnable`.
- Necesita un atributo que sea el contador. Este contador se inicializa en el constructor.
- El comportamiento en el método `run` será un bucle que intenta incrementar el contador 1000 veces.

Counter.java

- Esta es una clase normal.
- Necesita un atributo que lleve la cuenta del número. Comienza en 0.
- También tiene un método llamado `increment` que, cada vez que es llamado, incrementa el número en 1 (por ejemplo, `a++`).

Objetivos: Simula hasta que encuentres un error. Ejemplo: El número de personas x los intentos de incrementar el contador no coincide con la cantidad final en el contador. Soluciona el error usando la palabra reservada `synchronized`.

Ejercicio 5: Parking de Consumidores y Productores

El objetivo es construir una aplicación que simule el mecanismo de un estacionamiento con un número limitado de espacios.

La clase **Parking** es responsable de gestionar el estacionamiento, indicando el número total de espacios y cuántos están disponibles. Si un coche intenta entrar al estacionamiento y no hay espacios disponibles, deberá esperar hasta que se le notifique que un espacio está disponible.

La clase **Car** representa a cada coche, que será el hilo intentando entrar en el estacionamiento. Cuando un coche se inicializa, trata de entrar al estacionamiento. Si no hay espacios libres, esperará hasta que se le notifique que hay espacios disponibles. Una vez aparcado, ocupará el espacio por un tiempo aleatorio y luego saldrá del estacionamiento, liberando un espacio y notificando que un espacio está disponible.

Ejemplo de ejecución:

- Coche 1 listo para aparcar
- Coche 5 listo para aparcar
- Coche 6 listo para aparcar
- Coche 2 listo para aparcar
- Coche 3 listo para aparcar
- Coche 1 ha aparcado
- Coche 3 ha aparcado
- Coche 4 listo para aparcar
- Coche 2 ha aparcado
- Coche 6 ha aparcado
- Estacionamiento lleno, el coche 4 tiene que esperar
- Estacionamiento lleno, el coche 5 tiene que esperar
- Coche 3 ha dejado un espacio libre.
- Coche 4 ha aparcado
- Coche 6 ha dejado un espacio libre.
- Coche 2 ha dejado un espacio libre.
- Coche 1 ha dejado un espacio libre.
- Coche 5 ha aparcado
- Coche 4 ha dejado un espacio libre.
- Coche 5 ha dejado un espacio libre.
- Proceso finalizado con código de salida 0.

Ejercicio 6: Productor-Consumidor de Patas y Tableros

El objetivo es sincronizar la línea de producción de mesas siguiendo estas instrucciones:

- Para fabricar una mesa, se necesitan 4 patas y 1 tablero.
- Hay un productor de patas que las deposita una por una en un almacén.
- El almacén tiene un límite de capacidad (6 patas). Cuando está lleno, el productor de patas deja de producir hasta que haya espacio disponible.
- Hay un productor de tableros que deposita los tableros uno por uno en el almacén. El almacén tiene un límite de capacidad (2 tableros). Cuando está lleno, el productor de tableros deja de producir hasta que haya espacio disponible.
- Hay un fabricante de mesas que, cuando hay al menos 4 patas y 1 tablero en el almacén, los saca del almacén y fabrica una mesa. Si no hay suficientes materiales disponibles, espera hasta que sean producidos.

Se te pide desarrollar una clase Java llamada **Warehouse** que sincronice este proceso, asegurando que la producción de patas y tableros se detenga cuando se alcance la capacidad máxima y que el sistema de fabricación de mesas no continúe si faltan piezas.

El monitor sigue la estructura dada, y debes completarla para que se ajuste al resto de las clases proporcionadas.

java
Copiar código

```
class WareHouse {
    private final int MAX_LEGS = 6;
    private final int MAX_TABLETOPS = 2;

    storeLeg() {
    }

    storeTabletop() {
    }

    buildTable() {
    }
}
```

Ejercicio 7: Consumidor-Productor de Sopas

Simula un escenario donde los clientes quieren comer sopa en un restaurante:

- Cada cliente será un hilo que compartirá el objeto restaurante.
- Dependerá de ti si los clientes aparecen todos de una vez o en intervalos.
- Todos los clientes eventualmente comerán sopa, incluso si tienen que esperar. Una vez que comen, se van y no regresan.
- Habrá un cocinero, que será otro hilo. El cocinero de sopa solo tiene 4 tazones de sopa, y su método de trabajo es el siguiente:
    - Prepara 4 tazones de sopa y luego se va a dormir.
    - Cuando se termina el último tazón de sopa, el cliente que lo tomó despierta al cocinero, quien entonces prepara 4 tazones más (mientras los clientes esperan) y vuelve a dormir.

Los tazones de sopa son una metáfora para un búfer con 4 ranuras.

Mediante constantes definidas al inicio de la clase **Main**, puedes configurar el ejercicio, por ejemplo, el número de clientes o la cantidad de tazones de sopa.

Ejercicio 8: Productor-Consumidor: Filósofos Comensales con Palillos

En concurrencia, los recursos y los hilos que los utilizan son fundamentales. El problema de los filósofos comensales resalta algunos de los problemas que surgen cuando los recursos se comparten entre hilos y provee soluciones.

El problema del deadlock El problema de los filósofos comensales establece que 5 filósofos en un restaurante se sientan en una mesa redonda para una comida. Para comer, cada uno necesita 2 palillos, pero hay un problema, solo se proporcionaron 5 palillos en la mesa.

Un filósofo hambriento solo puede comer si ambos palillos están disponibles. Intentará adquirir primero el palillo derecho y luego el izquierdo. De lo contrario, un filósofo deja los palillos y comienza a pensar de nuevo.

Cada filósofo se representa como un hilo y cada palillo como un recurso.

El filósofo Un filósofo solo puede realizar 2 acciones: pensar y comer. Un filósofo está pensando o comiendo y nada en medio.

- **Pensar:** Un filósofo debe dejar ambos palillos.
- **Comer:** Un filósofo debe estar en posesión de ambos palillos.

Para más soluciones y su implementación, puedes seguir los ejemplos de sincronización proporcionados en el texto original.
