**Concurrencia y sincronización**

Contenido

[1 Ejercicio bar 2](#_Toc148718747)

[1.1 Bar 1 3](#_Toc148718748)

[1.2 Bar 2 3](#_Toc148718749)

[1.3 Bar 3 3](#_Toc148718750)

[2 Bar cañas 4](#_Toc148718751)

[2.1 BarCaña1 4](#_Toc148718752)

[2.2 BarCaña2 5](#_Toc148718753)

[3 Taller 6](#_Toc148718754)

[3.1 Taller 1 7](#_Toc148718755)

# Ejercicio bar

Un bar quiere limitar el aforo. Para ello, se ha instalado una puerta de entrada automática, que siempre está cerrada, junto con un pulsador. Cuando un cliente quiere entrar, acciona el pulsador de la puerta de entrada y, si el bar no tiene su aforo completo, la puerta se abre y se ilumina un letrero indicando que puede entrar. En caso contrario, el cliente tendrá que esperar fuera hasta que se encienda el letrero.

Cuando un cliente quiere salir, utiliza una puerta distinta a la de entrada (para evitar que algún cliente intente colarse). Esta puerta permanece cerrada y tiene un pulsador que funciona de forma análoga a la de entrada, con la diferencia de que siempre que se acciona el pulsador la puerta se abre automáticamente. Cuando sale un cliente, si había clientes esperando para entrar, se iluminará el letrero que indica que se puede entrar, y la puerta de entrada se abrirá para permitir la entrada de uno de los que están esperando entrar.

En el ordenador central se ejecuta una aplicación java multihilo para controlar el sistema descrito. Esta aplicación está compuesta por las siguientes clases:

**Clase Cliente**. Los clientes se representarán mediante hilos instanciados a partir de esta clase

**public class** Cliente **implements** Runnable

{

Bar bar;

**public** Cliente (Bar bar) {

**this**.bar=bar;

}

**public void** run() {

bar.entrar(); // Esta acción simula accionar el pulsador de

// entrada y esperar a que se abra la puerta

Thread.Sleep(10000);

// Estar en el bar

bar.salir(); // Esta acción simula accionar el pulsador de

// salida y esperar a que se abra la puerta.

}

}

**Clase Bar**. Al arrancar la aplicación, se instancia un objeto de esta clase. Contiene los métodos a los que

invocan los hilos que representan a los clientes:

**public class** Bar

{

// Atributos para controlar el acceso al bar.

/\*\*

\* **@param** aforo Nº de clientes que caben en el local

\*/

**public** Bar (**int** aforo)

{

// inicializar el bar

}

**public void** entrar()

{

// Permitir la entrada de un cliente si no se ha alcanzado

// el aforo, y quedarse bloqueado en caso contrario.

abrirPuertaE(); // Muestra el letrero de “se puede entrar” y

// abre la puerta de entrada

}

**public void** salir()

{

abrirPuertaS(); // Muestra el letrero de “se puede salir” y

// abre la puerta de salida

// Anotar que un cliente sale del bar, y desbloquear a

// quien pudiera estar esperando, para que entre uno más

}

}

Suposiciones:

* Los clientes que han pulsado el pulsador para entrar, siempre esperan (no se van) hasta que el indicador les diga que pueden entrar, y entonces entran.
* Los clientes siguen el protocolo de entrada y salida de forma estricta, es decir, siempre entran y salen de uno en uno por las puertas.

## Bar 1

En la clase Bar implementar los métodos entrar() y salir(), añadir los atributos que sean necesarios y añadir al constructor las acciones que sean precisas, usar **synchronized**

## Bar 2

Igual que el anterior, pero usando **Semaphore**

## Bar 3

Una variación del ejercicio anterior.

Ahora en el bar entran dos tipos de alienígenas:

* Ewok
* Gorax

Los ewok, al ser más adorable, siempre entrarán antes que los Gorax en cuanto quede libre un hueco en el bar.

Para ello usar ReentrantLock y sus Condition

Ejemplo de salida por pantalla

Texto

Descripción generada automáticamente con confianza media

# Bar cañas

En el bar “Cañas gratis” hay barra libre de cerveza. Para no dedicar un camarero a servir cañas, se han colocado varios grifos conectados a un mismo barril, que los clientes pueden usar libremente. El problema que tiene el dueño del local es cómo enterarse de que se ha acabado la cerveza y que hay que cambiar el barril. Para ello, ha diseñado un mecanismo que funciona de la siguiente manera:

* Cuando un cliente se sirve su caña, el grifo informa a un ordenador, de forma que se puede llevar la cuenta de las cañas servidas.
* Cuando el barril está agotado, se enciende un letrero en el grifo con el mensaje “Esperar al cambio de barril” y se avisa al camarero para que cambie el barril.

En el ordenador central se ejecuta una aplicación java multihilo para controlar el sistema descrito. Esta aplicación está compuesta por las siguientes clases:

**Clase Cliente**. Los clientes se representarán mediante hilos instanciados a partir de esta clase. Como hay clientes que beben más que otros, se ha simulado su comportamiento con un argumento del constructor que indica el nº de cañas que va a tomar cada cliente:

**public class** Bar

{

// Atributos para controlar las cañas que se sirven de cada barril

// y cuándo debe ser repuesto por uno nuevo.

/\*\*

\* **@param** nCañas Nº de cañas que pueden ser servidas de un barril

\*/

**public** Bar (**int** nCañas)

{

// inicializar el bar

}

**public void** tomarCaña ()

{

// Si el barril no está vacío, servirá la caña directamente.

// En caso contrario, avisará al camarero para que lo reponga, y

// se quedará bloqueado hasta que se haya cambiado el barril, y

// entonces servirá la caña.

servirCaña(); // Servir una caña

// anotar que se ha servido una caña

}

**public void** reponer()

{

// Esperar a que le avisen de que hay que cambiar el barril

cambiarBarril(); // Esta acción indica que se repone el barril

// anotar que se ha repuesto el barril

}

}

Suposiciones:

* Cuando un cliente obtiene que el barril está agotado se quedará esperando hasta que se produzca el cambio de barril y luego tomará su caña.
* El camarero siempre está pendiente de que le avisen del cambio de barril y lo cambia cuando se lo soliciten.

**Se pide:** En la clase Bar implementar los métodos tomarCaña() y reponer(), añadir los atributos que sean necesarios y añadir al constructor las acciones que sean precisas.

**Nota:** Por simplicidad no capturar o tratar las excepciones que arrojan las herramientas de concurrencia utilizadas para resolver el ejercicio.

## BarCaña1

Hacer el ejercicio con **synchronized**

## BarCaña2

Para ello usar ReentrantLock y sus Condition

# Taller

Un taller de mecánica rápida ofrece dos tipos de servicios: cambio de aceite (CA) y revisión general del vehículo (RG). Estos dos servicios son realizados siempre por los clientes, empezando por el cambio de aceite y luego la revisión general. El taller ofrece estos servicios de la siguiente manera: existen tres puestos de trabajo para el cambio de aceite y otros cinco puestos de trabajo para la revisión general.

Todos los puestos de trabajo son independientes y pueden realizar el servicio que prestan simultáneamente.

Cuando un coche llega al taller se dirigirá a uno de los puestos de trabajo para el cambio de aceite. En caso de que no existan puestos disponibles, deberá esperar. Una vez ocupado el puesto, el conductor recogerá un bidón de aceite y realizará el cambio correspondiente. Existe un almacén con 100 bidones de aceite. Si el conductor se encuentra que no existen bidones en el almacén, debe dar un aviso a un montacargas automático que se encuentra permanentemente a la espera de la orden de reposición. El montacargas repondrá los bidones con 100 unidades y quedará a la espera de una nueva orden de reposición.

Al terminar el cambio de aceite se dirigirá a uno de los puestos de trabajo para la revisión general. En caso de que no existan puestos disponibles, deberá esperar. El sistema está modelado mediante la clase Taller. Cada conductor es un hilo que ejecuta el método vehiculo(), y el montacargas es otro hilo que ejecuta el método robotMontacargas().

**public class** Taller {

**final int** PUESTOS\_CA=3;

**final int** PUESTOS\_RG=5;

**final int** MAX\_BIDONES=100;

// Atributos

**public** Taller() {

// Inicializar atributos de la clase

}

**public void** vehículo() {

// Intentar meterse en un puesto de trabajo CA, si no hay disponibles esperar

// Recoger un bidón de aceite, si no hay disponible avisar al montacargas y

// esperar

cambiarAceite();

// Intentar meterse en un puesto de trabajo RG, si no hay disponibles esperar

revisionGeneral();

}

**public void** robotMontacargas() {

**while** (**true**) {

// Esperar a que le avisen que faltan bidones de aceite

reponerBidones();

// Avisar a quien estuviera esperando por bidones

}

}

}

**Se pide:** escribir en java el código necesario de la clase Taller teniendo en cuenta:

* Indicar qué atributos pueden ser necesarios para implementar las funcionalidades de la clase Taller
* Añadir al constructor las acciones precisas para inicializar la clase.
* Implementar el método vehículo()
* Implementar el método robotMontacargas()

Tenga en cuenta lo siguiente:

* Por simplicidad, no es necesario tener en cuenta las posibles *excepciones* que se puedan producir.
* Todos los coches usan el mismo tipo de aceite.
* Debe existir la máxima concurrencia entre hilos que ejecutan el método vehículo().
* Los métodos cambiarAceite(), revisionGeneral() y reponerBidones() no presentan problemas de concurrencia.

## Taller 1

Resuelve el problema como consideres (con la máxima concurrencia), recuerda que un mismo problema puedas juntar **synchronized, lock y semaforos**