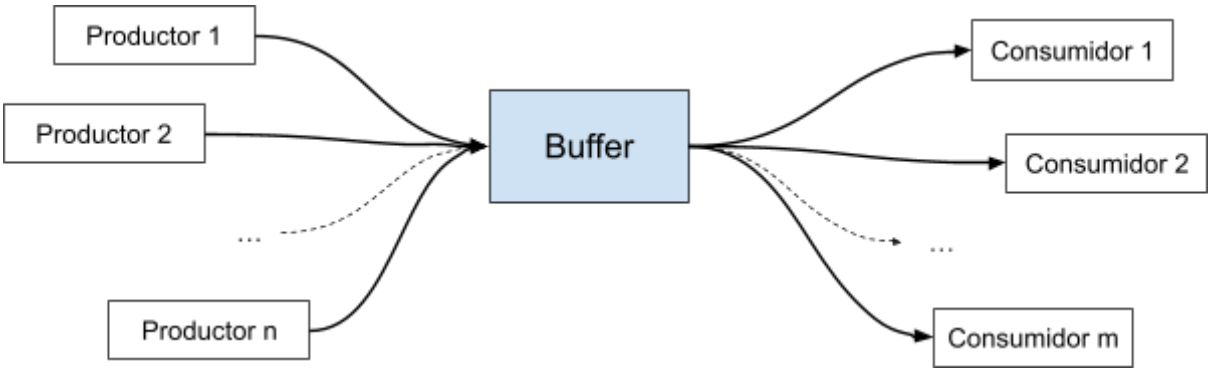
Programación multihilo

Productor - Consumidor

Índice:

Enunciado

En un entorno de concurrencia, es común que un conjunto de hilos genere datos (Productores) y los almacene en **algún lugar** donde otro conjunto de hilos accederá para obtenerlos y procesarlos (Consumidores). Ese lugar suele llamarse ***buffer***.



En general, el ritmo al que los productores producen esos datos no es el mismo al que los consumidores los procesan. Ambos grupos deben interactuar de forma **segura** a través del *búfer*. Por tanto, ese *buffer* debe coordinar a productores y consumidores para que sincronicen su actividad y no haya problemas de condiciones de carrera, ni pérdidas o duplicidades de datos.

Por eso vamos a hacer este ejercicio de dos formas:

- Usando el método clásico de `.wait()` y `notify()`.
- Gestionando una `BlockingQueue`.

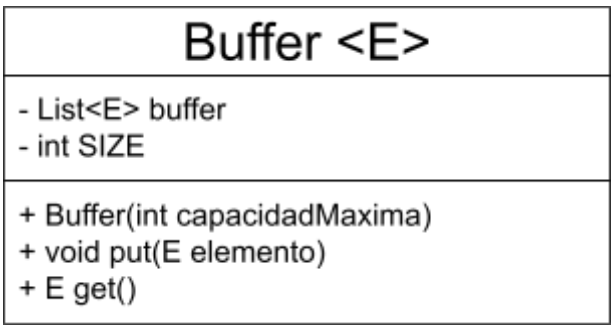
Usando wait() y notifyAll()

Implementa en Java el patrón Productor-Consumidor utilizando los mecanismos de sincronización de bajo nivel de Java (`synchronized`, `wait()`, `notifyAll()`) para gestionar el acceso a un *buffer* compartido implementado con una lista.

Buffer

Implementa la clase genérica `Buffer<T>` que actúa como el almacén de mensajes entre productores y consumidores.

El *buffer* tendrá la siguiente estructura:



Admitirá una **lista de cualquier tipo de objetos** (genérico E).

El atributo `SIZE` será la constante que define el **tamaño máximo** de la lista.

- **`Buffer(int capacidadMaxima)`**
  En este constructor se inicializa el estado del buffer, estableciendo el tamaño de la lista.

- **`void put(E elemento)`**
  Añade un elemento a la lista sólo si la lista no está llena. De ser así, deberá esperar a que haya hueco.

  Por la naturaleza de esta operación, el método debe estar **sincronizado**.

- **`E get()`**
  Saca el primer elemento de la lista y lo devuelve. Si la lista está vacía, debe esperar.

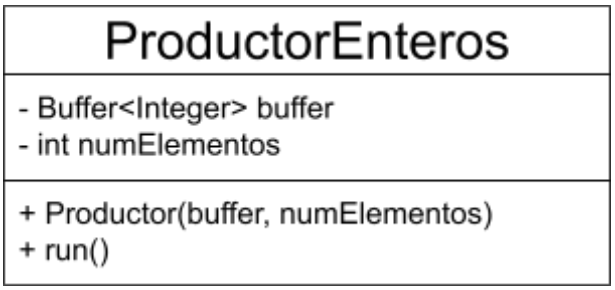
  Al igual que en el método anterior, el método debe cuidar la sincronización para operar.

⚠️ Tanto `put` como `get` deben escalar la interrupción `InterruptedException`.

⚠️ En cada caso habrá que notificar **cuándo hay elementos para consumir** y **cuándo hay hueco** para introducir más datos en el *buffer*.

ProductorEnteros

Esta clase representa a hilos que producirán números enteros y los guardarán en el *buffer*.



Los atributos de esta clase son el `buffer` en el que depositará la cantidad de números enteros indicados en `numElementos`.

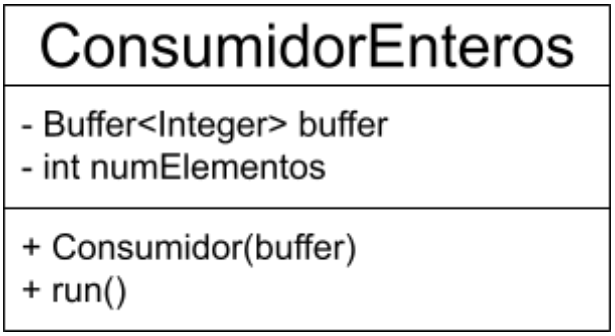
En su constructor recibe la referencia al buffer y la cantidad de números que debe introducir en él.

El método **`run()`** debe generar la cantidad de números enteros **aleatorios** indicada en `numElementos` y lo hará:

1. Generando un entero aleatorio.
2. Insertándolo en el buffer.
3. Imprimiendo un mensaje claro indicando **el hilo** que lo ha hecho y **el número** que ha insertado.
4. Dejando una **pausa entre 0 y 3 segundos** antes de generar el siguiente para simular distintas cargas de trabajo.

ConsumidorEnteros

Esta clase representa a hilos que sacarán números enteros del *buffer*.



El único atributo de esta clase es el `buffer` del que extraerá los números enteros. En su constructor recibe la referencia al buffer.

El método **`run()`** debe ejecutarse **indefinidamente** mientras el hilo principal esté activo.

Después de consumir un elemento, debe hacer una pausa aleatoria **entre 0 y 1 segundos** para simular distintas cargas de trabajo.

Debe imprimir un **mensaje claro** cada vez que consume un elemento, **indicando el hilo** y **el valor** consumido.

Main

Crea un `Main.java` donde:

- Se creará el buffer para albergar tantos números enteros como años tengas tú.
- Crear varios hilos productores (4 ó 5) con distintas cantidades a generar.
- Crear uno o, como máximo, dos consumidores.
- El hilo principal **debe esperar** a que los productores terminen.

Lanzar la ejecución y ver el resultado por consola.

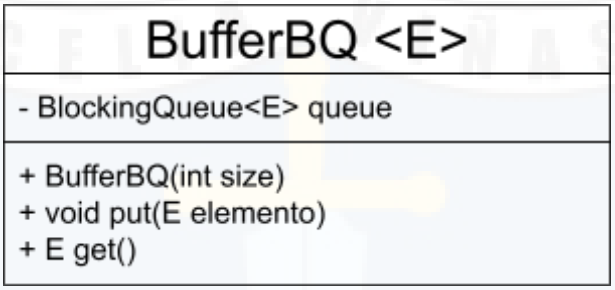
Usando BlockingQueue

Vamos a hacer una versión más moderna del ejercicio usando BlockingQueue.

BufferBQ

Implementa la clase genérica `Buffer<T>` que actúa como el almacén de mensajes entre productores y consumidores.

El *buffer* tendrá la siguiente estructura:



El constructor establecerá el tamaño de la cola con el valor introducido como parámetro.

Revisa la documentación de BlockingQueue. Los métodos put y que son mucho más sencillos…

Productor / Consumidor

Se pueden usar las clases anteriores.

MainBQ

Crea un `MainBQ.java` que hará lo mismo que el Main anterior.

Entrega

Adjunta en la entrega tanto el **código** como un **documento** (en formato Google Documentos) que explique ambos casos.