**Java Util Concurrent**: messo a disposizione da Java che permette di affrontare problemi di concorrenza in modo più ad alto livello, fornendoci classi che ci risolvono problemi comuni.

Andiamo a creare la prima interfaccia, nel nostro caso public interface BlockingQueue<T>, osservando il manuale di Java nella sezione BlockingQueue, andiamo a selezionare solamente alcuni metodi come:

public void put(E e) throws InterruptedException // il quale inserisce un elemento alla cosa, ma se non c’è spazio si mette in attesa.

Public T take() throws InterruptedException // il quale toglie e rimuove l’elemento alla testa, ma se non c’è si mette in attesa.

Public int remainingCapacity() // ritorna in termine di numero quanto spazio è ancora disponibile nella coda.

Public boolean isEmpty() // controlla se la coda è vuota

Public void clear() // svuota la coda

Nota: Quando si ha un’astrazione si deve avere un’interfaccia.

LinkedBlockingQueue si appoggia a una linkedlist e non è limitata superiormente

**#FILE LinkedBlockingQueue.java**

Abbiamo una public LinkedList<T> queue, la utilizziamo per mantenere i dati, poi mettiamo l’implementazione della nostra interfaccia cosi da sincronizzare gli accessi (LinkedBlockingQueue)

Poi implementiamo la put sopracitata

public void put(E e) throws InterruptedException{

synchronized(queue){

queeu.addLast(object);

if(queue.size()==1)

queue.notify();

}

}

Nella put quindi abbiamo una sincronizzazione di queue in modo che se abbiamo più accessi concorrenti, proteggiamo l’oggetto queue con una synchronized, questo ci permetterà di garantire che il codice dentro al blocco synchronized sia eseguito in mutua esclusione, quindi si abbiamo due put non c’è un problema di contesa nell’accesso alla risorsa coda perché mettiamo sezione critica e quindi mettendola solo una delle due va a avanti; La put che è all’interno della sezione fa addLast mette in fondo, poi se la coda ha dimensione 1 che vuol dire che prima della put la coda era vuota, fa una notify, che avremmo preferito scrivere notifyall così da svegliarli tutti, ma visto che la coda è di lunghezza 1, non ha senso svegliarli tutti perché chi è in wait? Quelli che cercano di leggere dalla coda vuota, se il size è 1 facciamo una notify e svegliamo solo uno di quelli che sono in coda perché la coda è vuota, non ci serve svegliarli tutti perché comunque 1 passa tira via l’oggetto dalla coda e gli altri si rimettono in wait.

public T take() throws InterruptedException{

synchronized(queue) {

while(queue.size() == 0)

queue.wait();

T object = queue.removeFirst();

if(queue.size() >0)

queue.notify();

return object;

}

}

La take() cosa fa? Se la coda è vuota si blocca, nel momento che almeno un elemento è nella coda, questa si sblocca e verrà svegliata da una delle notify della put() , allora removeFirst() tira via un oggetto dalla coda e a questo punto se la coda contiene ancora elementi facciamo notify.

public boolean isEmpty(){

synchronized(queue){

return queue.isEmpty();

}

}

public int remaingCapacity(){

return Integer.MAX\_VALUE;

}

}

remaingCapacity() ritorna numero di celle disponibili da parte della coda e ritorna integer.MAX\_VALUE se la coda non è limitata superiormente

public void clear() {

synchronized(queue){

queue.clear;

queue = null;

}

}

La clear() svuota la coda in mutua esclusione;

**#PROGETTO EXAMPLE02**

**#FILE PRODUCER.JAVA**

Cosa fa il nostro file produttore?

Parte con un certo id che gli viene attribuito al momento della costruzione, riceve come argomento un Blockingqueue, se l’ID è negativo gli viene lanciato IllegalArgumentException, dobbiamo imparare a verificare sempre le precondizioni minime dei nostri argomenti, controlliamo anche che sia diverso da nullo , cosi il nostro produttore contiene un ID e una coda se non è così viene lanciata un’eccezione di tipo RunTimeException, a questo punto implementiamo un runnable perché poi faremo un thread, cioè metteremo i producer ognuno sul suo thread a produrre dati e a metterli nella coda.

Questo producer come tutti gli altri conta da 0 a 5 costruisce un messaggio che non è nient’altro una stringa con l’id e il contatore , stampa un log sulla console e poi aspetta un tempo casuale compreso da 100 a 150 ; a questo punto viene ripetuto questo ciclo e vengono rifatte stampe-put stampe-put, siccome qui abbiamo a che fare con una BlockinqQueue che sarà una linckedBlockingQueue la limitazione superiore non c’è a questo punto la put non si blocca mai, quella che si può bloccare è la take() fatta dal consumatore,

**#FILE CONSUMER.JAVA**

Il nostro consumatore parte con un ID che deve essere maggiore uguale a zero e un oggetto queue non nullo e a questo punto costruisce il proprio stato; e poi scoda via con un ciclo 5 messaggi mentre li scoda tra un messaggio e altro fa un’attesa da 40 a 140 mille secondi.

**#FILE EXAMPLE02**

Il main cosa fa? Costruisce 5 consumer e li attiva siccome consumer implementa Runnable come abbiamo visto possiamo attivare un thread e fare start di consumer, quindi partono i 5 consumer, la coda è vuota e si bloccano tutti e 5, poi nel mentre questi partono i producer , questi incominciano subito a riempire la coda e a questo punto il nostro esempio2 se tutto farà produrre 5 messaggi a ogni producer con totale 25 e poi questi terminano e i consumer sono 5 leggerano ognuno 5 messaggi e ne leggeranno 25 totali, mentre vengono letti vengono stampati.