Come mai non riuscivamo a lavorare con la coda senza esplicitarne la classe concreta?

L’unica interfaccia è l’interfaccia Queue<E> perché add aggiungi alla testa e remove è coordinata con l’add.

Andiamo a modificare la soluzione di ieri, andiamo a costruire una lista concatenata (ogni nodo è collegata solo al nodo successore); L’ultimo valore punterà al successore NULL.

Noi costruiamo la nostra classe nodo statica, statica perché non accede allo stato del contenitore,

private static class Node {

private T value;

private Node next;

}

T è un parametro perché un parametro del tipo contenitore e non è visibile, due soluzioni: la prima togliamo static, e all’interno di node abbiamo un riferimento al contenitore e dal punto di vista di memoria stiamo aggiungendo un reference in più; oppure la mettiamo statica e quel T lo facciamo diventare un Object, e perdiamo il tipo generico T ma ci guadagniamo una reference; Nel nostro caso scegliamo di mantenere lo static e quindi di tenere il tipo generico T.

Creiamo la classica lista concatenata e poi iniziamo a fare i metodi.

Public void clear() {

Synchronized(mutex){

Head = tail = null;

}

}

Tutti gli oggetti vengono liberati.

Public boolean isEmpty(){

Synchronized (mutex){

return head == null;

}

}

Se la coda è nulla ritorna true altrimenti ritorna false.

Public void put (T object){

Synchronized (mutex) {

Node node = new Node ();

node.next = null;

node.value = object;

If (tail == null) {

Head = new Node ();

head.next = null;

head.value =object;

tail = head;

} else {

tail.next = node;

tail = node;

}

If (head.next == null)

mutex.notify();

}

A questo punto dobbiamo aggiungere alla coda, e la condizione lista vuota è testa = coda = null, quindi se la coda è nulla allora faremo che la testa è uguale a tail che è uguale node e avrà next come nullo, se invece non è vuota next di tail diventa il nodo e il nuovo tail diventa il nodo.

Facciamo le notifiche quando la coda ha unicamente un elemento, e basta controllare che head.next è nullo allora la lista contiene solamente un elemento.

Public T take () throws InterruptedException {

Synchronized (mutex) {

While(head==null)

Mutex.wait();

T result = (T)head.value;

head = head.next;

if (head == null)

tail = null;

else

mutex.notify();

return result;

}

}

Quando dobbiamo fare la wait sul mutex, quando la list è vuota quindi quando head = tail = null, nel momento in cui usciamo dall’attesa e vuol dire che c’è head ed è diversa da null ed è il valore che vogliamo ritornare. Se spostandoci avanti arriviamo alla fine della coda siamo nella situazione in cui head = tail = null, e per qualche motivo head è diventato null allora deve esserlo anche tail; Quindi quando entriamo a Object result = head.value, head è diverso da nullo, spostiamo avanti head, avanti di un passo e non tocchiamo tail e a forza di spostare avanti prendiamo via anche tail, head diventa nullo. Se la coda non è vuota facciamo una notifica, a questo punto ritorniamo result ma per farlo deve essere di tipo T, siccome head.value è un object la cosa più ragionevole è fare un cast (T).

Senza le conditions, i lock espliciti non sono convenienti, usare i lock aumenta notevolmente i bug di sincronizzazioni, rispetto alle sezioni critiche con i synchronized.

Un lock può essere acquisito e rilasciata, solo un thread può acquisire il lock e se non può, si mette in attesa.

La lock può ovviamente lanciare un InterruptedException perché si piò mettere in attesa come detto prima.

Cosa succede se un thread cerca di prende un lock e lo ha già lui? Se il lock l’ho già preso non mi fermo e questo tipo di lock si chiama rientrante, ma ci sono ance quelli non rientranti e invece di andare avanti mi blocco e questi si usano in situazioni dove vengono gestiti con più attenzione.

La classe concreta la chiamiamo ReentrantLock

Cosa ci mettiamo dentro? Andiamo a permettere a un thread di acquisire un lock e contando quante volte lo acquisisce, questo ci serve perché l’unlock sarà fatta solamente quando contatore sarà zero, ma la prima unlock non sarà quella che libera, perché per liberare deve esserci lo stesso numero di lock che unlock, allora li la risorsa sarà libera.

CI serve un oggetto mutex per fare effettivamente la sincronizzazione, poi ci serve contare gli accessi agli lock e agli unlock fintanto che vengano fatte queste chiamate; Ci serve sapere anche chi è l’owner del lock perché solo il thread corrente è l’owner del lock si entra al contrario dovremo fare un accesso sincronizzato, quindi un riferimento a un oggetto Thread.

Public final class ReentrantLock implements Lock {

Private Thread owner;

private int counter;

private Object mutex;

public ReentrantLock(){

this owner = null;

this.mutex = new Object();

this.counter = 0;

}

@Override

Public void lock() throws InterruptedException {

if(counter < 0)

throw new IllegalMonitorStateException(“ counter <= 0”);

Thread currentThread = Thread.currentThread();

synchronized(mutex){

while (owner != null && owner != currentThread)

mutex.wait();

if(owner == null)

owner = currentThread;

counter ++;

}

}

Il metodo Lock crea una sezione critica sull’oggetto mutex, perché tutte le cose fatte all’interno volutamente devono essere in mutua esclusione, es: capire chi è l’owner se controllo chi è e non ho la mutua esclusione magari poco prima che controllo chi è viene cambiato e non riusciamo a garantire la coerenza di quello che stiamo facendo;

@Override

Public void unlock() throws InterruptedException {

synchronized(mutex){

if (owner != Thread.currentThread)

throw new IllegalMonitorStateException(“owner != Thread.currentThread()”);

if(counter <= 0)

throw new IllegalMonitorStateException(“ counter <= 0”);

counter--;

if(counter == 0) {

owner = null;

mutex.notifyAll();

}

}

}

Se lo stato attuale del lock è coerente con le nostre aspettative allora decrementiamo il contatore; se il contatore è arrivato a zero siamo nelle condizioni dove il lock viene rilasciato, quindi l’owner diventa null e viene fatta una notifyAll();