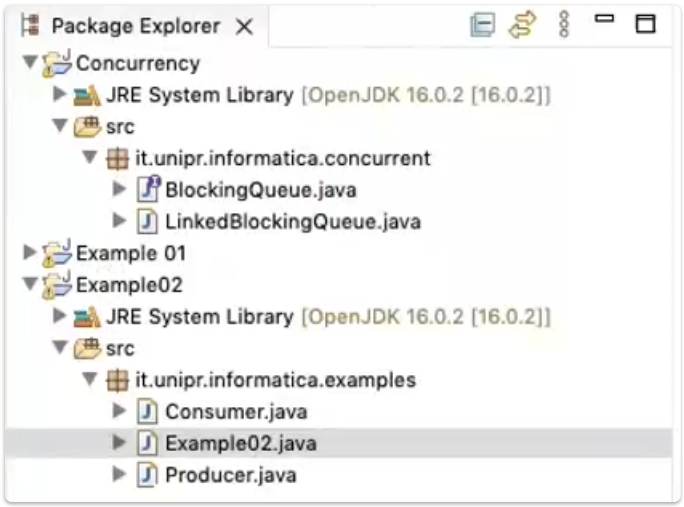
Example02

Table of contents

Astrazioni di alto livello

In una versione semplice di quello che abbiamo visto in Example01, il livello di astrazione in cui separavamo `Notifier` e `Waiter` non era molto alto.
Programmare vicino alla macchina, vicino al SO, ha vantaggio se quello che andiamo a fare è critico, causa quantità risorse di calcolo elevate, non lo è tuttavia sempre.

JAVA permette di affrontare problemi di concorrenza fornendo **astrazioni** di *alto livello*, perché non possiamo sempre pensare a risolvere problemi di sincronizzazione. Ci serve qualcosa abbastanza generale da poter usare spesso, che sfrutti bene il sistema a disposizione e che permetta al nostro software di acquisire nuove funzionalità.



`Concurrency`

`BlockingQueue.java`

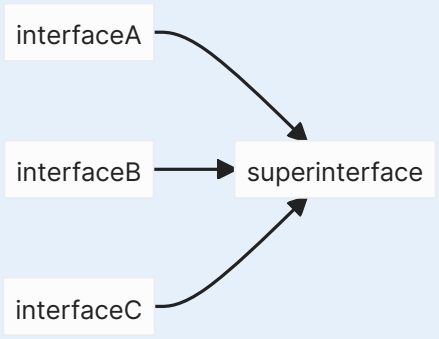
Esiste una coda per accodare thread.
Inseriamo elementi o togliamo elementi dalla nostra coda, in modo FIFO.

- *creazione*, usando la classe, manipolandola con l'interfaccia;

- *distruzione* della coda;
- *is empty*;
- *is full*, ritorna di solito sempre `TRUE` (lunghezza arbitraria, lunghezza limitata);
- *enqueue* per mettere in coda;
- *dequeue* per togliere dalla coda.

La coda bloccante ha in più che se piena si metterà in attesa, aspettando finché lo spazio non si crea, oppure se vuota la dequeue si bloccherà.

Nota sulle interfacce e super-interfacce



Una super-interfaccia necessita che tutti i metodi delle interfacce che la compongono, venghino implementati. Per esempio: `BlockingQueue<E>` è interfaccia della super-interfaccia `Queue<E>`.

BlockingQueue documentation Oracle

La nostra `BlockingQueue` ha argomento `<E>` e ha delle super-interfacce (che non ci interessano). Dei suoi metodi, guardiamo solo:

- `void put(T e)`,
  per aggiungere aspettando nel caso non ci sia spazio
  - lancia quindi `InterruptedException` se non c'è;
- `public T take()`
  rimuove la testa della coda e se non la contiene, aspetta
  - lancia quindi `InterruptedException` se non c'è;
- `int remainingCapacity()`
  ritorna quanto spazio ci rimane nella coda se definito (altrimenti `MAX_VALUE`);
- `boolean isEmpty()`
  verifica in modo sincrono se la coda è vuota;
- `void clear()`
  per liberare tutte le reference che l'oggetto ha, senza garanzia che gli oggetti vengano liberati per davvero.

`BlockingQueue.java`

```java
package it.unipr.informatica.concurrent;

public interface BlockingQueue<T> {
        // aggiungo
        public void put(T e)
                throws InterruptedException;
        // rimuovo
        public T take()
                throws InterruptedException;
        // elementi disponibili di coda limitata superiormente
```

```
        public int remainingCapacity();
        // verifichaimo la coda vuota o meno
        public boolean isEmpty();
        // eliminiamo reference alla coda
        public void clear();
}
```

`LinkedBlockingQueue.java`

Implementa l'interfaccia che abbiamo visto sopra.

LinkedBlockingQueue documentation Oracle

```
package it.unipr.informatica.concurrent;

import java.util.LinkedList;
import java.util.List;

public final class LinkedBlockingQueue<T>
        implements BlockingQueue<T> {
                private LinkedList<T> queue;

                public LinkedBlockingQueue() {
                        this.queue = new LinkedList<>();
                }

                @Override
                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;
                        }
                }

                @Override
                public void put(T object) {
                        synchronized (queue) {
                                queue.addLast(object);
                        if (queue.size() == 1)
                        queue.notify();
                        }
                }

                @Override
                public boolean isEmpty() {
                        synchronized (queue) {
                                return queue.isEmpty();
                        }
                }

                @Override
                public int remainingCapacity() {
                        return Integer.MAX_VALUE;
                }

                @Override
                public void clear() {
```

```
                        synchronized (queue) {
                                queue.clear();
                        }
                }
        }
```

`put()`

```
@Override
public void put(T object) {
        synchronized (queue) {
                queue.addLast(object);

        if (queue.size() == 1)
                queue.notify();
        }
}
```

La `notifyAll()` si potrebbe usare anziché `notify()`, ma siccome facciamo un controllo per verificare se la coda ha qualcuno già al suo interno, non è necessaria.

`take()`

```
@Override
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;
        }
}
```

Uno alla volta acquisiamo l'elemento (se ce ne servono due, lanciamo 2 volte)

`remainingCapacity()`

```
@Override
public int remainingCapacity() {
        return Integer.MAX_VALUE;
}
```

Ritorna il numero di elementi ancora disponibili nella coda limitata superiormente, se non limitata, ritorna `MAX_VALUE`.

`isEmpty()`

```
@Override
public boolean isEmpty() {
        synchronized (queue) {
                return queue.isEmpty();
        }
}
```

Non riproduciamo la catena d'interfacce, `isEmpty()` e' gia' contenuta nella super-interfaccia `Collection`.

`clear()`

```
@Override
public void clear() {
        synchronized (queue) {
                queue.clear();
        }
}
```

La distruzione non avviene veramente, dovremmo mettere a `NULL` il riferimento alla `queue`, ma noi lo lasciamo nel caso serva ancora ad altri oggetti.

`Example02`

`Producer.java`

```
package it.unipr.informatica.examples;

import it.unipr.informatica.concurrent.BlockingQueue;

public class Producer implements Runnable {
        // id ⩾ 0
        private int id;
        // in ingresso
        private BlockingQueue<String> queue;
        public Producer(int id, BlockingQueue<String> queue) {
                if (id < 0)
                        throw new IllegalArgumentException("id < 0");
                if (queue == null)
                        throw new IllegalArgumentException("queue == null");
                this.id = id;
                this.queue = queue;
        }

        @Override
        public void run() {
                try {
                        for (int i = 0; i<5; ++i) {
                                String message = id + "/" + 1;
                                System.out.println("P" + id + " sending" + message);
                                queue.put(message);
                                System.out.println("P" + id + " sent" + message);
                                // attesa tra 100ms a 150ms
                                Thread.sleep(100 + (int) (50 * Math.random()));
                        }
                } catch (InterruptedException interrupted) {
                        System.err.println("Producer " + id + " interrupted");
                }
        }
}
```

Ciascun thread ha il compito di *produrre* 5 messaggi ciascuno e di fornirli per essere letti dal consumer.

`Consumer.java`

```
package it.unipr.informatica.examples;

import it.unipr.informatica.concurrent.BlockingQueue;
```

```
public class Consumer implements Runnable {
        private int id;
        private BlockingQueue<String> queue;
        public Consumer(int id, BlockingQueue<String> queue) {
                if (id < 0)
                        throw new IllegalArgumentException("id < 0");
                if (queue == null)
                        throw new IllegalArgumentException("queue == null");
                this.id = id;
                this.queue = queue;
        }

        @Override
        public void run() {
                try {
                        for (int i = 0; i < 5; ++i) {
                                String message = queue.take();
                                System.out.println("Consumer" + id + " received " + message);
                                Thread.sleep(40 + (int) (100 * Math.random()));
                        }
                } catch(InterruptedException interrupted) {
                        System.err.println("Consumer" + id + " interrupted");
                }
        }
}
```

Ciascun thread ha il compito di *consumare* 5 messaggi, prodotti da un producer.

`Example02.java`

```
package it.unipr.informatica.examples;

import it.unipr.informatica.concurrent.BlockingQeueue;

public class Example02 {
        private void go() {
                BlockingQueue<String> queue = new LinkedBlockingQueue<>();
                // costruisce e attiva i 5 consumer
                for (int i=0; i<5; ++i) {
                        Consumer consumer = new Consumer(i, queue);
                        new Thread(consumer).start();
                }
                // costruisce e attiva i 5 producer
                for (int i=0; i<5; ++i) {
                        Producer producer = new Producer(i, queue);
                        new Thread(producer).start();
                }
        }

        public static void main(String[] args) {
                new Example02().go();
        }
}
```

Avvia i 5 consumer e 5 producer, per un totale di 25 stampe su std:out.
Notare che i messaggi vengono stampati non per ordine, ma in confusione.

Cosa succede se ci sono più *consumer* che *producer*?

Se modificassimo il codice del `Consumer.java` in modo che prenda invece che 5, 6 messaggi da consumare

```
// ...
public void go() {
        try {
                for(int i=0; i < 6; ++i) {
                        String message = queue.take();
                        // ...
                }
        }
}
```

allora la nostra JVM non terminerà mai, siccome non ci saranno mai a disposizione i messaggi a sufficienza per i consumer, che rimarranno in attesa indefinitivamente.