21a.
Condition variable e code bloccanti

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

Le condition variable (variabili di condizione)
sono un classico meccanismo di sincronizzazione,
che consente ad un thread di attendere una **condizione arbitraria**,
che altri thread renderanno vera

Le *condition variable*

- Supponiamo che due thread condividano una variabile intera e che il primo thread debba **aspettare che il secondo ne modifichi il valore** per poter andare avanti

- La soluzione “ingenua” consiste nell'utilizzare una ulteriore variabile condivisa, di tipo booleano, e un ciclo del tipo:

```
while (!modified) { /* ciclo vuoto */ }
```

- Questa soluzione prende il nome di *attesa attiva*, perché durante l'attesa il thread occupa inutilmente la CPU, controllando costantemente la condizione di uscita dal ciclo

Le *condition variable*

- La soluzione corretta, invece, consiste nel realizzare *attesa passiva*, utilizzando una condition variable

- Una condition variable è un meccanismo di sincronizzazione che, unito ad un mutex, permette di attendere il verificarsi di una condizione tramite attesa passiva e senza rischi di *race condition*

- Si consulti un testo di sistemi operativi per ulteriori dettagli
- In questa lezione, le condition variable saranno presentate nella loro versione Java

Le condition variable in Java

- Come i mutex, anche le condition variable sono realizzate in Java in modo implicito, ovvero senza utilizzare esplicitamente oggetti di tipo “condition variable”
- In particolare, la loro funzionalità viene offerta dai seguenti metodi della classe Object

```
public void wait() throws InterruptedException
public void notify()
public void notifyAll()
```

- Intuitivamente, il metodo wait, chiamato su un oggetto “x”, mette il thread corrente in attesa che qualche altro thread chiami notify o notifyAll sull'oggetto “x”
- Quindi, l'oggetto “x” funge da tramite per permettere al secondo thread di comunicare al primo che può andare avanti nelle sue operazioni

- Come tutti i metodi bloccanti, wait è sensibile allo stato di interruzione del thread, e in caso di interruzione solleva l'eccezione verificata I.E.

- La differenza tra notify e notifyAll è che il primo risveglia uno solo dei thread che sono potenzialmente in attesa della condizione, mentre notifyAll li sveglia tutti

- Le prossime slide chiariscono e approfondiscono questi concetti

Funzionamento interno di wait

- Tutti e tre i metodi in esame possono essere chiamati su un oggetto "x" solo se il thread corrente possiede il monitor di "x"
  - in caso contrario, i metodi lanceranno un'eccezione a runtime

- Vediamo passo per passo il funzionamento interno di una chiamata del tipo "**x.wait**()":

1) Se il thread corrente non possiede il monitor di x, lancia un'eccezione
2) Se lo stato di interruzione del thread corrente è *vero*, lancia un'eccezione
3) In un'unica operazione atomica:
   a) mette il thread corrente nella lista di attesa di x
   b) rilascia il monitor di x
   c) sospende l'esecuzione del thread

Se il thread viene risvegliato da notify(All) o da un *risveglio spurio* (spiegato più avanti):

4) Riacquisisce il mutex di x
5) Restituisce il controllo al chiamante

Se invece il thread viene interrotto:

4) Riacquisisce il mutex di x
5) Lancia I.E.

Osservazioni e il funzionamento di notify(All)

- Riguardo la slide precedente, si osservi che al passo 4 il thread che ha invocato wait tenta di riacquisire il monitor dell'oggetto “x”, ma non è detto che ci riesca subito

- Quindi, il thread resta bloccato al passo 4 finché il monitor non si rende disponibile

Funzionamento di notify(All)

Funzionamento interno di x.notify(All):

1) Se il thread corrente non possiede il monitor di x, lancia un'eccezione

2) Se si tratta di notify:

preleva un thread dalla coda di attesa di “x” e lo rende nuovamente eseguibile (informalmente, diremo che lo “sveglia”)

Se invece si tratta di notifyAll:

preleva *tutti* i thread dalla coda di attesa di “x” e li rende nuovamente eseguibili

3) Restituisce il controllo al chiamante

Produttori e Consumatori

Pattern architetturale produttori-consumatori

- Una situazione ricorrente nella programmazione concorrente consiste nel paradigma **produttore-consumatore**

- Si tratta di due o più thread, divisi in due categorie:
  - I produttori (*producer*) sono fonti di informazioni destinate ai consumatori
  - I consumatori (*consumer*) devono elaborare le informazioni fornite dai produttori, non appena queste si rendono disponibili

- Non è possibile prevedere quanto tempo impiega un produttore a produrre un'informazione, né quanto impiega un consumatore ad elaborarla

- Quindi, è opportuno prevedere uno o più *buffer*, che contengono le informazioni prodotte e non ancora consumate

Scenario concreto produttori-consumatori

- Supponiamo di voler realizzare un **web server**

- Il server riceve un flusso di richieste con cadenza variabile e imprevedibile
- Il server ha una capacità massima di richieste che può soddisfare al secondo

Architettura produttore-consumatore:

- Il server crea una **coda di richieste** da evadere, inizialmente vuota
- Il server avvia:
  - n thread *produttori*, che accettano le richieste dai client e le mettono in coda
  - k thread *consumatori*, che prelevano le richieste dalla coda e inviano il contenuto HTML al client
- n e k vanno scelti in base alle caratteristiche hardware del server
- Tipicamente, n << k, perché il compito del produttore è molto più semplice di quello del consumatore

Queues exist to smooth out variations.

P.K. Janert, Feedback Control for Computer Systems

Condizioni di attesa

- Supponiamo che ci sia un unico buffer, con una capienza limitata
- Se un **produttore** trova il **buffer pieno** quando è pronto a produrre una nuova informazione, deve **attendere** che un consumatore liberi un posto nel buffer
- Simmetricamente, se un **consumatore** trova il **buffer vuoto**, deve **attendere** che un produttore vi inserisca almeno un'informazione
- Le condition variable permettono di realizzare queste attese in modo passivo e senza rischio di race condition
- Nella prossima slide, vediamo lo schema di un'implementazione in Java di questa situazione

- Supponiamo che il riferimento "buf" punti ad una struttura dati con metodi:
  - **put**: aggiunge un elemento
  - **take**: rimuove un elemento

- I due thread seguenti usano il buffer non solo per comunicare, ma anche per sincronizzarsi

Produttore:

```
synchronized (buf) {
   // attende che il buffer non sia pieno
   while (buf.isFull()) {
      try {
         buf.wait();
      } catch (InterruptedException e) {
         return;
      }
   }
   buf.put(some_value);
   // notifica i consumatori
   buf.notifyAll();
}
```

Consumatore:

```
synchronized (buf) {
   // attende che il buffer non sia vuoto
   while (buf.isEmpty()) {
      try {
         buf.wait();
      } catch (InterruptedException e) {
         return;
      }
   }
   some_value = buf.take();
   // notifica i produttori
   buf.notifyAll();
}
```

- Questo schema è adatto anche alla situazione con tanti produttori e tanti consumatori

Osservazioni

- Viene naturale chiedersi perché sia il produttore sia il consumatore basano la loro attesa su di un ciclo "**while**", invece di un semplice "**if**"

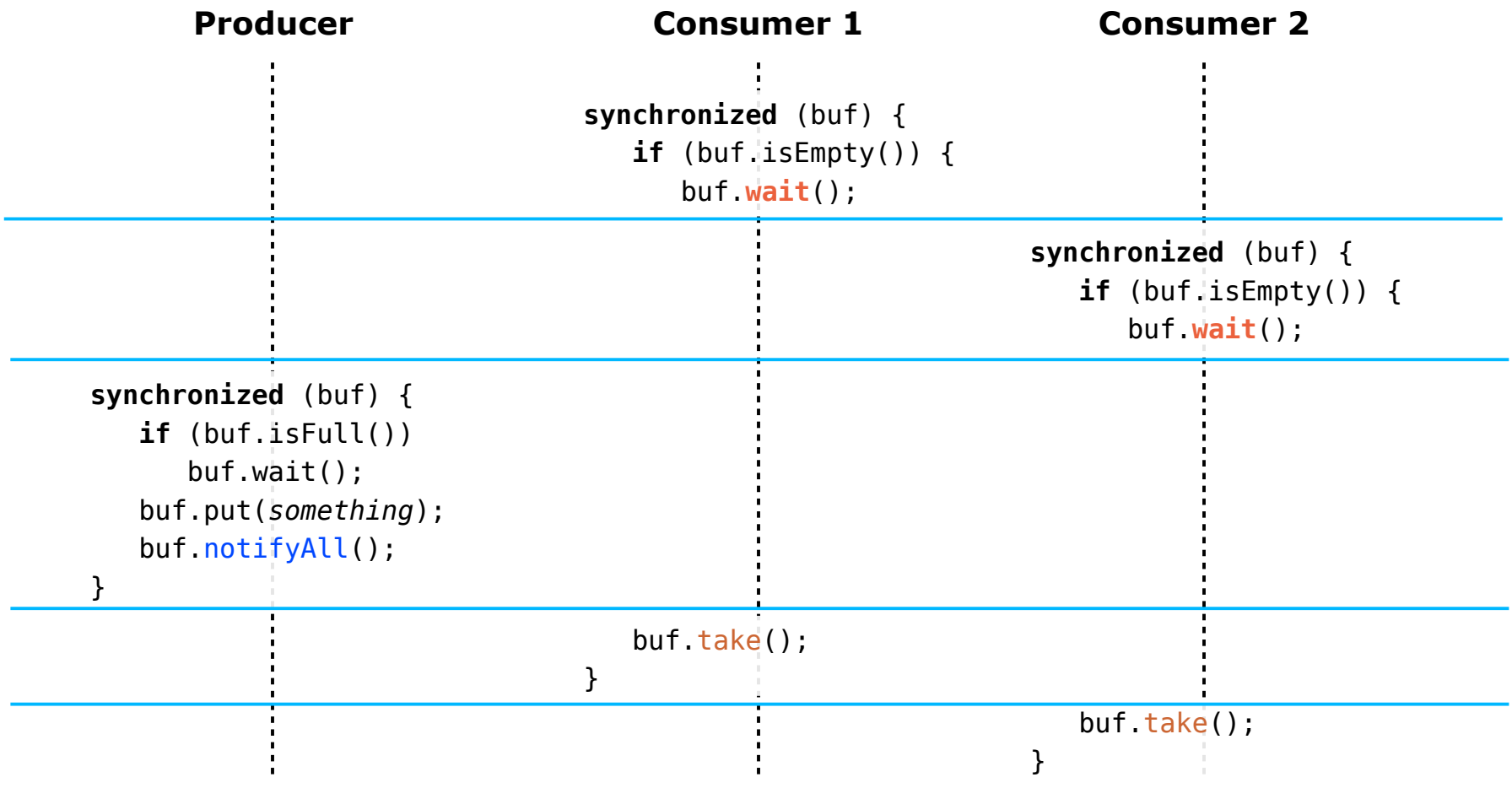
- Cosa ci sarebbe di sbagliato se il **consumatore** fosse strutturato come segue?

```
if (buf.isEmpty())
   buf.wait();
...
```

Si possono presentare **tre problemi**, illustrati nelle prossime slide:

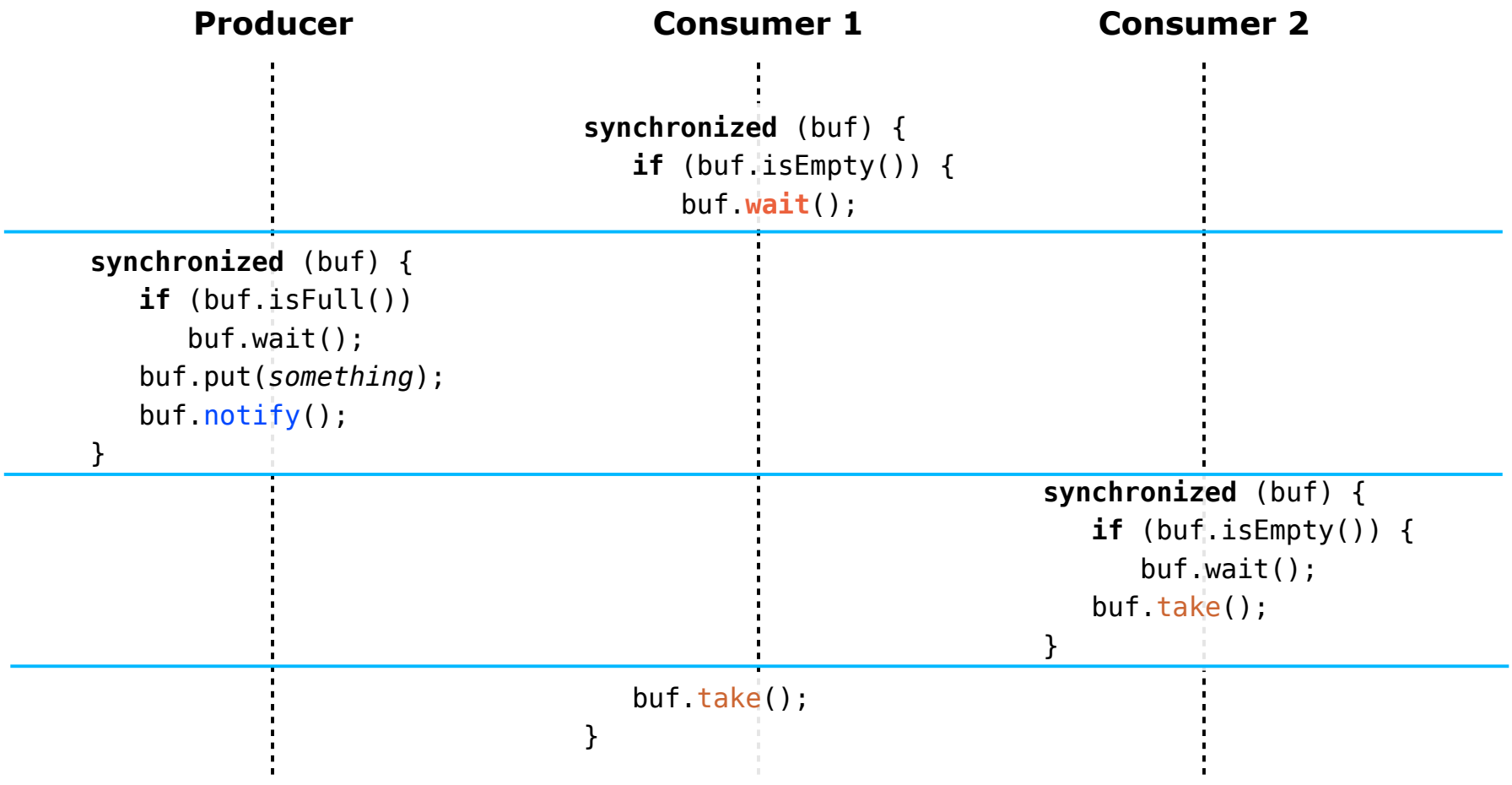
1) Se il produttore usa notifyAll: vengono svegliati due consumatori, ma c'è un solo valore nel buffer
2) Se il produttore usa notify: viene svegliato un solo consumatore, ma un altro lo anticipa
3) In tutti i casi: un *risveglio spurio* da wait può portare a leggere da un buffer vuoto

Problema 1: Consumatori multipli e notifyAll



Risultato: take da buffer vuoto!

Problema 2: Consumatori multipli e notify



Risultato: take da buffer vuoto!

Problema 2: Spurious wake-ups

- In casi eccezionali, il metodo wait può restituire il controllo al chiamante anche se non sono stati invocati i metodi notify/notifyAll
- In tal caso, si parla di uno *spurious wake-up* (risveglio spurio)

- Questa eventualità è stata prevista per compatibilità con alcuni sistemi operativi (come Linux), nei quali le system call che la JVM utilizza per implementare le condition variable vengono interrotte in caso di segnali

- Quindi, un consumatore che abbia trovato il buffer vuoto può uscire da wait senza un motivo specifico e tentare di leggere dal buffer vuoto

- Questo problema si può risolvere soltanto **ricontrollando la condizione di attesa**, quindi usando un ciclo

Perché notifyAll?

- Sorprendentemente, conviene usare notifyAll *anche se ogni prodotto è destinato ad un unico consumatore*
- Usare **notify può creare un deadlock**, come mostrato nella prossima slide

- Schematizziamo produttore e consumatore come segue:

Produttore:

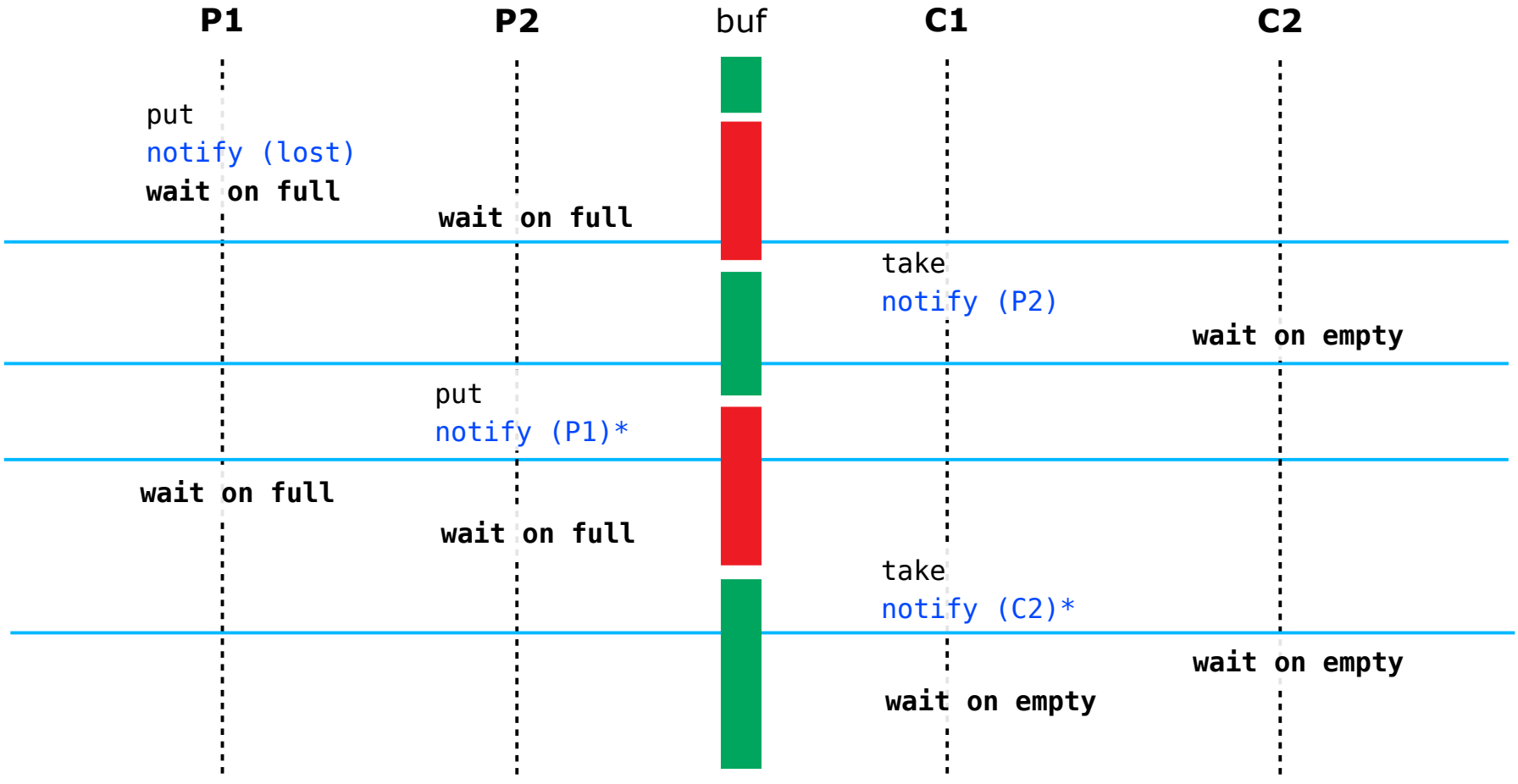
```
while (true) {
   wait on full
   put
   notify
}
```

Consumatore:

```
while (true) {
   wait on empty
   take
   notify
}
```

- Nella prossima slide, uno scenario con un buffer di capienza 1, condiviso tra due produttori e due consumatori
- Ogni notify è annotata con il thread che viene svegliato

Deadlock dovuto a notify



Risultato: deadlock!

vuoto

pieno

Esercizio 1: VoteBox

[7/2/2011, #1]

Si implementi la classe **VoteBox**, che rappresenta un'**urna elettorale**, tramite la quale diversi thread possono votare tra due alternative, rappresentate dai due valori booleani.

Il **costruttore** accetta il numero totale *n* di thread aventi diritto al voto.
La votazione termina quando *n* thread diversi hanno votato. In caso di pareggio, vince il valore *false*.

Metodi:

- Il metodo ***vote***, con parametro boolean e nessun valore di ritorno, permette ad un thread di votare, e solleva un'eccezione se lo stesso thread tenta di votare più di una volta.
- Il metodo ***waitForResult***, senza argomenti e con valore di ritorno booleano, restituisce il risultato della votazione, mettendo il thread corrente in attesa se la votazione non è ancora terminata.
- Infine, il metodo ***isDone*** restituisce true se la votazione è terminata, e false altrimenti.

E' necessario evitare attesa attiva e *race condition*.

[Caso d'uso sulla slide successiva]

Esempio d'uso:

```
VoteBox b = new VoteBox(10);
b.vote(true);
System.out.println(b.isDone());
b.vote(false);
```

Output dell'esempio d'uso:

```
false
Exception in thread "main"...
```

Code bloccanti

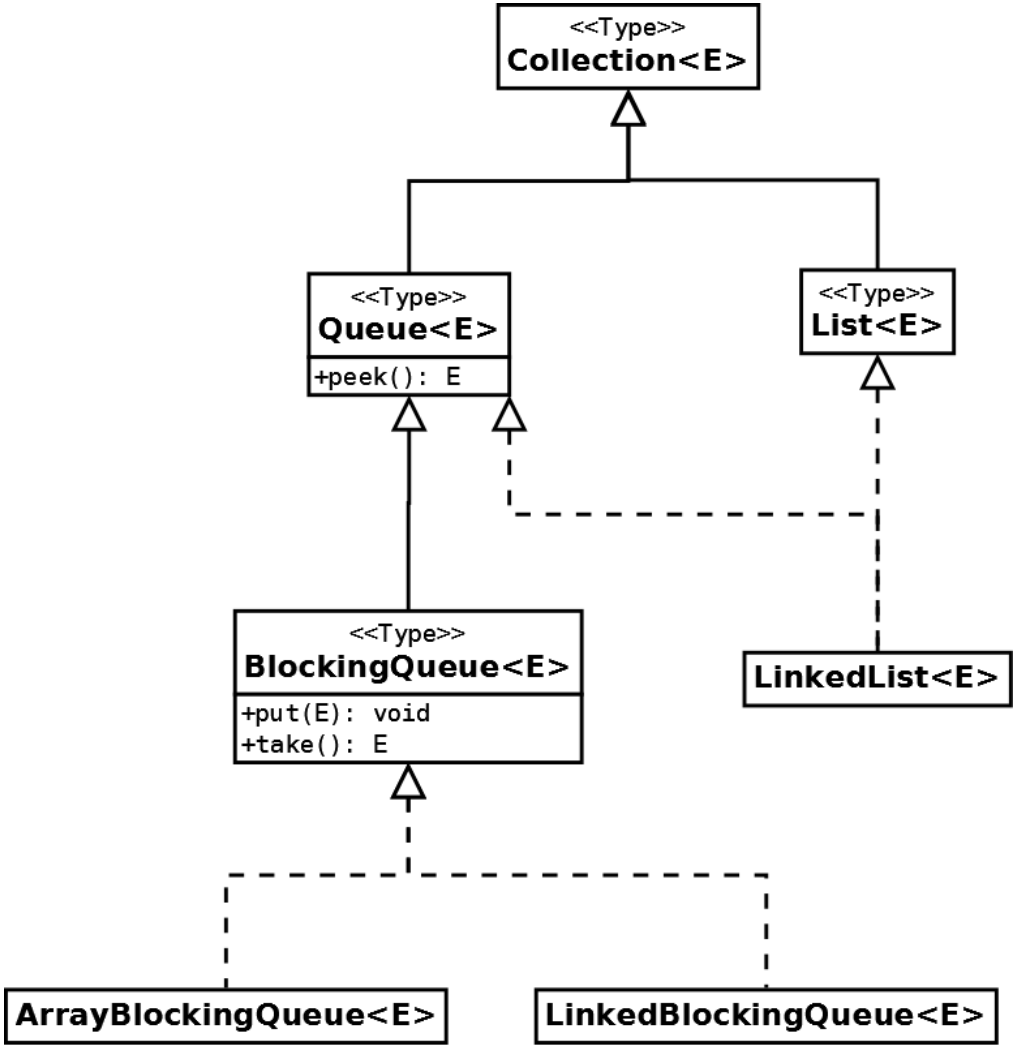


Figura 1: Le principali interfacce e classi relative alle code bloccanti.

Code bloccanti

- Situazioni simili al produttore-consumatore sono così frequenti che la libreria standard Java offre all'interno del Java Collection Framework delle collezioni predisposte all'utilizzo come buffer in un ambiente concorrente
- Si tratta delle cosiddette *code bloccanti*

- Nella Figura 1, vediamo le principali interfacce e classi relative alle code bloccanti, nonché il loro rapporto con alcune interfacce e classi che abbiamo già esaminato in riferimento al Java Collection Framework

- Si parte dall'interfaccia parametrica **Queue**, che rappresenta una generica coda, non necessariamente bloccante
- Queue viene estesa da **BlockingQueue**, che rappresenta una generica coda bloccante
- Vengono fornite diverse implementazioni concrete di BlockingQueue, delle quali noi esamineremo **ArrayBlockingQueue** e **LinkedBlockingQueue**

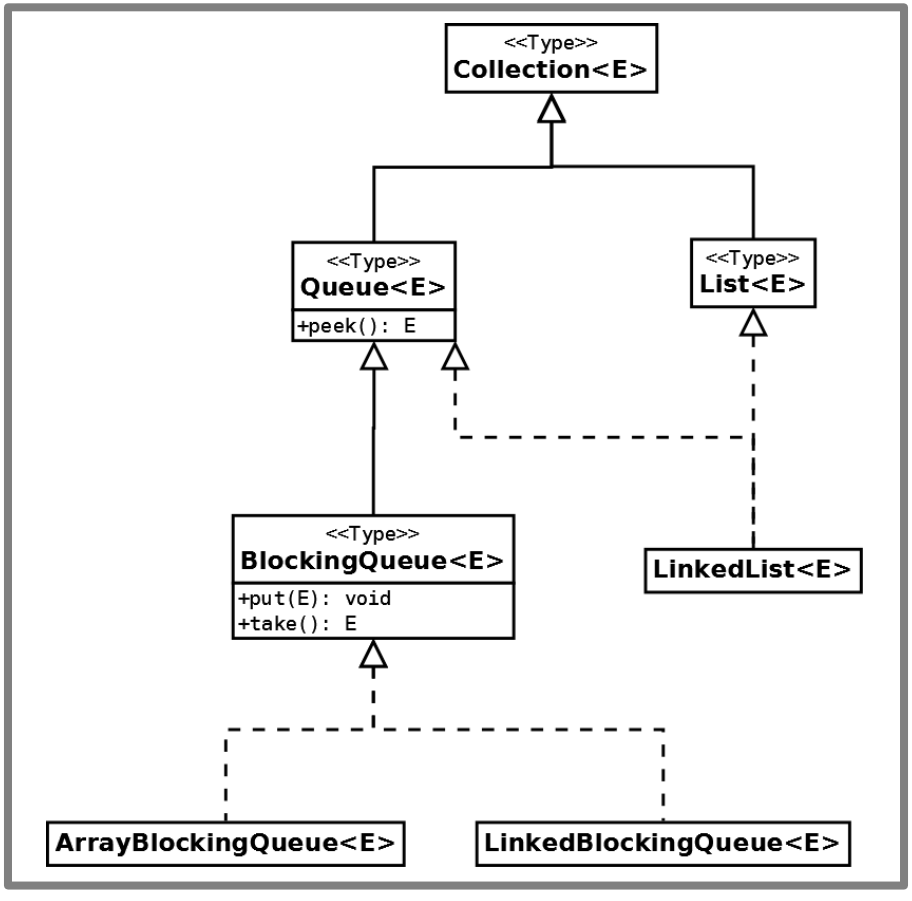


Figura 1: Le principali interfacce e classi relative alle code bloccanti.

L'interfaccia Queue

- L'interfaccia **Queue** aggiunge alcuni metodi a Collection, tra i quali menzioniamo solamente *peek:*

```
public E peek()
```

- Il metodo peek restituisce l'elemento in cima alla coda, senza rimuoverlo
- Se la coda è vuota, viene restituito *null*
- Questo metodo *non* è bloccante

- Siccome Queue estende Collection, tutte le code dispongono dei metodi offerti da Collection, quali *add*, *contains* e *remove*
- Inoltre, Queue estende indirettamente Iterable
- Si noti che LinkedList implementa, oltre a List, anche Queue, ma non BlockingQueue

L'interfaccia BlockingQueue

- L'interfaccia parametrica **BlockingQueue** rappresenta una generica coda bloccante
- Essa offre dei metodi per inserire e rimuovere elementi, che *si bloccano se la coda è piena o vuota*, rispettivamente
- Così facendo, essa permette a produttori e consumatori di sincronizzarsi senza usare esplicitamente le apposite primitive (mutex e condition variable)
- Nella prossima slide, esamineremo i due metodi principali di BlockingQueue

Metodi delle code bloccanti

- L'interfaccia **BlockingQueue** offre, tra gli altri, i metodi *put* e *take*:

```
public void put(E elem) throws InterruptedException
```

- Inserisce l'oggetto *elem* nella coda
- Le implementazioni scelgono in che posto inserirlo, tipicamente all'ultimo posto (ordine FIFO)
- Se la coda è **piena**, questo metodo mette il thread corrente in **attesa** che venga rimosso qualche elemento dalla coda
- Come tutti i metodi **bloccanti**, put è sensibile allo stato di interruzione del thread corrente, e solleva l'eccezione I.E. se tale stato diventa vero durante l'attesa, o era già vero all'inizio dell'attesa

```
public E take() throws InterruptedException
```

- Restituisce e rimuove l'elemento in cima alla coda
- Se la coda è **vuota**, questo metodo mette in **attesa** in thread corrente, finché non viene inserito almeno un elemento nella coda
- Vale per *take* lo stesso discorso fatto per *put*, in relazione allo stato di interruzione dei thread

Implementazioni di BlockingQueue

- La classe **ArrayBlockingQueue** (in breve, ABQ) è un'implementazione di BlockingQueue, realizzata internamente tramite un array circolare
- Una ABQ ha una **capacità fissa**, dichiarata una volta per tutte al costruttore:

```
public ArrayBlockingQueue(int capacity)
```

- Una ABQ rappresenta un classico buffer con capacità limitata (*bounded buffer*)

- La classe **LinkedBlockingQueue** (in breve, LBQ) è un'altra implementazione di BlockingQueue, realizzata tramite una lista concatenata
- Una LBQ ha una **capacità** potenzialmente **illimitata**
- Quindi, una LBQ non risulta **mai piena**
- Pertanto, il metodo put di LBQ *non* è bloccante

- Oltre ad essere bloccanti, queste classi sono anche *thread-safe*
- Lo stesso non si può dire per le altre classi del Java Collection Framework, come LinkedList o HashSet, che possono dare risultati inattesi se utilizzate concorrentemente da più thread senza le opportune precauzioni di sincronizzazione

- I metodi put e take di queste due classi rispettano l'ordine FIFO

Produttore-consumatore con coda bloccante

- Le code bloccanti permettono di implementare in modo molto semplice lo schema produttore-consumatore, senza doversi occupare manualmente della sincronizzazione

- I seguenti thread condividono il buffer

```
BlockingQueue<T> buffer = new ArrayBlockingQueue<T>(capacity);
```

Produttore:

```
T x;
// "produce" x
try {
   // attende se il buffer è pieno
   buffer.put(x);
} catch (InterruptedException e) {
   return;
}
```

Consumatore:

```
T x;
try {
   // attende se il buffer è vuoto
   x = buffer.take();
} catch (InterruptedException e) {
   return;
}
// "consuma" x
```

Architetture a confronto

- Supponiamo di voler realizzare un **web server**

- Il server riceve un flusso di richieste con cadenza variabile e imprevedibile
- Il server ha una capacità massima di richieste che può soddisfare al secondo

- Si confrontino le seguenti architetture, producendo un grafico qualitativo che abbia sulle ascisse il numero di richieste che arrivano al server per unità di tempo e sulle ordinate il tempo per soddisfare ciascuna richiesta

- **Architettura 1:**
  - Il server crea un thread per ogni nuova richiesta
  - Quel thread gestisce per intero la richiesta

- **Architettura 2:**
  - Il server crea un thread per ogni nuova richiesta, con un numero massimo k di thread contemporaneamente attivi

- **Architettura 3:** produttore-consumatore con buffer illimitato (es., LinkedBlockingQueue)

- **Architettura 4:** produttore-consumatore con buffer limitato di capacità k (es., ArrayBlockingQueue)

Esercizio 2

[14/9/2010, #4]

Implementare il metodo statico ***executeInParallel***, che accetta come argomenti un array di oggetti di tipo *Runnable* e un numero naturale *k*, ed esegue tutti i *Runnable* dell'array, *k* alla volta.

In altre parole, all'inizio il metodo fa partire in parallelo i primi *k Runnable* dell'array.
Poi, non appena uno dei *Runnable* in esecuzione termina, il metodo ne fa partire un altro, preso dall'array, fino ad esaurire tutto l'array.

Risolvere l'esercizio senza utilizzare attesa attiva.

Esercizio 3

[18/6/2012, #4]

Implementare il metodo statico **threadRace**, che accetta due oggetti Runnable come argomenti, li esegue contemporaneamente e restituisce 1 oppure 2, a seconda di quale dei due Runnable è terminato prima.

Implementare due versioni che differiscono per il criterio di terminazione:

1) il metodo termina quando entrambi i Runnable terminano
2) il metodo termina quando almeno un Runnable è terminato

1) Java Concurrency in Practice

di Goetz, Peierls, Bloch, Bowbeer, Holmes e Lea

Addison-Wesley

2) Il **Java Memory Model**

Disponibile in rete come *Java Specification Request (JSR) 133*, oppure come capitolo 17 del *Java Language Specification* (definizione del linguaggio Java)

3) Effective Java: a Programming Language Guide

di Joshua Bloch

Addison-Wesley

4) The JSR-133 Cookbook for Compiler Writers

pagina web curata da Doug Lea