# Lezione n.10
# LPR-B-08
# Collezioni Sincronizzate & Code Bloccanti
# 10/12/2008

Andrea Corradini
Laura Ricci

# Sommario

- Collezioni Sincronizzate
- Code Bloccanti
- Secure Sockets

# COLLEZIONI SINCRONIZZATE

- **JAVA Collections**: strutture dati predefinite incluse nel package java.util

  a partire da JAVA 1.2
- Alcuni esempi: HashTables, Vectors
- Synchronized Collections: Definiscono strutture dati thread safe, cioè garantiscono che lo stato della struttura risulti corretto anche nel caso in cui la struttura venga acceduta in modo concorrente da più threads
- Thread Safety: la struttura dati viene incapsulata e ogni metodo pubblico viene sincronizzato
- Se l'operazione che il client (il thread che utilizza la collezione) è composta da una serie di operazioni elementari, il client deve spesso implementare ulteriori sincronizzazioni.
- Si possono definire blocchi di codice sincronizzati che incapsulano più operazioni elementari effettuate su una collezione sincronizzata

# COLLEZIONI SINCRONIZZATE

```
import java.util.*;
public class ThreadRemover extends Thread {
        Vector v;
        public ThreadRemover(Vector v) {this.v=v;}
            public void run( ) {
                int lastIndex = v.size(   ) - 1;
                    Object o = v.remove(lastIndex); }
            }
```

- il thread ThreadRemover elimina da un Vector l'elemento che si trova nella sua ultima posizione
- Le operazioni size( ) e remove( ) sono definite nella classe Vector come operazioni sincronizzate

# COLLEZIONI SINCRONIZZATE

```
import java.util.Vector;
public class ThreadGet extends Thread{
      Vector v;
      public ThreadGet(Vector v){this.v = v;}
            public void run(  ){
                  int lastIndex = v.size( ) -1;
                  try {Thread.sleep(5000);}  catch(InterruptedException e){  };
                  v.get(lastIndex);   } }
```

- Il thread ThreadGet restituisce l'ultimo elemento del Vector
- La sleep( ) è stata introdotta per costringere ThreadGet a rilasciare il processore a ThreadRemover
- In questo modo si ottiene un interleaving scorretto tra i due threads ed il programma segnala la seguente eccezione

# COLLEZIONI SINCRONIZZATE

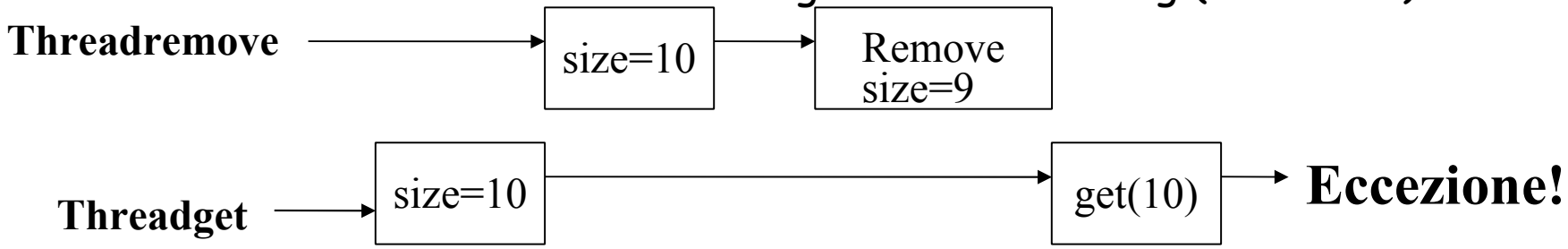
Eccezione sollevata dalla esecuzione del programma:

Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException:
    Array index out of range: 9

at java.util.Vector.get(Unknown Source)

at ThreadGet.run(ThreadGet.java:10)

L'eccezione viene sollevata a causa del seguente interleaving (scorretto)



# COLLEZIONI SINCRONIZZATE

Versione corretta di ThreadGet

```
import java.util.Vector;
public class ThreadGet extends Thread {
    Vector v;
    public ThreadGet(Vector v){this.v = v;}
    public void run ( ){
        synchronized(v) {
            int lastIndex = v.size( ) -1;
            Object x=v.get(lastIndex);
            System.out.println("oggetto prelevato"+x);
            }}}
```

# COLLEZIONI SINCRONIZZATE

Versione corretta di ThreadRemover

```
import java.util.*;
public class ThreadRemover extends Thread{
      Vector v;
      public ThreadRemover(Vector v){this.v=v;}
      public void run( )
            {
            synchronized(v) {
                  int lastIndex = v.size( )-1;
                  Object o=v.remove(lastIndex);} }
            }
```

# BLOCCHI SYNCHRONIZED E METODI SYNCHRONIZED

```
public class Coda { //....
    public synchronized boolean isSpaceAvailable( ) { //....
    public synchronized void add(Item i){ //....
    public synchronized Item remove( ){ //....
    ......
    Coda c =//....
        syncronized (c)
                { if (c.isSpaceAvailable()) {
                        c.add(i); }}
```

- Se un thread entra nel blocco sincronizzato acquisisce la lock su c. Quando entra in un metodo sincronizzato, possiede già la lock e non la deve quindi richiedere nuovamente.

# CODE BLOCCANTI (O SINCRONIZZATE)

- Code bloccanti: introdotte in JAVA 5 come supporto per il paradigmi computazionali di tipo produttore/consumatore
- Sono code sincronizzate. Comportamento di base:
  - inserimento: aggiunge un elemento in fondo alla coda, se la coda non è piena, altrimenti blocca il thread che ha invocato l'operazione.
  - rimozione: elimina il primo elemento della coda, se questo esiste, altrimenti blocca il thread che ha invocato l'operazione
- L'interfaccia java.util.concurrent.BlockingQueue definisce questo tipo di code
- Sono stati definiti
  - diverse varianti della coda
  - metodi caratterizzati da diversi comportamenti per l'inserzione e la rimozione di elementi dalla coda

# CODE BLOCCANTI: TIPI DEFINITI

Classi che implementano l'interfaccia BlockingQueue

- LinkedBlockingQueue: non si definisce un limite superiore alla capacità della coda
- ArrayBlockingQueue: definisce un numero fisso di posizioni della coda
- SynchronousQueue: non è una vera e propria coda, in quanto non ha capacità di memorizzazione. Mantiene solo le code per la gestione dei threads che aspettano in attesa di produrre/consumare elementi. Risparmia il tempo necessario per la bufferizzazione degli elementi prodotti
- PriorityBlockingQueue: implementa una coda con priorità

# CODE BLOCCANTI: METODI

Metodi che generano un'eccezione quando si tenta di aggiungere un elemento ad una coda piena o di estrarre un elemento da una coda vuota:

add, remove, element (element restituisce l'elemento in testa, e non lo rimuove)

```
import java.util.concurrent.*;
public class ProvaQueue {
    public static void main (String args[]){
        BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
        queue.add("el1"); queue.add("el2");   }}
```

```
Exception in thread "main" java.lang.IllegalStateException: Queue full
at java.util.AbstractQueue.add(Unknown Source)
at java.util.concurrent.ArrayBlockingQueue.add(Unknown Source)
at ProvaQueue.main(ProvaQueue.java:7)
```

# CODE BLOCCANTI: METODI

Metodi che restituiscono una segnalazione del fallimento dell'operazione di inserzione/estrazione: offer, poll, peek (anche con timeout)

```
import java.util.concurrent.*;
public class ProvaQueue1 {
public static void main (String args[]){
    BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
    boolean success = false;
    success = queue.offer("el1");  System.out.println(success);
    try{success = queue.offer("el2", 100, TimeUnit.MILLISECONDS);
    }catch (InterruptedException e){ }
    System.out.println(success);}}
```

Ouput del programma  true,  false

# CODE BLOCCANTI:METODI

Metodi che bloccano il thread nel caso del fallimento della operazione di inserzione/rimozione di un elemento

put, take

```
import java.util.concurrent.*;
public class ProvaQueue2 {
public static void main (String args[ ]){
BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
try  {queue.put("el1");} catch (InterruptedException e){ }
try{ queue.put("el2");} catch (InterruptedException e){ }
      }}
```

la seconda put blocca il thread. Il thread viene risvegliato quando l'inserzione/rimozione può essere effettuata

# ESERCIZIO

Si vuole realizzare un programma Crawler che analizza tutti i file presenti in una directory specificata e nelle sue sottodirectory e visualzza tutte le righe presenti in tali file che contengono una determinata parola chiave P.

Il programma deve attivare due thread,

- un thread produttore che enumera tutti i file e inserisce un riferimento ad ogni file individuato in una coda bloccante
- un consumatore che estrae i riferimenti ai file dalla coda e, per ognuno di essi, visualizza tutte le righe che contengono la parola chiave P.

I due thread si scambiano i dati mediante una coda bloccante. Scegliere il tipo di coda bloccante ritenuto più opportuno.