Lezione n.10
LPR A
RMI CallBacks
Threads Miscellanea

1/12/2008

Laura Ricci

RIASSUNTO DELLA PRESENTAZIONE

- RMI: il meccanismo delle callbacks
- Thread Miscellanea:
  - Thread Pool: politiche di saturazione
  - Blocchi Sincronizzati
  - Collezioni Sincronizzate
  - Code Bloccanti

RMI: IL MECCANISMO DELLE CALLBACK

RMI utilizza una comunicazione

- sincrona a rendez vouz esteso: il client invoca un metodo remoto e si blocca finchè il metodo non termina

Molte applicazioni fanno riferimento an pattern di interazione asincrono, in cui

- il client è interessato ad un evento che si verifica sul server e notifica il suo interesse al server (ad esempio utilizzando RMI)
- il server registra che il client è interessato in quell'evento
- il client prosegue la sua elaborazione dopo la notifica al server
- quando l'evento si verifica, il server lo notifica ai clients interessati l'accadimento dell'evento

RMI : IL MECCANISMO DELLE CALLBACK

Esempi di applicazioni:

- un utente partecipa ad una un gruppo di discussione (es: Facebook) e vuol essere avvertito quando un nuovo utente entra nel gruppo.

- lo stato di un gioco multiplayer viene gestito da un server. I giocatori notificano al server le modifiche allo stato del gioco. Ogni giocatore deve essere avvertito quando lo stato del gioco subisce delle modifiche.

- gestione distribuita di un'asta: un insieme di utenti partecipa ad un'asta distribuita. Ogni volta che un utente fa una nuova offerta, tutti i partecipanti all'asta devono essere avvertiti

RMI: IL MECCANISMO DELLE CALLBACK

Soluzioni possibili per realizzare nel server un servizio di notifica di eventi al client:

- Polling: il client interroga ripetutamente il server, per verificare se l'evento atteso si è verificato. L'interrogazione avviene mediante l'invocazione di un metodo remoto (mediante RMI).
  - Svantaggio: alto costo per l'uso non efficiente delle risorse del sistema

- Registrazione dei client interessati agli eventi e successiva notifica (asincrona) del verificarsi dell'evento ai client da parte del server
  - Vantaggio: il client può proseguire la sua elaborazione senza bloccarsi ed essere avvertito, in modo asincrono, quando l'evento si verifica
  - Problema: quale meccanismo utilizza il server per risvegliare il client?

RMI: IL MECCANISMO DELLE CALLBACK

- E' possibile utilizzare RMI sia per l'invocazione client-server (registrazione del client) che per quella server-client (notifica del verificarsi di un evento) utilizzando il meccanismo delle callback
- Il server definisce una interfaccia remota ServerInterface che include un metodo remoto che può essere utilizzato dal client per registrarsi
- Il client definisce una interfaccia remota ClientInterface che definisce un metodo remoto utilizzato dal server per notificare un evento al client
- Il client ha a disposizione la ServerInterface e reperisce il puntatore all'oggetto remoto tramite il registry
- Il server ha a disposizione la ClientInterface e riceve al momento della registrazione del client il riferimento all'oggetto remoto sul client

RMI: IL MECCANISMO DELLE CALLBACK

- Il client, al momento della registrazione sul server, passa al server un riferimento RC ad un oggetto che implementa la ClientInterface
- Il server memorizza RC in una sua struttura dati (ad esempio, un vector)
- al momento della notifica, il server utilizza RC per invocare il metodo remoto di notifica definito dal client.
- In questo modo rendo 'simmetrico' il meccanismo di RMI, ma…

  il client non registra l'oggetto remoto in un rmiregistry, ma passa un riferimento a tale oggetto al server, al momento della registrazione

RMI CALLBACKS

Un client può richiedere servizi ad un servente mediante RMI. Talvolta è utile poter consentire al servente di contattare il client

- il servente notifica degli eventi ai propri client per evitare il polling effettuato dai clients
- il servente accede allo stato della sessione se questo è memorizzato presso il client meccanismo, analogo a quello dei cookie

La soluzione è offerta dal meccanismo delle callback in cui il servente può richiamare il client

In questo modo il meccanismo di RMI viene utilizzati in modo bidirezionale

- dal client al server (oggetto reperito via registry)
- Dal server al client (oggetto reperito mediante passaggio di parametri

RMI CALLBACKS

- Il client crea un oggetto remoto, oggetto callback *OC*, che implementa un'interfaccia remota che deve essere nota al servente
- Il servente definisce un oggetto remoto *OS*, che implementa una interfaccia remota che deve essere nota al client
- Il client reperisce *OS* mediante il meccanismo di lookup di un registry
- *OS* contiene un metodo che consente al client di registrare il proprio *OC* presso il server
- quando il servente ne ha bisogno, può contattare ***OC***, reperendo un riferimento ad *OC* dalla struttura dati in cui lo ha memorizzato al momento della registrazione

CALLBACKS: UN ESEMPIO

Server:

- Definisce un oggetto remoto che fornisce ai clients metodi per
  - Registrare/cancellare una callback
  - Contattare il server (metodo SayHello)

Client

- Registra una callback presso il server. La callback consente al server di notificare ai clients registrati ogni contatto stabilito dai clients mediante il metodo SayHello
- Effettua un numero casuale di richieste del metodo sayHello
- Cancella la propria registrazione

L'INTERFACCIA DEL CLIENT

```
import java.rmi.*;
public interface CallbackHelloClientInterface extends Remote {
    /* Metodo invocato dal  server per effettuare
     una callback a un client remoto. */
    public void  notifyMe(String message) throws RemoteException;
    { ...  }
```

notifyMe(...)

è il metodo esportato dal client e che viene utilizzato dal server per la notifica di un nuovo contatto da parte di un qualsiasi client. Viene notificato il nome del client che ha contattato il server.

L'INTERFACCIA DEL CLIENT:IMPLEMENTAZIONE

```
import java.rmi.*; import java.rmi.server.*;
public class CallbackHelloClientImpl  implements CallbackHelloClientInterface {
                /* crea un nuovo callback client */
                public CallbackHelloClientImpl( ) throws RemoteException
                    { super( ); }
               /* metodo che può essere richiamato dal servente */
                public void notifyMe(String message) throws RemoteException {
                   String returnMessage = "Call back received: " + message;
                   System.out.println( returnMessage); }
                }
```

ATTIVAZIONE DEL CLIENT

```
import java.rmi.*;
public class CallbackHelloClient {
public static void main(String args[ ]) {
      try {
                  vedi lucido successivo.....

        } catch (Exception e) {
           System.err.println("HelloClient exception: " +e.getMessage(  )); }
                                                  }
                                               }
```

IL CODICE DEL CLIENT

```
System.out.println("Cerco CallbackHelloServer");
Registry registry = LocateRegistry.getRegistry("localhost", 2048);
String name = "CallbackHelloServer";
CallbackHelloServerInterface h =
        (CallbackHelloServerInterface) registry.lookup(name);
/* si registra per il callback */
System.out.println("Registering for callback");
CallbackHelloClientImpl callbackObj = new CallbackHelloClientImpl( );
CallbackHelloClientInterface stub = (CallbackHelloClientInterface)
                    UnicastRemoteObject.exportObject(callbackObj, 0);
h.registerForCallback(stub);
```

IL CODICE DEL CLIENT

```
/* accesso al server - fa una serie casuale di 5-15 richieste */
int n = (int) (Math.random( )*10+5);
String nickname=  "mynick";
for (int i=0; i< n; i++) {
        String message = h.sayHello(nickname);
        System.out.println( message);
        Thread.sleep(1500);};
/* cancella la registrazione per il callback */
System.out.println("Unregistering for callback");
h.unregisterForCallback(callbackObj);
```

L'INTERFACCIA DEL SERVER

```
import java.rmi.*;
public interface CallbackHelloServerInterface extends Remote {
    /* metodo di notifica */
    public String sayHello(String name) throws RemoteException;
    /* registrazione per il callback */
    public void registerForCallback(CallbackHelloClientInterface callbackClient)
    throws RemoteException;
    /* cancella registrazione per il callback */
    public void unregisterForCallback(CallbackHelloClientInterface callbackClient)
    throws RemoteException;
```

L'IMPLEMENTAZIONE DEL SERVER

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class CallbackHelloServerImpl implements CallbackHelloServerInterface
        /* lista dei client registrati */
        private List<CallbackHelloClientInterface> clients;
        /* crea un nuovo servente */
        public CallbackHelloServerImpl( ) throws RemoteException {
                super( );
                clients = new ArrayList<CallbackHelloClientInterface>( );
                                                                        }
```

L'IMPLEMENTAZIONE DEL SERVER

```
public synchronized void registerForCallback
        (CallbackHelloClientInterface callbackClient) throws RemoteException
{ if (!clients.contains(callbackClient)) { clients.add(callbackClient)
                System.out.println(" New client registered." );   }
/* annulla registrazione per il callback */
public synchronized void unregisterForCallback
      ( CallbackHelloClientInterface callbackClient)   throws RemoteException
{if (clients.remove(callbackClient)) {System.out.println("Client unregistered");}
else { System.out.println("Unable to unregister client."); }
}  }
```

L'IMPLEMENTAZIONE DEL SERVER

```
/* metodo di notifica
* quando viene richiamato, fa il callback a tutti i client registrati */
public String sayHello (String name) throws RemoteException {
                    doCallbacks(name);
                    return "Hello, " + name + "!";
                                                    }
```

IL SERVER:IMPLEMENTAZIONE

```
private synchronized void doCallbacks(String name ) throws RemoteException
{    System.out.println("Starting callbacks.");
     Iterator i = clients.iterator( );
     int numeroClienti = clients.size( );
     while (i.hasNext()) {

          CallbackHelloClientInterface client =

                              (CallbackHelloClientInterface) i.next();

          client.notifyMe(name);
                              }
     System.out.println("Callbacks complete.");}      }
```

IL CODICE DEL SERVER

```
import java.rmi.server.*; import java.rmi.registry.*;
public class CallbackServer   {
public static void main(String[  ] args) {
try { /* registrazione presso il registry */
      System.out.println("Binding CallbackHello");
      CallbackHelloServerImpl server =  new CallbackHelloServerImpl(    );
      CallbackHelloServerInterface stub=(CallbackHelloServerInterface)
                  UnicastRemoteObject.exportObject (server, 39000);
       String name = "CallbackHelloServer";
      Registry registry = LocateRegistry.getRegistry ("localhost",2048);
      registry.bind  (name, stub);
      System.out.println("CallbackHello bound");
} catch (Exception e) { System.out.println("Eccezione" +e);   }}}}
```

RMI:ECCEZIONI

- Eccezione che viene sollevata se non trova un servizio di registry su quella porta. Esempio:

  HelloClient exception: Connection refused to host: 192.168.2.103; nested exception is: java.net.ConnectException: Connection refused: connect

- Eccezione sollevata e si tenta di registrare più volte lo stesso stub con lo stesso nome nello stesso registry

  Esempio

  CallbackHelloServer exception: java.rmi.AlreadyBoundException: CallbackHelloServer java.rmi.AlreadyBoundException: CallbackHelloServer

ESERCIZIO 1: CALLBACKS

In una lezione precedente è stato assegnato un esercizio per la gestione Elettronica di una elezione a cui partecipano un numero prefissato di candidati. Si chiedeva di realizzare un server RMI che consentisse al client di votare un candidato e di richiedere il numero di voti ottenuti dai candidati fino ad un certo punto.

Modificare l'esercizio in modo che il server notifichi ogni nuovo voto ricevuto a tutti i clients che hanno votato fino a quel momento. La registrazione dei clients sul server avviene nel momento del voto.

ESERCIZIO 2: GESTIONE FORUM

Si vuole implementare un sistema che implementi un servizio per la gestione di forum in rete. Un forum è caratterizzato da un argomento su cui diversi utenti possono scambiarsi opinioni via rete. Il sistema deve prevedere un server RMI che fornisca le seguenti funzionalita':

a) apertura di un nuovo forum, di cui è specificato l'argomento (esempio: giardinaggio)

b) inserimento di un nuovo messaggio indirizzato ad un forum identificato dall'argomento (es: è tempo di piantare le viole, indirizzato al forum giardinaggio)

c) reperimento dell'ultimo messaggio inviato ad un forum di cui è specificato l'argomento.

Il messaggio può essere richiesto esplicitamente dal client oppure può essere notificato ad un client precedentemente registrato

SERVER PROGRAMMING

- Server Web, Mail Server, File Server, Database Server sono caratterizzati dalla seguente struttura:

```
ServerSocket socket = new ServerSocket(80);
    while (true)
    { Socket connection = socket.accept( );
      handleRequest(connection);}
```

- La gestione di ogni servizio (handleRequest) è indipendente dalla gestione degli altri
- Esempio: la gestione di un messaggio inviato ad un mail server non dipende da quella degli altri messaggi elaborati contemporaneamente
- E' naturale associare un thread diverso alla gestione di ogni richiesta di servizio.

SERVER PROGRAMMING

```
import java.net.*; import java.util.concurrent.*; import java.io.*;
public class server_pool {
private static final int NTHREADS = 100;
private static final Executor exec=Executors.newFixedThreadPool(NTHREADS);
    public static void main(String [ ] args) throws IOException{
    ServerSocket socket = new ServerSocket(80);
    while (true) {
        final Socket connection = socket.accept();
        Runnable task = new Runnable( )
            {  public void run( )   {  handleRequest(connection); }  };
        exec.execute(task);  }   }   }
```

THREAD POOL: POLITICHE DI SATURAZIONE

- Politica di saturazione:
  - utilizzata nel caso si utilizzino thread pool con code di dimensione limitata
  - indica le azioni che devono essere effettuate nel caso in cui il pool è saturo, cioè la coda è piena ed i threads del pool sono tutti attivi
- Strategia di default: abort il task viene rifiutato e viene sollevata una RejectedExecutionException
- E' possibile definire una politica 'ad hoc' mediante un Rejected Execution Handler
- JAVA mette a disposizione diversi Rejected Execution Handler, ognuno dei quali implementa una diversa politica di saturazione
- Selezione della politica: setRejectedExecutionHandler

THREAD POOL:POLITICHE DI SATURAZIONE

Politiche di saturazione: quando viene sottomesso un task T e la coda è piena, possono essere adottate le seguenti politiche

- abort T viene scartato e viene sollevata un'eccezione
- discard policy rifiuta T, ma non solleva alcun tipo di eccezione
- discard oldest scarta il primo task della coda (quello che avrebbe dovuto essere eseguito successivamente) e inserisce T in coda
- caller-runs
  - non scarta il task e non solleva eccezioni.
  - cerca di rallentare (throttling, to throttle=strozzare) il flusso dei tasks restituendo alcuni task al chiamante per l'esecuzione. Il task non viene eseguito in un thread del pool, ma nel thread che ha invocato la execute

THREAD POOL: POLITICHE DI SATURAZIONE

```
import java.util.concurrent.*;
public class prova {
public static void main (String args[ ])
{ThreadPoolExecutor executor=new
    ThreadPoolExecutor(10,11,0L,TimeUnit.MILLISECONDS,  new
                                LinkedBlockingQueue<Runnable>(100));
    executor.setRejectedExecutionHandler    ( new
                            ThreadPoolExecutor.CallerRunsPolicy());}  }
```

THREAD POOL:POLITICHE DI SATURAZIONE

- Consideriamo un Web Server
  - riceve richieste di servizio da parte dei clients
  - ogni richiesta corrisponde ad un diverso task
- Supponiamo che il Web Server definisca un thread pool che utilizzi una coda a dimensione limitata ed una politica di saturazione caller-runs.
- Se tutti i threads sono occupati e la coda è piena, il task successivo (la successiva richiesta di servizio) viene eseguita nel thread che ha invocato la execute (il main thread del server)
- Poiché l'esecuzione del servizio richiesto richiederà un intervallo di tempo ΔT, il main thread non sottometterà ulteriori taks per l'esecuzione al pool, durante ΔT
- Alcuni thread del pool possono terminare l'esecuzione del task assegnato durante ΔT, rendendosi disponibili a nuove sottomissioni

THREAD POOL: POLITICHE DI SATURAZIONE

- Inoltre, il main thread, impiegato nell'esecuzione del task rifiutato dal pool, non accetta ulteriori connessioni (non esegue l'accept sul ServerSocket) durante l'intervallo di tempo ΔT
- Le richieste di servizio vengono quindi accodate a livello TCP
- Se l'overload persiste, sarà il livello TCP che eventualmente inizierà a scartare richieste, quando la sua coda sarà, a sua volta, satura
- Conclusione: se il server è sovraccarico, si può implementare una graceful degradation, spostando gradualmente il sovraccarico
  - dai thread del pool alla coda associata al pool,
  - dalla coda alla applicazione,
  - dall'applicazione al livello TCP
  - Il livello TCP è in grado di bloccare il client, mediante il meccanismo di controllo del flusso

BLOCCHI SINCRONIZZATI

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
    { // blocco di codice
    }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock su l'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- La lock non viene rilasciata nel caso in cui si invochi un altro metodo all'interno del blocco di codice (lock rientrante)

BLOCCHI SINCRONIZZATI

- Utilizzo:
  - ridurre la lunghezza di una sezione critica
  - rendere indivisibile un frammento di codice che invoca due o più metodi synchronized
- L'esempio presentato nei lucidi successivi mostra in modo operativo il vantaggio di diminuire la lunghezza di una sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
double A,B.....
public void setValues (int x, double r)
    double tempA= // questa elaborazione richiede molto tempo...
    double tempB= // questa elaborazione richiede molto tempo...
    synchronized ( this ) {
        A = tempA;
        B = tempB; }
```

- L'elaborazione che richiede molto tempo viene eseguita fuori dalla sezione critica ed i risultati vengono assegnati a variabili locali
- la sezione critica contiene solo l'aggiornamento delle variabili di istanza del metodo
- si riduce il tempo in cui un thread che ha acquisito la lock rimane all'interno della sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe Pair incapsula

- una coppia di valori interi (x,y)
- un valore intero accesscount.

La coppia di valori deve essere aggiornata in modo atomico e in mutua esclusione rispetto all'aggiornamento della variabile accesscount

```
public class Pair {
      private int x,y;
      private int accesscount=0;
      public Pair (int x, int y)
            {  this.x=x;   this.y=y;  };
      public synchronized void accessincrement( )
                  {accesscount++;};
      public synchronized int accessreturn( )
                  {return accesscount;};
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

Per l'aggiornamento della coppia di valori, si definiscono due diversi metodi

- increment_syn_method( ): utilizza un metodo sincronizzato
- increment_syn_block( ): utilizza un blocco sincronizzato
- Nota Bene: la sleep simula un'operazione computazionalmente costosa

```
public synchronized void increment_syn_method(  )
      {x++;  y++;
      try {Thread.sleep(100);} catch(Exception e)  {  };
       }

public void increment_syn_block()
      {synchronized(this)
                  {x++;  y++;}
      try {Thread.sleep(100);} catch(Exception e)  {  };
      }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class Thread_Syn_Method  extends Thread
{Pair p;
public Thread_Syn_Method(Pair p) {this.p=p;};
  public void run(   )
          {while (true)  {p.increment_syn_method(  );   } } }

public class Thread_Syn_Block extends Threadm{
 Pair p;
public Thread_Syn_Block(Pair p) {this.p=p;};
  public void run(   )
           {while (true)  {p.increment_syn_block();  }   }    }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class CountAccess extends Thread {

        Pair p;

        public CountAccess(Pair p)

                {this.p=p;}

        public void run(  ) {

        while (true)

             p.accessincrement( );

                                }          }
```

OTTIMIZZAZIONE SEZIONI CRITICHE

```
public class MainPair {  public static void main(String args[])
    {Pair syn_method_pair= new Pair(1,2);
    Pair syn_block_pair= new Pair(3,4);
    new Thread_Syn_Method(syn_method_pair).start( );
    new CountAccess(syn_method_pair).start( );
    new Thread_Syn_Block(syn_block_pair).start( );
    new CountAccess(syn_block_pair).start( );
    try      {Thread.sleep(200);}      catch(Exception e)  {  }
    System.out.println(
    "Accessi con blocco sincronizzato "+syn_block_pair.accessreturn( )+
    "     Accessi con metodo sincronizzato
      "+syn_method_pair.accessreturn( ));  }}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

Risultati di alcune esecuzioni:

Accessi con blocco sincronizzato 2258106 Accessi con metodo sincronizzato 2843

Accessi con blocco sincronizzato 2242580 Accessi con metodosincronizzato 2

Accessi con blocco sincronizzato 3749759 Accessi con metodo sincronizzato 1252

Conclusioni:

- il thread che accede all'oggetto coppia con blocco sincronizzato (invece che con un metodo sincronizzato) consente un maggior numero di accessi al thread Countaccess, che viene eseguito concorrentemente
- La versione che utilizza il blocco sincronizzato aumenta il numero di accessi concorrenti effettuati sull'oggetto coppia
- NOTA BENE: La differenza nel numero di accessi dipende ovviamente dal fatto che il programma simula la computazione costosa con una sleep( ). Se si sincronizza tutto il metodo, quando il thread esegue la sleep rilascia il processore, ma non la lock().

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 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

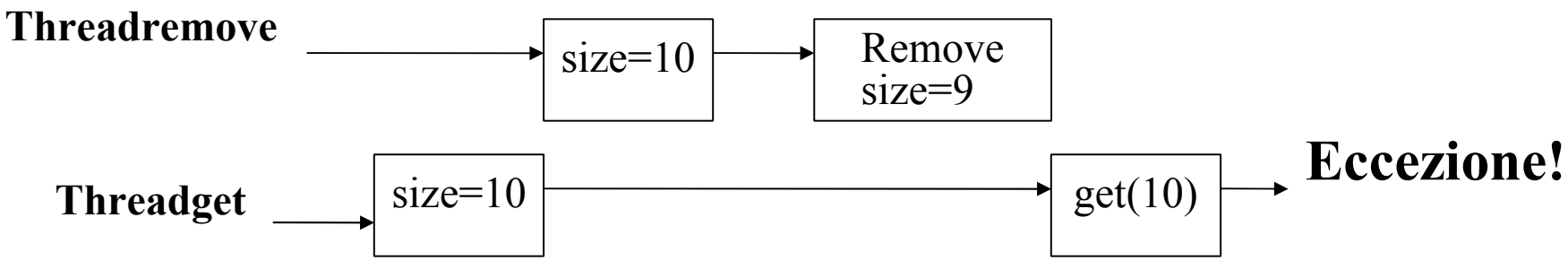
Eccezione sollevata dalla esecuzione del programma:

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

```
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 del 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 DI CODICE 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
- Code sincronizzate:comportamento di base
  - inserimento: aggiunge un elemento infondo alla cosa, 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/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 a 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 procacrawler.main(procacrawler.java:7)
```

CODE BLOCCANTI: METODI

Metodi che restituiscono una segnalazione del fallimento dell'operazione di inserzione/estrazione: offer,poll, peek

```
import java.util.concurrent.*;
public class procacrawler {
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 procacrawler {
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 files 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.