Lezione n.3
LPR-A-09
Threads: Sincronizzazione e Mutua Esclusione

6/10/2008

Vincenzo Gervasi

SULLA TERMINAZIONE: THREAD DEMONI

- Thread Demone (Daemon): fornisce un servizio, generalmente in background, fintanto che il programma è in esecuzione, ma non è considerato parte fondamentale di un programma
- Esempio: thread temporizzatori che scandiscono il tempo per conto di altri threads
- Quando tutti i thread ***non demoni*** hanno completato la lori esecuzione, il programma termina, anche se ci sono thread demoni in esecuzione
- Se ci sono thread non demoni in esecuzione, il programma non termina
  - Esempio: i thread attivati nel thread pool rimangono attivi anche se non esistono task da eseguire
- Si dichiara un thread demone invocando il metodo setdaemon(true), prima di avviare il thread
- Se un thread è un demone, allora anche tutti i threads da lui creati lo sono

TERMINAZIONE: THREAD DEMONI

```
public class SimpleDaemon extends Thread {
    public SimpleDaemon (  ) {
            setDaemon(true);
            start( );   }
    public void run(  ) {
             while(true) {
             try {
                    sleep(100);
              } catch (InterruptedException e) {
                        throw new RuntimeException(e);}
             System.out.println("mi sono svegliato"+this);
        }   }
```

TERMINAZIONE: THREAD DEMONI

```
public static void main(String[ ] args) {
      for(int i = 0; i < 5; i++)
                  new SimpleDaemon( );
      Thread.sleep(300);
}    }
```

- il main crea 5 threads demoni
- ogni thread 'si addormenta' e si risveglia per un certo numero di volte, poi quando il main termina (non daemon thread), anche i threads terminano)
- Se pongo **setDaemon(false)** (il default), il programma non termina, i thread continuano ad 'addormentarsi' e 'risvegliarsi'.

GRUPPI DI THREAD

- Alcuni programmi contengono un gran numero di thread. Può essere utile organizzare i thread in una struttura gerarchica in base alle loro funzionalità.
- Esempio: un browser attiva molti thread il cui compito è scaricare le immagini contenute in una pagina web. Se l'utente preme il pulsante stop(), è comodo utilizzare un metodo per interrompere ***tutti*** i threads simultaneamente.
- **Gruppi di thread**: sono insiemi di thread e di (sotto)gruppi di thread, raggruppati per esempio in base alle loro funzionalità, in modo che si possa lavorare su tutti threads di un gruppo simultaneamente.

```
ThreadGroup g  =   new  ThreadGroup("GruppoPadre");
ThreadGroup f  =   new ThreadGroup(g, "GruppoFiglio");
Thread t1 = new Thread(g, aTask, "T1");
Thread t2 = new Thread(g, anotherTask, "T2");
/* ... */
g.interrupt(  );  // interrompe tutti i thread del gruppo
```

CONDIVISIONE RISORSE TRA THREADS

- Più thread attivati da uno stesso programma possono condividere un insieme di oggetti. Schema tipico: gli oggetti vengono passati al costruttore del thread:

```
public class OggettoCondiviso
  { ..... }
```

```
public class Condivisione extends Thread {
    OggettoCondiviso oc;
    public Condivisione (OggettoCondiviso oc)  { this.oc = oc; }
    public void run () { ..... }
    public static void main(String args[ ]){
        OggettoCondiviso oc = new OggettoCondiviso();
        new Condivisione(oc).start();
        new Condivisione(oc).start();
    }
}
```

ACCESSO A RISORSE CONDIVISE

- L'interazione incontrollata dei threads sull'oggetto condiviso può produrre risultati non corretti
- Consideriamo il seguente esempio:
  - Definiamo una classe **EvenValue** che implementa un generatore di numeri pari.
  - Ogni oggetto istanza della classe ha un valore uguale ad un numero pari
  - Se il valore del numero è x, il metodo **next( )**, definito in **EvenValue** assegna il valore x+2
  - Si attivano un insieme di threads che condividono un oggetto di tipo **EvenValue** e che invocano **concorrentemente** il metodo **next( )**
  - Non si possono fare ipotesi sulla strategia di schedulazione dei threads

ACCESSO A RISORSE CONDIVISE

```
public  interface ValueGenerator {
                public int next( );     }


public class EvenValue implements ValueGenerator {
      private int currentEvenValue = 0;
      public int next( ) {
            ++currentEvenValue;
            Thread.yield ( );
            ++currentEvenValue;
            return currentEvenValue;}; }
```

Thread.*yield* ( ): "suggerisce" allo schedulatore di sospendere l'esecuzione del thread che ha invocato la **yield( )** e di cedere la CPU ad altri threads

ACCESSO A RISORSE CONDIVISE

```
import java.util.concurrent.*;
public class ThreadTester implements Runnable {
      private ValueGenerator gen;
      public ThreadTester(ValueGenerator gen)  {this.gen = gen;}
      public void run( ) {
            for (int i = 0; i < 5; i++){
                  int val= gen.next();
                  if (val % 2 !=0) System.out.println(Thread.currentThread( )+"errore"+val);
                        else System.out.println(Thread.currentThread( )+"ok"+val); }}
      public static void test(ValueGenerator gen, int count){
            ExecutorService exec= Executors.newCachedThreadPool();
            for (int i=0; i<count; i++){
                  exec.execute(new ThreadTester(gen));};
            exec.shutdown( ); }}
```

ACCESSO A RISORSE CONDIVISE

```
public class AccessTest {
public static void main(String args[ ]){
    EvenValue gen = new EvenValue();
    ThreadTester.test(gen, 2);} }
```

OUTPUT:
```
Thread[pool-1-thread-1,5,main]ok2
Thread[pool-1-thread-2,5,main]ok4
Thread[pool-1-thread-1,5,main]errore7
Thread[pool-1-thread-2,5,main]errore9
Thread[pool-1-thread-1,5,main]ok10
Thread[pool-1-thread-2,5,main]errore13
Thread[pool-1-thread-1,5,main]ok14
Thread[pool-1-thread-1,5,main]errore17
Thread[pool-1-thread-2,5,main]ok18
Thread[pool-1-thread-2,5,main]ok20
```

RACE CONDITION

- Perchè si è verificato l'errore?
- Supponiamo che il valore corrente di currentEvenValue sia 0.
- Il primo thread esegue il primo assegnamento

  ++currentEvenValue

  e viene quindi deschedulato, in seguito alla yield( ): **currentEvenValue** assume valore 1
- A questo punto si attiva il secondo thread, che esegue lo stesso assegnamento e viene a sua volta deschedulato, **currentEvenValue** assume valore 2
- Viene riattivato il primo thread, che esegue il secondo incremento, il valore assume valore 3
- ERRORE! : Il valore restituito dal metodo next( ) è 3.

RACE CONDITION e THREAD SAFETY

- Nel nostro caso la **race condition** (**corsa critica**) è dovuta alla possibilità che un thread invochi il metodo next( ) e venga deschedulato prima di avere completato l'esecuzione del metodo
- In questo modo la risorsa viene lasciata in uno stato inconsistente

  (un solo incremento per **currentEvenValue**)
- Classi **Thread Safe**: l'esecuzione concorrente dei metodi definiti nella classe non provoca comportamenti scorretti
- **EvenValue** non è una classe **thread safe**
- Per renderla thread safe occorre garantire che le istruzioni contenute all'interno del metodo next( ) vengano eseguite in modo atomico o indivisibile o in mutua esclusione

RACE CONDITION IN SINGOLA ISTRUZIONE

- Race Condition: si può verificare anche nella esecuzione di una singola istruzione di assegnamento
- Consideriamo un'istruzione che incrementa una variabile intera:

  count = count + 1; o count++;
- L'istruzione può essere elaborata come segue
  1) il valore di count viene caricato in un registro
  2) si somma 1
  3) si memorizza il risultato in count
- Un thread T potrebbe eseguire i passi 1), 2) e poi venire deschedulato,
- Viene quindi schedulato un secondo thread Q che esegue tutta l'istruzione
- T esegue il passo 3) assegnando a count il valore che già contiene: un aggiornamento si è perso.

Esercizio 1

Si scriva un programma Java che dimostri che si possono verificare delle race conditions anche con una singola istruzione di incremento di una variabile.

- Scrivere una classe **Counter** che offre un metodo **next**() che incrementa una variabile locale, e un metodo **getCount**() che ne restituisce il valore.
- Scrivere un task **TaskCounter** che implementa **Callable** e che riceve nel costruttore un **Counter** e un intero **n**. Il task invoca la **next**() del **Counter** un numero casuale di volte compreso tra **n/2** e **n**, e restituisce il numero casuale calcolato.
- Il main crea un **Counter** e un pool di threads, in cui esegue **M** copie di **TaskCounter** passando a ognuna di esse il **Counter** e un valore **N**; quindi stampa la somma dei valori restituiti dagli **M** threads, e il valore finale del contatore ottenuto con **getCount**(): se questi due valori sono diversi c'è stata una race condition. **M** e **N** devono essere forniti dall'utente.

RACE CONDITION: LAZY INITIALIZATION

Un altro esempio di una classe non thread safe

```
public class LazyInitRace {
        private ExpensiveObject instance = null;
        public ExpensiveObject getInstance( ){
            if (instance == null)
                    instance = new ExpensiveObject();
             return instance:
        }}
```

- **Lazy Initialization** =
  - alloca un oggetto solo se non esiste già un'istanza di quell'oggetto
  - deve assicurare che l'oggetto venga inizializzato una sola volta
  - getInstance( ) deve restituire sempre la stessa istanza

RACE CONDITIONS: ESEMPI

```
public class LazyInitRace {
        private ExpensiveObject instance = null;
        public ExpensiveObject getInstance( ){
            if (instance == null)
                    instance = new ExpensiveObject();
             return instance:
          }}
```

- Il programma non è corretto perchè contiene una race condition
- Il thread **A** esegue getInstance(), trova (instance == **null**), poi viene deschedulato. Il thread **B** esegue getInstance() e trova a sua volta (instance == **null**). I due thread restituiscono due diverse istanze di **ExpensiveObject**

JAVA: MECCANISMI DI LOCK

- Occorre disporre di meccanismi per garantire che un metodo (es: next()) venga eseguito in **mutua esclusione** quando invocato su di un oggetto
- JAVA offre un meccanismo implicito di locking (***intrinsic locks***) che consente di assicurare la atomicità di porzioni di codice eseguite in modo concorrente sullo stesso oggetto
- L'uso del lock garantisce che se un thread T esegue un metodo di istanza di un oggetto, nessun altro thread che richiede il lock può eseguire un metodo sullo stesso oggetto fino a che T non ha terminato l'esecuzione del metodo

JAVA: IL COMANDO synchronized

- Sincronizzazione di un blocco:

```
synchronized (obj)
    { // blocco di codice che accede o modifica l'oggetto
    }
```

- 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 il lock** sull'oggetto **obj**
- Il lock viene **rilasciato** nel momento in cui il thread termina l'esecuzione del blocco (es: **return**, **throw**, esecuzione dell'ultima istruzione del blocco)

JAVA: RENDERE ATOMICO IL METODO NEXT

```
public class EvenGenerator implements ValueGenerator{
    private int currentEvenValue = 0;
    public int next(  ){
            synchronized(this){
                ++currentEvenValue;
                Thread.yield();
                ++currentEvenValue;
                return currentEvenValue;} } }
```

- Questa modifica consente di evitare le race conditions
- ATTENZIONE: in generale non è consigliabile l'inserimento di una istruzione che sospende il thread che la invoca (sleep( ), yield( ),....) all'interno di un blocco sincronizzato
- Infatti il thread che si blocca non rilascia il lock ed impedisce ad altri threads di invocare il metodo next( ) sullo stesso oggetto

JAVA 1.5: LOCK ESPLICITI

- A partire da JAVA 1.5 è possibile definire ed utilizzare oggetti di tipo lock

```
import java.util.concurrent.locks.*;
class X {
    private final ReentrantLock mylock = new ReentrantLock( );
    // .
    .public void m(  ) {
            mylock.lock(    );  // block until condition holds

            try {
            // ... method body
        } finally {lock.unlock( )  } } }
```

JAVA 1.5: LOCK ESPLICITI

- Vediamo un'altra versione del nostro esempio con lock esplicito:

```
import java.util.concurrent.locks.*;
public class LockingEvenGenerator implements ValueGenerator {
    private int currentEvenValue = 0;
    ReentrantLock evenlock=new ReentrantLock( );
    public int next( ) {
        try {
            evenlock.lock( );
            ++currentEvenValue;
            Thread.yield ( );
            ++currentEvenValue;
            return currentEvenValue;
        } finally {evenlock.unlock( );}}}
```

JAVA: MECCANISMO DEI LOCK

- Lock **espliciti**: si definisce un oggetto di tipo ReentrantLock( )
  - Quando un thread invoca il metodo **lock( )** su un oggettodi tipo ReentrantLock( ), il thread rimane bloccato se qualche altro thread ha già acquisito il **lock**
  - Quando un thread invoca **unlock( )** uno dei thread eventualmente bloccati su quella **lock** viene risvegliato
- Lock **impliciti** su metodi
  - Sono definiti associando al metodo la parola chiave **synchronized**
  - Equivale a sincronizzare tutto il blocco di codice che corrisponde al corpo del metodo
  - L'oggetto su cui si acquisisce il lock è quello su cui viene invocato il metodo

I METODI SYNCHRONIZED

- La parola chiave **synchronized** nella intestazione di un metodo ha l'effetto di serializzare gli accessi al metodo

```
public synchronized int next ( )
```

- Se un thread sta eseguendo il metodo next( ), nessun altro thread potrà eseguire lo stesso codice sullo stesso oggetto finchè il primo thread non termina l'esecuzione del metodo
- Implementazione:
  - Supponiamo che il metodo **m synchronized** appartenga alla classe *C*
  - ad ogni oggetto *O* istanza di *C* viene associata un lock, *L(O)*
  - quando un thread T invoca **m** su *O*, T tenta di acquisire L(O), prima di iniziare l'esecuzione di M. Se T non acquisisce L(O), si sospende

I METODI SYNCHRONIZED

- Se rendiamo synchronized il metodo next(), l'output che otteniamo sarà

```
Thread[pool-1-thread-1,5,main]ok2
Thread[pool-1-thread-2,5,main]ok4
Thread[pool-1-thread-1,5,main]ok6
Thread[pool-1-thread-2,5,main]ok8
Thread[pool-1-thread-1,5,main]ok10
Thread[pool-1-thread-2,5,main]ok12
Thread[pool-1-thread-1,5,main]ok14
Thread[pool-1-thread-2,5,main]ok16
Thread[pool-1-thread-1,5,main]ok18
Thread[pool-1-thread-2,5,main]ok20
```

I METODI SYNCHRONIZED

- Importante: il lock è associato all'istanza di un oggetto, non al metodo o alla classe (a meno di metodi statici che vedremo in seguito)
- Diversi metodi sincronizzati invocati sull'istanza dello stesso oggetto competono per lo stesso lock, quindi risultano mutuamente esclusivi
- Metodi sincronizzati che operano su istanze diverse dello stesso oggetto possono essere eseguiti in modo concorrente
- All'interno della stessa classe possono comparire contemporaneamente metodi sincronizzati e non (anche se raramente)
  - I metodi non sincronizzati possono essere eseguiti in modo concorrente
  - In ogni istante, su un certo oggetto, possono essere eseguiti concorrentemente più metodi non sincronizzati e solo uno dei metodi sincronizzati della classe

I METODI SYNCHRONIZED

L'esempio seguente crea **due** istanze dell'oggetto EvenValue1( ) e le passa a due thread distinti. Si considera la versione **non sincronizzata** del metodo next( )

```
public class EvenValue1 implements ValueGenerator{
private int currentEvenValue = 0;
    public int next( ){ ++currentEvenValue; ++currentEvenValue;
            return currentEvenValue; } }
public class SynchroTest {
        public static void main(String args[ ]){
        ValueGenerator eg1 = new EvenValue1();
        ValueGenerator eg2 = new EvenValue1();
        ThreadTester1.test(eg1,eg2);}}
```

I METODI SYNCHRONIZED

```
import java.util.concurrent.*;
import java.util.Random;
public class ThreadTester1 implements Runnable{
      private ValueGenerator g;
      public ThreadTester1 (ValueGenerator g)  {this.g = g;}
      public void run( ){
            for (int i=0; i<5; i++){
                  int val= g.next();
                  if (val % 2 !=0)
                  System.out.println(Thread.currentThread() + "errore" + val);
                  else System.out.println(Thread.currentThread() + "ok" + val);
                  try {Thread.sleep((int) Math.random() * 1000);}
                  catch (Exception e) {   };  }}
```

I METODI SYNCHRONIZED

```
public static void test(ValueGenerator g1, ValueGenerator g2){
    ExecutorService exec= Executors.newCachedThreadPool();
    exec.execute(new tester(g1));
    exec.execute(new tester(g2));   exec.shutdown() }  }
```

OUTPUT: il risultato è corretto anche se next() non è sincronizzato

```
Thread[pool-1-thread-1,5,main]ok2
Thread[pool-1-thread-2,5,main]ok2
Thread[pool-1-thread-2,5,main]ok4
Thread[pool-1-thread-2,5,main]ok6
Thread[pool-1-thread-2,5,main]ok8
Thread[pool-1-thread-2,5,main]ok10
Thread[pool-1-thread-1,5,main]ok4
Thread[pool-1-thread-1,5,main]ok6
Thread[pool-1-thread-1,5,main]ok8
Thread[pool-1-thread-1,5,main]ok10
```

Esercizio 2

- Si consideri il metodo **next**() della classe **Counter** dell'Esercizio 1. Modificarlo in modo da renderne l'esecuzione non interrompibile, e rieseguire il programma verificando che non si verificano più race conditions. Fare questo nei tre modi visti:
  - usando un comando **synchronized**
  - usando un **lock esplicito**
  - dichiarando **synchronized** il metodo **next**()
-

LOCK RIENTRANTI

- I lock intrinseci di JAVA sono rientranti, ovvero il lock( ) su un oggetto O viene ***associato al thread*** che accede ad O.

- se un thread tenta di acquisire un lock che già possiede, la sua richiesta ha successo

- Ovvero... un thread può invocare un metodo sincronizzato **m** su un oggetto O e all'interno di **m** vi può essere l'invocazione ad un altro metodo sincronizzato su O e così via

- Il meccanismo dei lock rientranti favorisce la prevenzione di situazioni di **deadlock**

LOCK RIENTRANTI

- Implementazione delle lock rientranti
  - Ad ogni lock viene associato un contatore ed un identificatore di thread
  - Quando un thread T acquisisce un lock, la JVM alloca una struttura che contiene l'identificatore T e un contatore, inizializzato a 0
  - Ad ogni successiva richiesta dello stesso lock, il contatore viene incrementato mentre viene decrementato quando il metodo termina
- Il lock( ) viene rilasciato quando il valore del contatore diventa 0

REENTRANT LOCK E EREDITARIETA'

```
public class ReentrantExample {
    public  synchronized  void doSomething(  ) {
                        ..............} }

public class ReentrantExtended extends ReentrantExample{
    public synchronized void doSomething( ){
        System.out.println(toString( ) + ": chiamata a doSomething");
        super.doSomething();
        } }
```

- La chiamata super.doSomething( ) si bloccherebbe se il lock non fosse rientrante, ed il programma risulterebbe bloccato (**deadlock**)

MUTUA ESCLUSIONE: RIASSUNTO

- Interazione implicita tra diversi threads: i thread accedono a risorse condivise.
- Per mantenere consistente l'oggetto condiviso occorre garantire la mutua esclusione su di esso.
- La mutua esclusione viene garantita associando un lock ad ogni oggetto
- I metodi synchronized garantiscono che un thread per volta possa eseguire un metodo sull'istanza di un oggetto e quindi garantiscono la mutua esclusione sull'oggetto

ESERCIZIO 3: ANCORA RACE CONDITIONS

Simulare il comportamento di una banca che gestisce un certo numero di conti correnti. In particolare interessa simulare lo spostamento di denaro tra due conti correnti.

Ad ogni conto è associato un thread T che implementa un metodo che consente di trasferire una quantità casuale di denaro tra il conto servito da T ed un altro conto il cui identificatore è generato casualmente.

- sviluppare una versione non thread safe del programma in modo da evidenziare un comportamento scorretto del programma
- definire 3 versioni thread safe del programma che utilizzino, rispettivamente
  - Lock esplicite
  - Blocchi sincronizzati
  - Metodi sincronizzati

THREADS COOPERANTI: I MONITOR

- L'interazione esplicita tra threads avviene in un linguaggio ad oggetti come JAVA mediante l'utilizzo di oggetti condivisi
- Esempio: produttore/consumatore il produttore P produce un nuovo valore e lo comunica ad un thread consumatore C
- Il valore prodotto viene incapsulato in un oggetto condiviso da P e da C, ad esempio una coda che memorizza i messaggi scambiati tra P e C
- La mutua esclusione sull'oggetto condiviso è garantita dall'uso di metodi synchronized, ma...non è sufficiente garantire sincronizzazioni esplicite
- E' necessario introdurre costrutti per sospendere un thread T quando una condizione C non è verificata e per riattivare T quando diventa vera
- Esempio: il produttore si sospende se il buffer è pieno, si riattiva quando c'è una posizione libera

THREADS COOPERANTI: I MONITOR

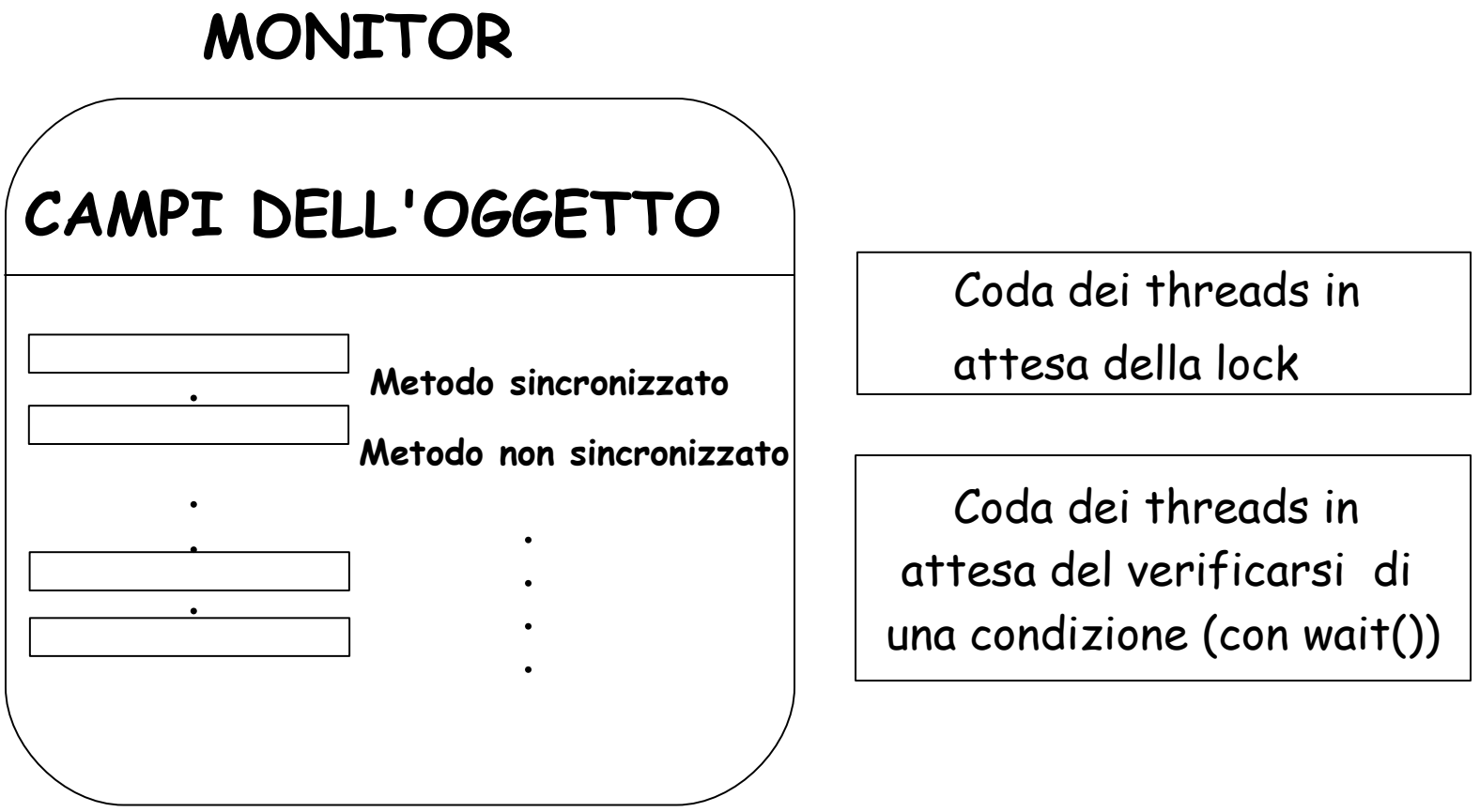
- Monitor = Classe di oggetti utilizzabili da un insieme di threads
- Come ogni classe, il monitor ha un insieme di campi ed un insieme di metodi
- La mutua esclusione può essere garantita dalla definizione di metodi synchronized. Un solo thread per volta si trova "all'interno del monitor"
- E' necessario inoltre
  - definire un insieme di condizioni sullo stato dell'oggetto condiviso
  - implementare meccanismi di sospensione/riattivazione dei threads sulla base del valore di queste condizioni
  - Implementazioni possibili:
    - definizione di variabili di condizione
    - metodi per la sospensione su queste variabili
    - definizione di code associate alle variabili in cui memorizzare i threads sospesi

THREADS COOPERANTI: I MONITOR

JAVA

- non supporta variabili di condizione
- assegna al programmatore il compito di gestire le condizioni mediante variabili del programma
- definisce meccanismi che consentono ad un thread
  - di sospendersi wait( ) in attesa che sia verificata una condizione
  - di segnalare con notify( ), notifyall ( ) ad un altro/ad altri threads sospesi che una certa condizione è verificata
- Per ogni oggetto implementa due code:
  - una coda per i thread in attesa di acquisire il lock
  - una coda in cui vengono memorizzati tutti i thread sospesi con la wait( ) (in attesa del verificarsi di una condizione).

THREADS COOPERANTI: I MONITOR



ESEMPIO: PRODUTTORE/CONSUMATORE

- **Produttore/Consumatore**: due thread si scambiano dati attraverso un oggetto condiviso buffer
- Ogni thread deve acquisire il lock sull'oggetto buffer, prima di inserire/prelevare elementi
- Una volta acquisito il lock
  - il **consumatore** controlla se c'è almeno un elemento nel buffer:
    - in caso positivo, preleva un elemento dal buffer e risveglia l'eventuale produttore in attesa;
    - se il buffer è vuoto si sospende,
  - il **produttore** controlla se c'è almeno una posizione libera nel buffer:
    - in caso positivo, inserisce un elemento nel buffer e risveglia l'eventuale consumatore in attesa,
    - se il buffer è pieno si sospende.

I METODI WAIT/NOTIFY

Metodi d'istanza invocati sull'oggetto condiviso (se non compare il riferimento all'oggetto, ovviamente l'oggetto implicito riferito è this)

- **void wait( )** sospende il thread sull'oggetto
- **void wait(long timeout)** sospende per al massimo timeout millisecondi
- **void notify( )** risveglia un thread in attesa sull'oggetto
- **void notifyall( )** risveglia tutti i threads in attesa sull'oggetto

Tutti questi metodi

- sono definiti nella classe Object (tutti le classi ereditano da Object,....)
- per invocare questi metodi occorre aver acquisito il lock sull'oggetto, altrimenti viene lanciata una **IllegalMonitorStateException**. Quindi vanno invocati all'interno di un metodo o di un blocco sincronizzato, o dopo aver acquisito un lock eslpicito.

WAIT, NOTIFY E IL LOCK

wait( )

- rilascia il lock sull'oggetto prima di sospendere il thread corrente
- quando risvegliato da una notify( ), il thread compete per riacquisire il lock

notify( )

- risveglia uno dei thread (non si sa quale...) nella coda di attesa dell'oggetto; non rilascia immediatamente il lock

notifyAll( )

- risveglia tutti i threads in attesa sull'oggetto; non rilascia il lock

Tutti i thread risvegliati competono per l'acquisizione del lock sull'oggetto, e verranno eseguiti uno alla volta, quando riusciranno a riacquisire il lock.

WAIT E NOTIFY

- Il metodo wait() permette di attendere il cambiamento di una condizione sullo stato dell'oggetto "fuori dal monitor", in modo passivo
- Evita il controllo ripetuto di una condizione (polling)
- A differenza di sleep() e di yield(), rilascia il lock sull'oggetto
- Quando ci si sospende su di una condizione, occorre controllarla quando si viene risvegliati:

```
synchronized(obj){
    while (<condizione non soddisfatta>)
        obj.wait();
    ... // esegui l'azione per cui richiedevi la condizione
}
```

- L'invocazione di un metodo wait(), notify(), notifyall() fuori da un metodo synchronized solleva l'eccezione IllegalMonitorException( ): prima di invocare questi metodi occorre aver acquisito il lock su un oggetto condiviso

PRODUTTORE/CONSUMATORE: IL BUFFER

```
public class Buffer {
    int[] buffer; int size = 0; // Array Parzialmente Riempito
    public Buffer(int capacity){ buffer = new int[capacity]; }
    public synchronized int get() throws InterruptedException{
        while (size == 0) wait();
        int result = buffer[--size];     // restituisce l'ultimo
        notify();                        // elemento inserito
        return result;  }
    public synchronized void put(int n) throws InterruptedException{
        while (size == buffer.length) wait();
        buffer[size++] = n;
        notify(); } }
```

IL PRODUTTORE

```
public class Producer implements Runnable {
    private Buffer buf;
    public Producer(Buffer buf){ this.buf = buf; }
    public void run() {
        try{
            while(true){
            int next = (int)(Math.random() * 10000);
            buf.put(next);
            System.out.println("[Producer] Put " + next);
            Thread.sleep((int)(Math.random() * 500)); }
        } catch (InterruptedException e){
            System.out.println("[Producer] Interrupted");
        } } }
```

IL CONSUMATORE

```
public class Consumer implements Runnable {
    private Buffer buf;
    public Consumer(Buffer buf){ this.buf = buf; }
    public void run() {
        try{
            while(true){
                System.out.println("[Consumer] Got " + buf.get());
                Thread.sleep((int)(Math.random() * 500)); }
        }catch (InterruptedException e){
            System.out.println("[Consumer] Interrupted");
        } } }
```

PRODUTTORE/CONSUMATORE: IL MAIN

```
import java.util.concurrent.*;
public class TestProducerConsumer {
public static void main(String[] args) {
      Buffer buf = new Buffer(5);
      Consumer cons = new Consumer(buf);
      Producer prod = new Producer(buf);
      ExecutorService exec = Executors.newFixedThreadPool(2);
      exec.execute(cons);
      exec.execute(prod);
      try { Thread.sleep(10000); } catch (InterruptedException e) { }
      exec.shutdownNow();
      } }
```

PRODUCER/CONSUMER: OUTPUT

Esempio di output del programma:

```
[Consumer] Got 7493
[Producer] Put 7493
[Producer] Put 4495
[Consumer] Got 4495
[Producer] Put 1515
[Producer] Put 8502
[Consumer] Got 8502
[Producer] Put 4162
[Producer] Put 954
[Producer] Put 880
[Consumer] Got 880
[Producer] Put 8503
[Producer] Put 4980
[Consumer] Got 4980
[Consumer] Got 8503
[Producer] Put 3013
[Consumer] Got 3013
[Consumer] Interrupted
[Producer] Interrupted
```

- Si noti che poiché il buffer ha una politica Last In First Out (LIFO), l'ordine non viene preservato. Per esempio, il numero 1515 non viene mai estratto dal buffer.

ESERCIZIO 4

- La classe Buffer ha una politica Last In First Out (LIFO), quindi non preserva l'ordine. Scrivere la classe CircularBuffer che estende Buffer e realizza una politica FIFO, gestendo l'array in modo circolare.
- Definire le interfacce generiche Producer<E>, Consumer<E> e Buffer<E>, che definiscono un sistema produttore/consumatore per un generico tipo di dati E.
- Implementare le interfacce in modo che il produttore produca una sequenza di stringhe, leggendole da un file passato come parametro al task, e il consumatore scriva le stringhe che prende dal buffer in un altro file.
- Nel main, creare e attivare un produttore e due o più consumatori. Verificare che la concatenazione dei file generati dai consumatori sia uguale, a meno dell'ordine delle righe, al file letto dal produttore.

ESERCIZIO 5 (a)

Il laboratorio di Informatica del Polo Marzotto è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.

b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice i, poichè su quel computer è istallato un particolare software necessario per lo sviluppo della tesi.

c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti. (prosegue nella pagina successiva)

ESERCIZIO 5(b)

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede k volte al laboratorio, con k generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo sleep. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.