Creazione di Thread in Java

```
public class CreaThread extends Thread {
     public void run() {
          System.out.println("Saluti dal thread " +
this.getName());
     }

     public static void main(String args[]) {
          CreaThread t = new CreaThread();
          t.start();
     }
}
```

Oppure, se c'è esigenza di estendere altre classi, i thread possono anche essere creati come oggetti della classe `Thread`, al cui costruttore viene passato un oggetto che implementa l'interfaccia `Runnable`, che richiede l'implementazione del metodo `run`.

```
public class CreaThread2 implements Runnable {
     public void run() {
          System.out.println("Saluti dal thread " +
Thread.currentThread().getName());
     }

     public static void main(String args[]) {
          CreaThread2 r = new CreaThread2();
          Thread t = new Thread(r);
          t.start();
     }
}
```

"addormentare" un thread

```
  public void run()  {
     try {
          sleep(1000); // attende un secondo
     } catch(InterruptedException e) {
          System.out.println("["+getName()+"]"+" Ah mi hanno
interrotto!!");
          return;
     }
```

I Monitor di Java

In Java è implementata una forma semplificata dei monitor con le seguenti caratteristiche:

Ogni oggetto ha un *mutex* implicito utilizzato per garantire mutua esclusione sui metodi;

I metodi sono eseguiti in mutua esclusione solo se dichiarati `synchronized`;

Ogni oggetto ha un'unica condition implicita sulla quale si possono effettuare le operazioni standard `wait()`, `notify()`, `notifyAll()`;

se il metodo è statico allora il mutex è a livello di classe invece che di oggetto;

è inoltre possibile sincronizzare parti di codice di un metodo non synchronized nel seguente modo:

```
synchronized(this) {
        contatore=contatore+1
    }
```

ESERCIZIO 1 (la soluzione è in basso, guardatela dopo aver provato da soli): creare `n` thread, rallentarli tramite `sleep()`, provare ad interromperne l'attesa utilizzando `t.interrupt()` e attendere la terminazione con `t.join()` (analogamente a quanto si fa con i thread POSIX), dove `t` è l'oggetto thread (Aggiungere opportune print per osservare l'esecuzione).

```
public class CreaTantiThread extends Thread {
    static final int NTHREAD=5;      // numero di
thread da creare
    final int index;                 // indice del
thread appena creato

    // costruttore: memorizza l'indice del thread
    CreaTantiThread(int i) {
        index = i;
    }

    // codice da eseguire allo startup del thread
    public void run()  {
        try {
            // NOTA: anche se l'interruzione arriva
```

```
prima dello sleep viene bufferizzata!
            sleep(1000*index);  // dorme per index
secondi
        } catch(InterruptedException e) {
            System.out.println("["+getName()+"]"+"
Ah mi hanno interrotto!!");
            return;
        }
        // saluta ed esce
        System.out.println("Saluti dal thread " +
getName());
    }

    /* main: crea i NTHREAD thread ne interrompe
alcuni e attende la terminazione
       NOTA: con join devo gestire
InterruptedException, ma nessuno interrompera' mai
le
       join di questo main quindi la ignoriamo */
    public static void main(String args[]) throws
InterruptedException {
        int i;
        Thread t[] = new Thread[NTHREAD];

        // crea 5 thread e li esegue
        for(i=0;i<NTHREAD;i++) {
            t[i] = new CreaTantiThread(i);
            t[i].start();
        }

        // interrompe il terzo thread
        t[3].interrupt();

        // attende la terminazione dei thread
        for(i=0;i<NTHREAD;i++) {
            t[i].join();
        }

        // saluta ed esce
        System.out.println("Saluti dal thread " +
Thread.currentThread().getName());
```

```
}
}
```

ESERCIZIO 2 (la soluzione è in basso, guardatela dopo aver provato da soli): Implementare due thread che lavorano su un contatore condiviso, osservare le usuali interferenze e rimediare ponendo il codice all'interno di metodi `synchronized` o all'interno del costrutto `synchronized(this) { ... }` . Suggerimento: mettere il contatore in una classe apposita (che fungerà da monitor) e implementare i metodi per l'incremento e la lettura/stampa del valore.

```
public class Interferenze2 extends Thread {
    static final int MAX=1000000;           // iterazioni
massime
    Contatore c;                            // Monitor
passato dal main

    // costruttore, memorizza il monitor nel campo c
    Interferenze2(Contatore cont) {
        c=cont;
    }
    //   i thread incrementano MAX volte il contatore
    //   NOTA: non possono fare c.count++ perche' e'
privato!
    public void run() {
        int i;
        for (i=0;i<MAX;i++)
            c.incrementa();     // questo metodo e' in
MUTEX perche' synchronized
    }

    //   il main crea i thread, attende la terminazione e
stampa il contatore
    public static void main(String args[]) throws
InterruptedException {
        int j=0;
        Thread t[] = new Thread[2];
        Contatore cont = new Contatore(); // crea un
```

```
singolo monitor

        // crea i 2 thread e li esegue
        for(j=0;j<2;j++) {
            t[j] = new Interferenze2(cont); // passa il
monitor ai thread
            t[j].start();
        }

        // attende la terminazione
        for(j=0;j<2;j++) t[j].join();

        // stampa il contatore, il valore atteso ed esce
        System.out.println("FINITO " + cont.valore() + "
mi aspettavo " + MAX*2);
    }
}

/*  questa classe implementa un Monitor in cui e'
possibile incrementare il valore di un
 *  contatore da diversi thread in mutua esclusione.
 *  provare a togliere il 'synchronized' dal metodo
incrementa per osservare
 *  le interferenze */
class Contatore {
    private int count=0;    // privato: no accessi
diretti!

    // il metodo synchronized garantisce mutua esclusione
sullo stesso oggetto
    synchronized void incrementa() {
        //oppure: synchronized(this) {count++;}
        count++;
    }

    // non serve sincronizzarlo visto che lo usiamo alla
fine dal main: gli altri thread sono
    // gia' terminati (la join garantisce che il main e'
l'unico thread in esecuzione)
    // inoltre la lettura non crea mai interferenze
    int valore() {
```

```
        return(count);
    }
}
```

ESERCIZIO 3: Implementare i filosofi a cena. Provare a `schedularli' (tramite opportune sleep) in modo da osservare lo stallo.

```
import java.util.*;

public class Filosofi2 extends Thread {
    private Monitor2 monitor; // monitor, uguale per
tutti, passato dal main
    public int id; // id, differente per tutti, passato
dal main

    // costanti numero di filosofi e di pasti
    public static final int FILOSOFI = 5;
    public static final int PASTI = 9;

    // Costruttori
    public Filosofi2() {
    }

    public Filosofi2(int i, Monitor2 m) {
        id = i;
        monitor = m;
    }

    // operazioni svolte da un filosofo
    public void run() {
        // continua a mangiare fino a che ci sono pasti
        while(monitor.ancoraPasti()) {
            try {
                System.out.println(getName() + " sta
pensando");
                sleep(500); // pensa
                monitor.raccogliSx(this); // raccogli
```

```
prima bacchetta
                sleep(1000); // pausa messa per forzare il
deadlock
                monitor.raccogliDx(this); // raccogli
l'altra bacchetta
                // mangia
                System.out.println("                " +
getName() + " sta mangiando");
                sleep(1000);
                System.out.println("                " +
getName() + " ha finito di mangiare");
                // deposita le bacchette
                monitor.depositaSx(id);
                monitor.depositaDx(id);
            }
            catch(InterruptedException e) {
                System.out.println(e);
            }
        }
        //System.out.println("Fine" + getName());
    }

    public static void main(String argv[]) throws
InterruptedException {
        int i;
        // creo un monitor condiviso
        Monitor2 monitor = new Monitor2();

        // creo un array di thread
        Filosofi2[] filosofi = new
Filosofi2[Filosofi2.FILOSOFI];

        // lancio i thread passando a ciascuno il proprio
id e il monitor condiviso
        for(i = 0; i < Filosofi2.FILOSOFI; i++) {
            filosofi[i] = new Filosofi2(i, monitor);
            filosofi[i].start();
        }

        // attendo la fine dei thread
        for(i = 0; i < Filosofi2.FILOSOFI; i++)
```

```
            filosofi[i].join();

        System.out.println("\nFine del
programma");

    }

}

class Monitor2 {
    private boolean[] bacchette = new
boolean[Filosofi2.FILOSOFI]; // array di bacchette, true
se è libera
    private int pasti = Filosofi2.PASTI; // contatore
pasti rimasti da servire
    private int posti = 0; // contatore dei filosofi che
concorrono a prendere una bacchetta

    // il costruttore inizializza a true tutte le
bacchette
    public Monitor2() {
        int i;
        for(i = 0; i < Filosofi2.FILOSOFI; i++)
            bacchette[i] = true;
    }

    // metodi in mutua esclusione
    // torna true se ci sono altri pasti da servire
    public synchronized boolean ancoraPasti() {
        return (pasti > 0);
    }

    // raccoglie la bacchetta sx
    public synchronized void raccogliSx(Filosofi2
filosofo) throws InterruptedException {
        // decrementa il numero dei pasti disponibili,
appena un filosofo arriva in raccogli sicuramente prima o
poi mangia
        pasti--;
        int id = filosofo.id;
        // controllo sul numero di filosofi che può
```

```
competere a prendere la bacchetta
        while((posti + 1) == Filosofi2.FILOSOFI) {
            System.out.println("        " +
filosofo.getName() + " attende perché le bacchette non
bastano");
            wait(); // se non può attendi
        }
        posti++; // se può contalo
        // controllo se la bacchetta è occupata
        while(!bacchette[id]) {
            System.out.println("        " +
filosofo.getName() + " attende la bacchetta sx");
            wait();
        }
        // se non lo è prendila
        bacchette[id] = false;
        System.out.println("        " + filosofo.getName()
+ " raccoglie la bacchetta " + id);
    }

    // preferisco scrivere raccogliDx senza sfruttare
raccogliSx
    // perché raccogliSx modifica le variabili posti e
pasti che non servono a raccogliDx
    // e perché è più chiaro l'output del programma
    public synchronized void raccogliDx(Filosofi2
filosofo) throws InterruptedException {
        int id = (filosofo.id + 1) % Filosofi2.FILOSOFI;
        // pasti e posti sono già stati incrementati, mi
blocco solo se la bacchetta dx non è libera
        while(!bacchette[id]) {
            System.out.println("        " +
filosofo.getName() + " attende la bacchetta dx");
            wait();
        }
        // se è libera prendi la bacchetta
        bacchette[id] = false;
        System.out.println("        " + filosofo.getName()
+ " raccoglie la bacchetta " + id);
    }
```

```
    // deposito le bacchette, settale a true
    public synchronized void depositaSx(int id) {
        bacchette[id] = true;
    }

    public synchronized void depositaDx(int id) {
        depositaSx((id + 1) % Filosofi2.FILOSOFI);
        posti--; // libera un posto
        notifyAll(); // sblocca tutti i thread
    }

}
```

ESERCIZIO 5 (NEW): Evitare la starvation della soluzione precedente introducendo una coda in Java che implementi una politica First Come First Serve (FCFS o FIFO). L'idea è che prima di un blocco while-wait il thread si accodi. Nella condizione del while controlliamo in primo luogo se il thread non è il primo della coda, e in tal caso lo blocchiamo di nuovo indipendentemente dalla condizione di attesa. Quando il thread supera il blocco while lo rimuoviamo dalla coda.

```
import java.util.*;

public class Filosofi extends Thread {
    private Monitor monitor; // monitor, uguale per tutti,
passato dal main
    public int id; // id, differente per tutti, passato
dal main

    // costanti numero di filosofi e di pasti
    public static final int FILOSOFI = 5;
    public static final int PASTI = 9;

    // Costruttori
    public Filosofi() {
    }
```

```
    public Filosofi(int i, Monitor m) {
        id = i;
        monitor = m;
    }

    // operazioni svolte da un filosofo
    public void run() {
        // continua a mangiare fino a che ci sono pasti
        while(monitor.ancoraPasti()) {
            try {
                // pensa per 1 secondo
                System.out.println("filosofo " + id + "
sta pensando");
                sleep(1000);
                // raccogli le bacchette
                monitor.raccogli(this);
                // mangia per 2 secondi
                System.out.println("
filosofo " + id + " sta mangiando");
                sleep(2000);
                System.out.println("
filosofo " + id + " ha finito di mangiare");
                // deposita le bacchette
                monitor.deposita(this);
            }
            catch(InterruptedException e) {
                System.out.println(e);
            }
        }
        //System.out.println("Fine " +
getName());
    }

    public static void main(String argv[]) throws
InterruptedException {
        int i;
        // creo un monitor condiviso
        Monitor monitor = new Monitor();

        // creo un array di thread
        Filosofi[] filosofi = new
```

```
Filosofi[Filosofi.FILOSOFI];


        // creo i thread, passando a ciascuno il proprio
id e il monitor condiviso
        for(i = 0; i < Filosofi.FILOSOFI; i++) {
            filosofi[i] = new Filosofi(i, monitor);
        }

        // li aggiungo alla coda del monitor
        for(i = 0; i < Filosofi.FILOSOFI; i = i + 2) {
            monitor.aggiungi(filosofi[i]); // prima i pari
        }

        for(i = 1; i < Filosofi.FILOSOFI;  i= i + 2) {
            filosofi[i] = new Filosofi(i, monitor); // poi
i dispari
            monitor.aggiungi(filosofi[i]);
        }

        for(i = 0; i < Filosofi.FILOSOFI; i++) {
            filosofi[i].start(); // lancio i thread
        }

        // attendo la fine dei thread
        for(i = 0; i < Filosofi.FILOSOFI; i++)
            filosofi[i].join();

        System.out.println("\nFine programma");
    }

}

class Monitor {
    private boolean[] bacchette = new
boolean[Filosofi.FILOSOFI]; // array di bacchette, true se
è libera
    private int pasti = Filosofi.PASTI; // contatore pasti
rimasti da servire
    public Queue<Filosofi> queue = new
LinkedList<Filosofi>(); // coda in cui memorizzo i
```

```
filosofi per ordine di attesa

    // il costruttore inizializza a true tutte le
bacchette
    public Monitor() {
        int i;
        for(i = 0; i < Filosofi.FILOSOFI; i++)
            bacchette[i] = true;
    }

    // metodo per aggiungere un filosofo in coda, usato
dal main per riempire la coda all'inizio
    public void aggiungi(Filosofi f) {
        queue.add(f);
        System.out.println("aggiunto filosofo " + f.id);
    }

    // metodi in mutua esclusione
    // torna true se ci sono altri pasti da servire
    public synchronized boolean ancoraPasti() {
        return (pasti > 0);
    }

    // raccolgo le due bacchette in contemporanea
    public synchronized void raccogli(Filosofi filosofo)
throws InterruptedException {
        //System.out.println("\n" + queue + "\n");
        // decrementa il numero dei pasti disponibili,
appena un filosofo arriva in raccogli sicuramente mangia
prima o poi
        pasti--;
        int id = filosofo.id;
        // attendi quando c'è qualche bachetta occupata o
quando il filosofo non è il primo della coda di attesa
        while(!bacchette[id] || !bacchette[(id + 1) %
Filosofi.FILOSOFI] || filosofo != queue.peek()) {
            System.out.println("        filosofo " +
filosofo.id + " attende");
            wait();
        }
        // appena entrambe le bacchette sono libere
```

```
prendile, settale a false
        System.out.println("        filosofo " +
filosofo.id + " raccoglie le bacchette");
        bacchette[id] = false;
        bacchette[(id + 1) % Filosofi.FILOSOFI] = false;
        // rimuovi il filosofo che non è stato bloccato ed
era in testa
        queue.poll();
    }

    // deposito le bacchette
    public synchronized void deposita(Filosofi filosofo) {
        int i;
        int id = filosofo.id;
        // rilascia le bacchette, settale a true
        bacchette[id] = true;
        bacchette[(id + 1) % Filosofi.FILOSOFI] = true;
        // il filosofo che ha finito di mangiare si
rimette in fondo alla coda
        queue.add(filosofo);
        // sblocca tutti i thread
        notifyAll();
    }
}
```

implementare il monitor 'Spazio' come una classe Java il cui costruttore prende in input le dimenzioni x,y dello spazio di gioco (es. `Spazio(10,10)`) e con i tre metodi sopra descritti. Utilizzare il programma di test riportato qui sotto.

```
import java.util.HashSet;
import java.util.Set;

public class Test extends Thread {
  private static final int numAgenti =10;   // numero
agenti
  private static final int x=10, y=10;      // dimensione
spazio di gioco

  private final int num;    // id dell'agente
```

```
  private final Spazio s;   // monitor spazio di gioco
  private int my_x,my_y;

  // costruttore: salva id, monitor e posizione iniziale
dell'agente
  Test(int num, Spazio s) {
    this.num = num;
    this.s = s;
    this.my_x = numAgenti -1 - num;
    this.my_y = 0;
  }

  public void run() {
    try {
      code();
    } catch (InterruptedException e) {
      System.out.println("Agente numero "+num+"
interrotto!!");
    }
  }

  // codice dei thread
  void code() throws InterruptedException {
    int dx, dy, i, j, a;

    if (num == numAgenti) {
        // questo thread stampa solo la situazione e
controlla interferenze
        // vedere nel ramo 'else' per il codice degli
agenti
        boolean done = false;
        Set <Integer>check = new HashSet <Integer>();

        // attende che tutti gli agenti siano
registrati
        sleep(1000);

        // controlla la registrazione
        for (i=0;i<x;i++)
            if (s.getAgent(i,0) != numAgenti-1-i) {
                System.out.println("Errore: l'agente " + i
```

```
+ " non e' registrato correttamente");
                System.exit(1);
            }

        // stampa e controlla la situazione ogni secondo
        while(!done){
            // stampa
            check.clear(); // svuota l'insieme di id
            synchronized(s) {
                System.out.println("====");
                for (j=0;j<y;j++) {
                    for (i=0;i<x;i++) {
                        a = s.getAgent(i,j);
                        if (a == -1)
                            System.out.print(". ");
                        else {
                            System.out.print("."+a);
              if (check.contains(a)) {
                System.out.println("Errore: l'agente " + a
+ "e' presente 2 volte!");
                System.exit(1);
              } else // lo aggiungiamo
                               check.add(a);
                        }
                    }
                    System.out.println(".");
                    // se tutti gli agenti sono
sull'ultima linea
                    if (j+2 == y && check.isEmpty())
                        done = true;
                }
            }

            // controlla che non ci siano overlap: tutti
gli agenti devono essere presenti
            if (check.size() != numAgenti) {
                // manca qualche agente!
                System.out.println("Errore: sono presenti
solo gli agenti " + check);
                System.exit(1);
            }
```

```
            // se tutti gli agenti sono sull'ultima riga
controlla che siano nell'ordine
            // giusto
            if (done) {
                for (i=0;i<x;i++)
                    if (s.getAgent(i,y-1) != i) {
                        System.out.println("Errore:
l'agente " + i + " non e' posizionato correttamente");
                        System.exit(1);
                    }
            } else
                sleep(1000);
        }
        // se non siamo usciti prima il test e' superato
        System.out.println("Tutti gli agenti sono
posizionati correttamente");

    } else {
        // questo sono gli agenti

        // si registra
        s.register(num,my_x,my_y);

        // il giocatore e' pronto attende che tutti si
registrino
        System.out.println("Agente numero "+num+"
registrato!");
        sleep(500);

        // qui avvengono le mosse
        while (my_y != y-1) {
            sleep(1000); // si muovono tutti assieme ogni
secondo

            // calcola la mossa
            dy = 1; // scende sempre di una posizione
            if (num == my_x)
                dx = 0; // posizione giusta, non si muove
orizzontalmente
            else {
```

```
                dx = (num - my_x) / Math.abs(num - my_x);
// -1,1 a seconda della necessita'
            }

            // prova a fare la mossa
            if (!s.move(num,my_x,my_y,dx,dy)) {
                System.out.println("Agente numero "+num+
": la posizione non corrisponde!!");
                System.exit(1);
            }

            // aggiorna la posizione
            my_x += dx;
            my_y += dy;

        }

      }
    }

    public static void main(String argv[]) throws
InterruptedException {
        int j;
        Spazio s = new Spazio(x,y); // crea il monitor

        // crea i thread dei vari colori/numeri
        for (j=0; j<=numAgenti; j++) {
            (new Test(j,s)).start();
          }

    }
}
```

soluzione

```
public class Spazio {
    int matrice[][]; //matrice delle posizioni

    public Spazio(int x, int y){
        matrice = new int[x][y];
        for (int i = 0; i < x; i++) {
```

```
            for (int j = 0; j < y; j++) { //inizializzo
tutte le caselle a -1 (vuote)
                matrice[i][j] = -1;
            }
        }
    }
    /** 'Registra' l'agente n nella posizione x,y.
     *   Questo metodo viene invocato una sola volta
quando gli agenti vengono creati.
     *   Serve per inizializzare lo spazio nella
configurazione iniziale.
     */
    void register(int n, int x, int y){
        this.matrice[x][y] = n;
    }
    /** Muove l'agente n dalla posizione x,y alla
posizione x+dx, y+dy.
     *  Il valori dx e dy sono nel range [-1,1] in quanto
gli agenti si spostano di una sola posizione.
     *  Se la posizione è occupata l'agente attende. Il
metodo ritorna false nel caso l'agente n non sia nella
posizione x,y.
     * @throws InterruptedException
     */
    synchronized boolean move(int n, int x, int y, int dx,
int dy) throws InterruptedException{
        if(dx < -1 || dx > 1 || dy < -1 || dy > 1 ||
matrice[x][y] != n){
            return false;  //se il passo è troppo lungo o
l'agente n non corrisponda alle coordinate ritorno false
        }
        while(matrice[x+dx][y+dy] != -1){ //finché la
casella su cui voglio muovermi è occupata, aspetto
            wait();
        }
        matrice[x+dx][y+dy] = n; //mi sposto sulla nuova
casella
        matrice[x][y] = -1; //svuoto la casella precedente
        notify();
        return true;
    }
```

```
    /** Ritorna l'id dell'agente nella posizione x,y.
     *  Se la posizione è vuota ritorna -1.
     *   Questo metodo viene usato per stampare la
situazione ed eseguire test.
     */

    int getAgent(int x, int y){
        return this.matrice[x][y]; //mi basta ritornare la
casella (le vuote sono già a -1)
    }
```

implementare il monitor 'Porta' con i tre metodi sopra descritti

```
public class Porta{
    //aperta mi serve per indicare quando il mutex della
porta si sblocca
    private boolean aperta;
    //numComponenti indica quanti componenti della squadra
devo aspettare prima di far entrare il capitano di
un'altra squadra
    private int numComponenti;
    //indica il colore della squadra che sta passando; a
-1 indica che sta aspettando l'arrivo di un capitano che
setti il colore
    private int colore_squadra=-1;
    //ultimo_entrat indica il numero del giocatore entrato
per ultimo in modo da sapere qual'è il prossimo giocatore
che deve entrare
    private int ultimo_entrato=-1;

    public Porta(int numComponenti){
        this.numComponenti=numComponenti;
        aperta=false;
    }
   //il main apre la porta e quindi sblocco tutti i thread
```

```
in attesa nella coda di wait
   public synchronized void apri(){
       aperta=true;
       notifyAll();
   }
   //se la porta non è stata aperta dal main faccio
attendere il thread nella coda wait
   public synchronized void attendi()throws
InterruptedException{
       while(!aperta){
          wait();

       }


   }

   //metodo che gestisce l'ingresso corretto delle squadre
in modo ordinato per numero , con il colore del primo
capitano che vi accede
   public synchronized void entra(int col,int num)throws
InterruptedException{
       //in questo while controllo che il giocatore abbia
il numero successivo a quello entrato precedentemente
(ultimo_entrato!=num-1)
       // e che sia del colore della squadra che sto
facendo entrare
while( (num>0 && (ultimo_entrato!=num-1 || colore_squadra!
=col) ) || (num==0 && colore_squadra!=-1) )
    wait();
if(num==0)
    colore_squadra=col;
           else
                //altrimenti se il giocatore non rispetta
l'ordine o il colore non lo faccio passare
                wait();

       }
       //il blocco di istruzioni che segue viene eseguito
solo da quei giocatori che hanno il numero corretto e il
colore corretto
```

```
        //per la squadra che sto attualmente facendo
passare

        //devo controllare quando arriva l'ultimo giocatore
affinchè possa settare le impostazioni iniziali
        if(ultimo_entrato+1==numComponenti-1){
           colore_squadra=-1; //con questa impostazione
ristabilisco la contesa tra i capitani per settare il
proprio colore a colore_squadra
           ultimo_entrato=-1; //con questa impostazione
reinizializzo il contatore per mantenere l'ordine corretto

        }
        else
          //se il giocatore non è l'ultimo incremento
ultimo_entrato per mantenere l'ordine
          ultimo_entrato++;
        //ogni volta che mi passa il giocatore con il
colore e numero atteso sblocco tutti i thread bloccati
        //tra cui vi sarà presente il prossimo giocatore da
far passare
        notifyAll();
    }
}
```