

# Esercizio sul Monitor in Java

17 maggio 2012

# Il Bar dello Stadio

In uno stadio e' presente un unico bar a disposizione di tutti i tifosi che assistono alle partite di calcio. I tifosi sono suddivisi in due categorie: tifosi della squadra **locale**, e tifosi della squadra **ospite**.

Il bar ha una capacita' massima **NMAX**, che esprime il numero massimo di persone (tifosi) che il bar puo' accogliere contemporaneamente.

Per motivi di sicurezza, nel bar **non e' consentita la presenza contemporanea di tifosi di squadre opposte**.

Il bar e' gestito da un **barista** che puo' decidere di chiudere il bar in qualunque momento per effettuare la **pulizia** del locale. Al termine dell'attivita' di pulizia, il bar verra' riaperto.

Durante il periodo di chiusura non e' consentito l'accesso al bar a nessun cliente.

Nella fase di chiusura, potrebbero essere presenti alcune persone nel bar: in questo caso il barista attendera' l'uscita delle persone presenti nel bar, prima di procedere alla pulizia.

Utilizzando il linguaggio Java, si rappresentino i clienti e il barista mediante thread concorrenti e si realizzi una politica di sincronizzazione basata sul concetto di monitor che tenga conto dei vincoli dati, e che inoltre, **nell'accesso al bar dia la precedenza ai tifosi della squadra ospite**.

# Impostazione

## Quali thread?

- barista
- cliente ospite
- cliente locale

## Risorsa condivisa?

il bar

-> definiamo la classe **Bar**, che rappresenta il monitor allocatore della risorsa

# Struttura dei threads: ospite

```
public class ClienteOspite extends Thread
{
    Bar m;

    public ClienteOspite(Bar M){this.m = M;}

    public void run()
    {
        try{
            m.entraO();
            System.out.print( "Ospite: sto consumando...\n");
            sleep(2);
            m.esciO();

        }catch(InterruptedException e){}

    }
}
```

# Struttura dei threads: locale

```
public class ClienteLocale extends Thread
{
    Bar m;

    public ClienteLocale(Bar M){this.m =M;}

    public void run()
    {
        try{
            m.entraL();
            System.out.print( "Locale: sto consumando...\n");
            sleep(2);
            m.esclL();

        }catch(InterruptedException e){}

    }
}
```

# Struttura dei threads: barista

```
public class Barista extends Thread
{
    Bar m;

    public Barista(Bar M){this.m =M;}

    public void run()
    {
        try{ while(1)
            {
                m.inizio_chiusura();
                System.out.print( "Barista: sto pulendo...\n");
                sleep(8);
                m.fine_chiusura();
                sleep(10);
            }
        }catch(InterruptedException e){}

    }
}
```

## Progetto del *monitor* bar:

```
import java.util.concurrent.locks.*;

public class Bar
{ //Dati:
  private final int N=20; //costante che esprime la capacita` bar
  private final int Loc=0; //indice clienti locali
  private final int Osp=1; //indice clienti ospiti
  private int[] clienti; //clienti[0]: locali; clienti[1]: ospiti
  private boolean uscita;// indica se il bar è chiuso, o sta per chiudere
  private Lock lock= new ReentrantLock();
  private Condition clientiL= lock.newCondition();
  private Condition clientiO= lock.newCondition();
  private Condition codabar= lock.newCondition(); //coda barista
  private int[] sospesi;

  //Costruttore:
  public Bar() {
    clienti=new int[2];
    clienti[Loc]=0;
    clienti[Osp]=0;
    sospesi=new int[2];
    sospesi[Loc]=0;
    sospesi[Osp]=0;
    uscita=false;
  }
```

```
//metodi public:
```

```
public void entraL() throws InterruptedException
{ lock.lock();
  try
  {      while ((clienti[Osp] != 0) ||
              (clienti[Loc] == N) ||
              uscita ||
              (sospesi[Osp] > 0) )
      {      sospesi[Loc]++;
              clientiL.await();
              sospesi[Loc]--;
      }
      clienti[Loc]++;
  } finally{ lock.unlock(); }
  return;
}
```



```

public void entra0() throws
    InterruptedException
{ lock.lock();
  try
  {
      while ((clienti[Loc] != 0) ||
              (clienti[Osp] == N) ||
              uscita )
          {sospesi[Osp]++;
            clientiO.await();
            sospesi[Osp]--;}
      clienti[Osp]++;
      } finally{ lock.unlock();}
  return;
}

```

```
public void esciO() throws InterruptedException
{  lock.lock();
    try
    {clienti[Osp]--;
      if (uscita && (clienti[Osp]==0))
          codabar.signal();
      else if (sospesi[Osp]>0)
          clientiO.signal();
      else if ((clienti[Osp]==0) && (sospesi[Loc]>0))
          clientiL.signalAll();
    } finally{  lock.unlock();}
}
```

```
public void esciL () throws InterruptedException
{  lock.lock();
  try
  {clienti[Loc]--;
    if (uscita && (clienti[Loc]==0))
      codabar.signal();
    else if ((sospesi[Osp]>0) && (clienti[Loc]==0))
      clientiO.signalAll();
    else if ((sospesi[Loc]>0) && (sospesi[Osp]==0))
      clientiL.signal();
  } finally{  lock.unlock();}
}
```

```

public void inizio_chiusura() throws InterruptedException
{ lock.lock();
  try
  { uscita=true;
    while ((clienti[Loc]>0) || (clienti[Osp]>0))
      codabar.await();
  } finally{ lock.unlock();}
}

```

```

public void fine_chiusura() throws InterruptedException
{ lock.lock();
  try
  { uscita=false;
    if (sospesi[Osp]>0) clientiO.signalAll();
    else if (sospesi[Loc]>0) clientiL.signalAll();
  } finally{ lock.unlock();}
}
} // fine classe Bar

```

# Programma di test

```
import java.util.concurrent.*;

public class Bar_stadio {
    public static void main(String[] args) {

        System.out.println("Benvenuti allo stadio!");
        int i;
        Bar M=new Bar();
        ClienteOspite []CO= new ClienteOspite[50] ;
        ClienteLocale []CL= new ClienteLocale[50] ;
        Barista B=new Barista(M);
        for(i=0;i<50;i++)
        {
            CO[i]=new ClienteOspite(M);
            CL[i]=new ClienteLocale(M);
        }
        for(i=0;i<50;i++)
        {
            CO[i].start();
            CL[i].start();
        }
        B.start();
    }
}
```