

# Esercitazione 10

## **Gruppo LZ**

Accesso a risorse condivise tramite  
Monitor in Java

# Agenda

## **Esempio**

L'album delle figurine: gestione di una risorsa condivisa da più thread, con politica prioritaria

Esercizio 1 – URP

Esercizio 2 – URP con priorità statica

Esercizio 3 – URP con priorità dinamica

---

# Esempio - La collezione di figurine (1/3)

Una casa editrice vuole realizzare un **sito web** dedicato ai collezionisti di figurine dell'album “**Campionato di calcio 2022-2023**”.

L'album è composto da **N=100 diverse figurine**, ognuna individuata univocamente da un id intero  $[0, 99]$ ; tra di esse:

- **30** sono classificate come **figurine rare** (id da 0 a 29);
- le rimanenti **70** sono classificate come **figurine normali** (id da 30 a 99).

Il sito offre un servizio che permette ad ogni utente collezionista di effettuare **scambi di figurine**.

A questo scopo il sistema gestisce un **deposito di figurine**, nel quale, **per ogni diversa figurina vi può essere più di un esemplare**.

# Esempio - La collezione di figurine (2/3)

Il meccanismo di scambio, è regolamentato come segue:

- Si può scambiare solo **una figurina alla volta**, effettuando una **richiesta di scambio** con le seguenti regole:
  - ogni **utente U** che desidera una figurina **A** può ottenerla, se a sua volta offre un'altra figurina **B**;
  - in seguito a una richiesta di scambio, il **sistema aggiunge la figurina B** all'insieme delle figurine disponibili e **successivamente verifica se esiste almeno una figurina A** disponibile:
    - se **A è disponibile**, essa viene assegnata all'utente U, che può così continuare la propria attività;
    - se **A non è disponibile**, l'utente U viene messo in attesa.

# Esempio - La collezione di figurine (3/3)

Si progetti la politica di gestione del servizio di scambio che tenga conto delle specifiche date e che inoltre soddisfi il seguente vincolo:

le richieste di **utenti che offrono figurine rare abbiano la precedenza sulle richieste di utenti che offrono figurine normali.**

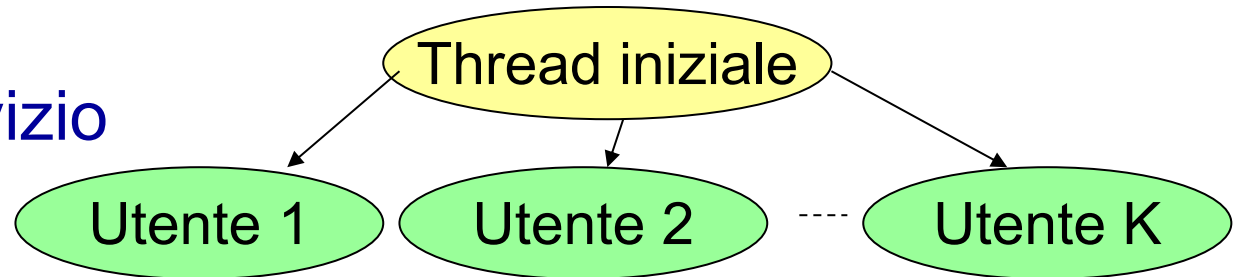
## Ad esempio:

1. Il thread TA chiede 7 offrendo 3[RARA]; 7 non disponibile → **TA attende**
2. Il thread TB chiede 7 offrendo 50[NORM]; 7 non disponibile → **TB attende**
3. 7 diventa disponibile => deve essere **attivato TA** (perchè offre una rara, quindi è più prioritario).

# Impostazione

## Quali thread?

- il thread iniziale
- K utenti del servizio



## Qual è la risorsa comune?

- Deposito delle Figurine
- associamo al Deposito un "**monitor**", che controlla gli accessi in base alla specifica politica di accesso. La sincronizzazione viene realizzata mediante **variabili condizione**.

# Thread Collezionista

```
public class Collezionista extends Thread{  
    private Monitor M;  
    private int offerta, richiesta, N;
```

riferimento  
al monitor

```
    public Collezionista(monitor m, int N){  
        this.M=m;  
        this.N=N;
```

numero di  
figurine diverse.  
Se ci atteniamo  
alle specifiche,  
N=100.

```
    }  
    public void run(){  
        try { while (true){  
            <definizione di offerta e richiesta>  
            M.scambio(offerta, richiesta); //entry call  
            Thread.sleep(...);  
        }} catch (InterruptedException e) {}  
    }  
}
```

# Monitor – Deposito figurine

## Stato del Deposito:

**Figurine disponibili:** vettore di  $N=100$  interi (uno per ogni figurina della collezione)

```
private int[] FIGURINE = new int[N];;
```

**Dove:** `FIGURINE[i]` è il numero di esemplari disponibili della figurina  $i$ .

**(Hp:** inizialmente 1 per ogni figurina)

**Convenzione adottata:**

**Se  $i < 30$ ,** si tratta di una **figurina rara**;

**Se  $i \geq 30$ ,** si tratta di una **figurina comune**.



# Monitor – Deposito figurine

**Lock per la mutua esclusione:**

```
private Lock lock = new ReentrantLock();
```

**Condition.** Per la sospensione dei thread in attesa di una figurina, definiamo 2 condition (una per ogni livello di priorità):

```
private Condition rare= ...;  
//thread sospesi che hanno offerto figurine rare  
private Condition normali=...;  
// thread sospesi che hanno offerto figurine  
normali
```

**Contatori dei thread sospesi in ogni coda:**

```
private int[] sospRare = new int[N];  
private int[] sospNormali = new int[N];  
//devo sapere chi è sospeso in attesa di quella  
specifica figurina dopo aver consegnato una  
figurina rara o una normale
```

# Monitor – Deposito figurine

```
public class Monitor {  
    private final int N=100; //numero totale di figurine  
    private final int maxrare=30;  
    private int[] FIGURINE; //figurine disponibili  
    private Lock lock = new ReentrantLock();  
    private Condition rare = lock.newCondition();  
    private Condition normali = lock.newCondition();  
    private int[] sospRare;  
    private int[] sospNormali;  
  
    public Monitor() {...} //Costruttore  
    public void scambio(int off,int rich) //metodo entry:  
        throws InterruptedException {...}  
}
```

politica di  
sincronizzazione

# Soluzione

```
import java.util.Random;

public class Collezionista extends Thread{
    private Monitor M;
    private int offerta, richiesta, max;

    public Collezionista(Monitor m, int NF, String name){
        this.M=m; this.max=NF;
        this.setName(name);
    }

    public void run(){
        int op, cc, somma;
        try { while (true)
            {
                offerta= r.nextInt(max);
                do { richiesta= r.nextInt(max);
                }while(richiesta==offerta);
                M.scambio(offerta, richiesta);
                Thread.sleep(250);
            }
        } catch (InterruptedException e) { }
    }
}
```

# Soluzione: monitor

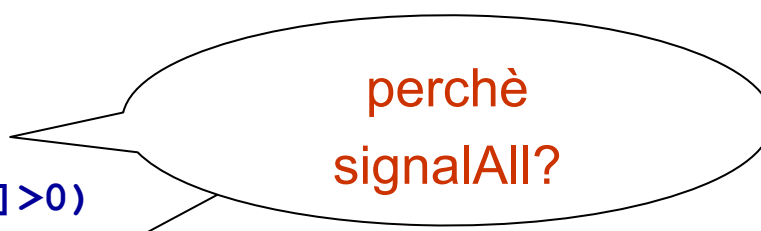
```
import java.util.concurrent.locks.*;

public class Monitor{ //Dati:
    private int N; //numero totale di figurine
    private final int maxrare;
    private int[] FIGURINE= new int[N];; //figurine disponibili

    private Lock lock= new ReentrantLock();
    private Condition rare= lock.newCondition();
    private Condition normali= lock.newCondition();
    private int[] sospRare= new int[N];
    private int[] sospNormali= new int[N];

    public Monitor(int N ) {
        int i;
        for(i=0; i<N; i++) {
            FIGURINE[i]=1;//valore arbitrario
            sospRare[i]=0;
            sospNormali[i]=0;
        }
        maxrare=N/100*30; }
}
```

```
//metodi "entry":
public void scambio(int off, int rich) throws InterruptedException {
    lock.lock();
    try{
        FIGURINE[off]++;
        if (sospRare[off]>0)
            rare.signalAll();
        else if (sospNormali[off]>0)
            normali.signalAll();
        if (off < maxrare) //ha offerto una figurina rara
            while (FIGURINE[rich]==0){
                sospRare[rich]++;
                rare.await();
                sospRare[rich]--; }
        else //ha offerto una normale
            while (FIGURINE[rich]==0 || sospRare[rich]>0){
                sospNormali[rich]++;
                normali.await();
                sospNormali[rich]--; }
        FIGURINE[rich]--; // tolgo la figurina scambiata
    } finally{ lock.unlock();}
    return;
}
```



perchè  
signalAll?

# Commenti finali

- La soluzione prevede solo 2 condition (rare, normali), ma le condizioni di sincronizzazione possibili sono  $2 \times 100$ : per ogni categoria (rare/normali), ogni thread si sospende in attesa di una particolare figurina.
- Per evitare le signalAll (poco efficienti) si potrebbero definire  $2 \times 100$  condition:

```
private Condition []rare=new Condition[N];  
//code thread che hanno offerto rare  
private Condition []normali=new Condition[N];  
//code thread hanno offerto normali
```

# Esercizio 1 (1/4)



Si consideri un Ufficio Relazioni con il Pubblico (URP) di un quartiere cittadino.

L'ufficio offre agli utenti i seguenti servizi:

- un servizio di erogazione/rinnovo **documenti di identità**
- un servizio di **cambi di residenza**

A questo scopo presso l'ufficio vi è una **sala di aspetto** e **NS sportelli** dedicati agli utenti; ad ogni sportello è costantemente disponibile un impiegato che può fornire entrambi i servizi.

# Esercizio 1 (2/4)



La sala d'aspetto dispone di MAX sedie e pertanto può accogliere al massimo MAX utenti.

Ogni utente  $U$ , una volta entrato nella sala d'aspetto, richiede il servizio  $S$  di cui necessita attraverso un distributore di biglietti.

Dopo un'eventuale attesa,  $U$  verrà autorizzato ad accedere a un particolare sportello  $K$  per ottenere il servizio  $S$ .



# Esercizio 1 (3/4)

Pertanto, il comportamento dell'utente U sarà il seguente:

1. **entra** nella sala d'aspetto, occupando uno dei MAX posti disponibili;
2. <si reca al distributore di biglietti>
3. **richiede** il servizio S, ottenendo (eventualmente dopo un intervallo di attesa) l'autorizzazione ad accedere a un particolare sportello K. Una volta autorizzato, U occuperà lo sportello K, liberando contestualmente un posto nella sala d'attesa.
4. <sosta per un intervallo di tempo arbitrario presso lo sportello K>
5. **esce** dall'URP, liberando lo sportello K.

# Esercizio 1 (4/4)

Quando saranno usciti tutti gli utenti, il thread iniziale dovrà **stampare il numero totale di cittadini serviti**, il numero totale di **erogazioni di documenti di identità** effettuati e il numero totale di **cambi di residenza** erogati.

Realizzare un'applicazione concorrente in Java basata sul monitor nella quale gli Utenti siano rappresentati da thread concorrenti.

La sincronizzazione tra i thread dovrà tenere conto di tutti i vincoli dati.

# Impostazione

Quali thread?

- thread iniziale
- Utenti

Quale risorsa comune?

- l'ufficio URP

Associamo all'ufficio un "**monitor**", che controlla gli accessi in base alla politica di accesso.

La sincronizzazione viene realizzata mediante **variabili condizione**.

# Struttura dei thread

```
public class Utente extends Thread{  
    private Monitor M;  
  
    public Utente(...){ // costruttore..  
    }  
  
    public void run(){  
        ...  
        M.entraSala();  
        <si reca al totem>  
        K=M.richiedeSportello(tiposerv S);  
        <sosta allo sportello>  
        M.esce(int K);  
        <va a casa>  
    }  
}
```

# Monitor: gestisce l'URP

## Variabili di stato:

**Sportelli[NS]:** tiene traccia dell'occupazione di ogni sportello.

**SportelliLiberi:** sportelli liberi

**PostiLiberi:** posti liberi nella sala d'aspetto

# Sincronizzazione

Un thread **Utente** si può sospendere sia in **entrata**, sia quando **richiede** uno sportello:

→ quante condition?

Tenere conto che:

- quando si libera un posto in sala, si potrà riattivare un utente sospeso in attesa di un posto nella sala;
- quando si libera uno sportello, si potrà riattivare un utente sospeso in attesa dello sportello

# Esercizio 2

Si estenda l'esercizio 1, prevedendo che, **nell'acquisizione degli sportelli**, gli utenti che desiderano effettuare **un erogazione/rinnovo di documento di identità** abbiano la **precedenza sui cambi di residenza**.

→ cosa cambia riguardo alla sincronizzazione?

- è necessario poter riattivare sempre il processo più prioritario tra quelli in attesa per lo sportello...

# Esercizio 3

Si estenda l'esercizio 1, prevedendo che, nell'acquisizione degli sportelli, vengano favoriti gli utenti che richiedono il servizio (rinnovo documenti o cambio residenza) che, fino a quel momento, **è stato erogato meno volte**.

- 👉 cosa cambia riguardo alla sincronizzazione rispetto all'es.2?
- ❑ le categorie di thread da trattare in base alla politica prioritaria sono ancora due, ma la priorità viene decisa a **run-time**...