



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercitazione: Sincronizzazione in Java mediante lock e variabili condizione (Parte 1)

- Gli oggetti *lock* del package *java.util.concurrent*
- Il problema del *Produttore-Consumatore*
- Esercizi (*Coke machine, Contatore*)

Prof.ssa Patrizia Scandurra

Corso di laurea
in Ingegneria Informatica

La sincronizzazione in Java

- Meccanismi di sincronizzazione

1. **I semafori** -- API *java.util.concurrent*
2. **Monitor mediante *lock* su oggetti** e i metodi *wait / notify / notifyAll* -- a livello di linguaggio
 - Detta anche *sincronizzazione indiretta / diretta*
3. **Monitor mediante lock e variabili condizione**
— API *java.util.concurrent*



Sui meccanismi di sincronizzazione...

Due aspetti da considerare:

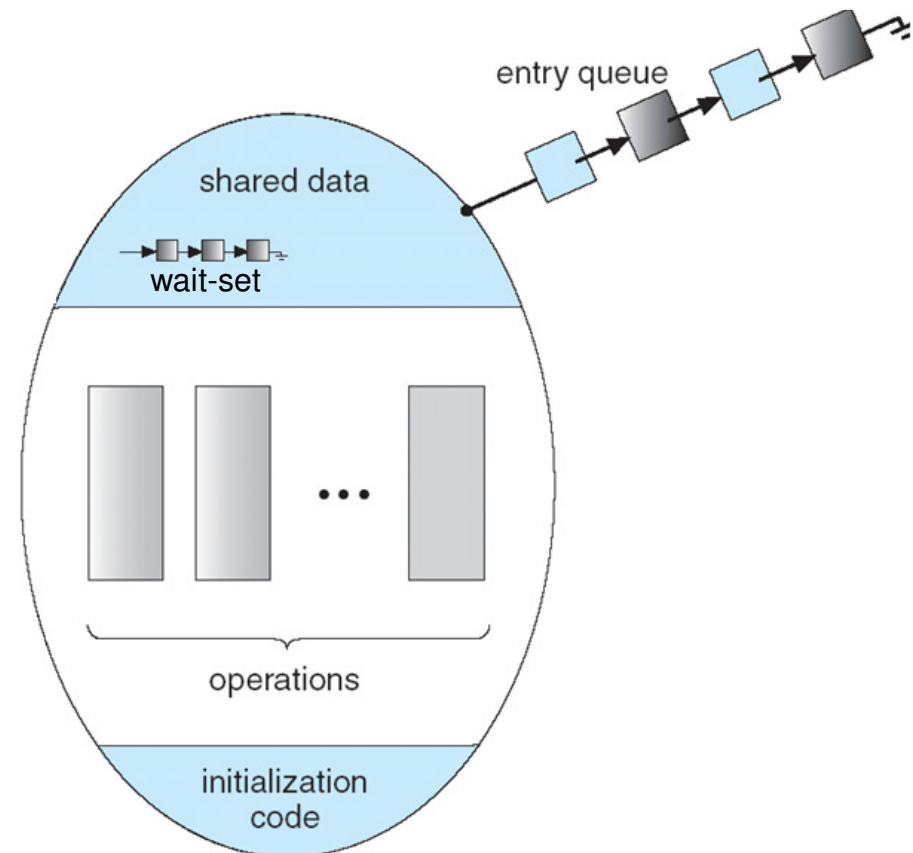
1. Come il meccanismo realizza la **mutua esclusione** dei thread per accedere in modo esclusivo alle risorse?
 - Risorse fisiche o logiche (ad esempio, oggetti Java)?
2. Come il meccanismo realizza la **cooperazione** tra i thread?

Visione schematica dei monitor

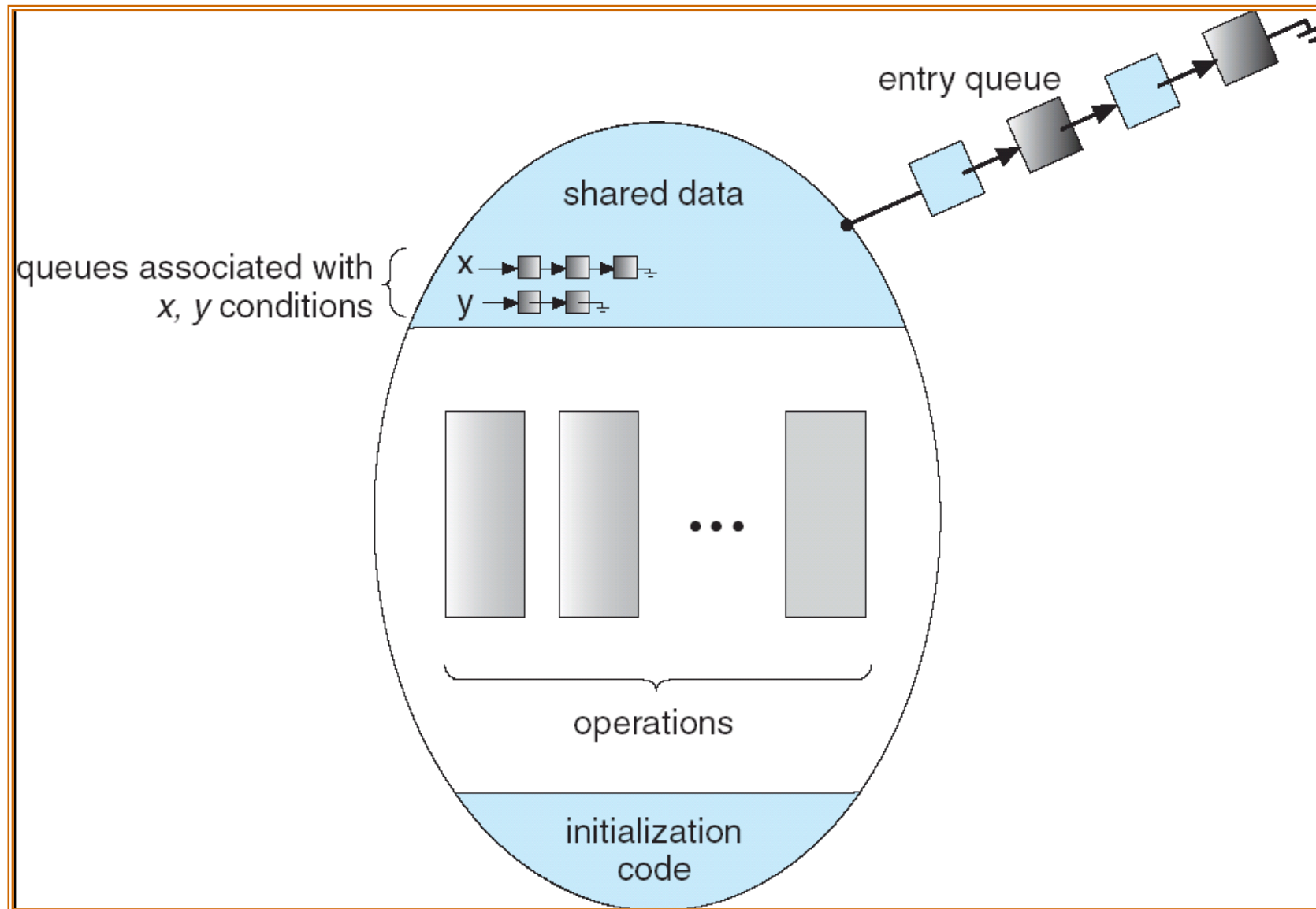
MONITOR è un costrutto linguistico

- **in Java:** classi con metodi sincronizzati (sincr. diretta/indiretta)

```
class monitor-name {  
  // variable declarations  
  synchronized public ...op1(...){  
    ...  
    while (...){  
      ...  
      wait();  
    }  
    ...  
    notify();  
  }  
  synchronized public ... op2(...){  
    ...  
  }  
  ...  
}
```



Monitor con le variabili “condizioni”



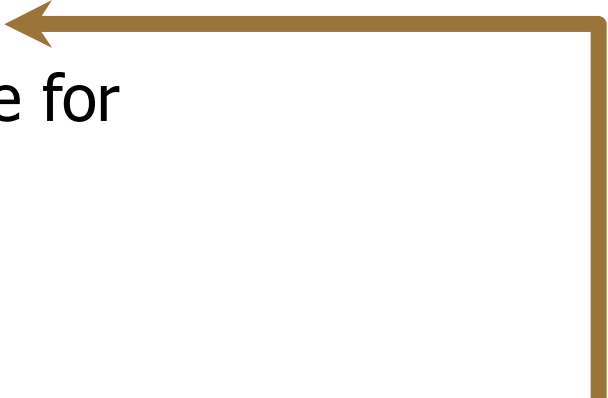
Mutua esclusione mediante lock e variabili condizione

Per realizzare i monitor con variabili condizione, da Java 5.0 si può con oggetti **lock** e **variabili condizione** di **java.util.concurrent.locks**

- mediante le interfacce **Lock** e **Condition**
- la classe **ReentrantLock**

L'interfaccia Lock

```
package java.util.concurrent.locks;  
public interface Lock {  
    public void lock(); // Wait for the lock to be acquired  
    public void unlock();  
    public Condition newCondition();  
    // Create a new condition variable for  
    // use with the Lock.  
    ...  
}
```



- Le variabili condizione sono associate ad un lock (oggetto *Lock*)!
- Non è possibile mettersi in attesa su una variabile condizione se non ho prima acquisito l'oggetto Lock a cui è associata

La classe ReentrantLock

```
package java.util.concurrent.locks;  
public class ReentrantLock implements Lock {  
    public ReentrantLock();  
    ...  
    public void lock();  
    public void unlock();  
    public Condition newCondition();  
    // Create a new condition variable and  
    // associated it with this lock object.  
}
```


L'interfaccia Condition

```
package java.util.concurrent.locks;  
public interface Condition {  
    public void await()  
        throws InterruptedException;  
    // Atomically releases the associated lock  
    // and causes the current thread to wait.  
    public void signal();  
    // Wake up one waiting thread.  
    public void signalAll();  
    // Wake up all waiting threads.  
    ...  
}
```

Uso dei lock di Java.util.concurrent

L'uso dei metodi **lock()** e **unlock()** su un oggetto lock **per garantire la *mutua esclusione*** è a carico del programmatore

Protocollo di accesso alla sezione critica:

//creazione del lock

Lock key = new ReentrantLock();

key.lock(); //Acquisizione del lock

try {

... // *sezione critica*: accedi alle risorse protette dal lock

}

finally { key.unlock(); //Rilascio del lock

}

Le variabili “condizioni” di Java.util.concurrent -- creazione

- Per **creare una variabile condizione** occorre prima creare un oggetto lock **ReentrantLock** e poi invocare su di esso il metodo **newCondition()**:

```
Lock key = new ReentrantLock();  
Condition condVar = key.newCondition();
```

- Dopo, sarà possibile invocare su “condVar” i metodi **await()**, **signal()** e **signalAll()**

Lock e variabili “condizioni” di Java.util.concurrent – uso combinato

```
//creazione del lock e delle variabili condizione x e y
Lock key = new ReentrantLock();
Condition x = key.newCondition();
Condition y = key.newCondition(); ...
key.lock();
try {
    //sezione critica: accedi alle risorse protette dal lock key e
    // a x,y, ecc. con wait e/o signal()/signalAll
    while (condX) {
        ...
        x.await(); //attesa su X
    }

    ...
    if (condY) y.signal(); //sveglia altri thread in Y
}
finally { key.unlock(); }
```

Protocollo di accesso alla sezione critica: meccanismi 2. e 3. a confronto

Rule 1: Instead of using `synchronized` keyword, use `Lock.lock()` and `Lock.unlock()`

Existing API

```
Object monitorObject;  
synchronized(monitorObject){  
    //critical section  
}
```

New API

```
Lock lockObject;  
try{  
    lockObject.lock();  
    //critical section  
}  
finally{  
    lockObject.unlock()  
}
```

blocco sincronizzato (similmente con i **metodi sincronizzati**, contrassegnando come `synchronized` i metodi della classe di `monitorObject`)

Cooperazione all'interno della sezione critica – meccanismi 2. e 3. a confronto

Rule 2: Instead of using `wait()` and `notify()` in the critical section use `await()` and `signal()` on condition variables

Existing API

```
////////////////////////////////////
//Inside your critical section://
////////////////////////////////////
boolean somecondition; //evaluate your wait criteria
while(somecondition){
    wait();
    //re-evaluate somecondition
}
```

```
////////////////////////////////////
//Inside of your critical section://
////////////////////////////////////
boolean someothercondition; //evaluate your notify criteria
if(someothercondition) {
    notify();
}
```

New API

```
////////////////////////////////////
//Outside your critical section://
////////////////////////////////////
Condition conditionVariable = lockObject.newCondition();
////////////////////////////////////
//Inside your critical section://
////////////////////////////////////
boolean somecondition; //evaluate your wait criteria
while(somecondition){
    conditionVariable.await();
    //re-evaluate somecondition
}
```

```
////////////////////////////////////
//Inside your critical section://
////////////////////////////////////
boolean someothercondition; //evaluate your signal criteria
if(someothercondition) {
    conditionVariable.signal();
}
```

Esempio del Producer-Consumer

- Vedi **<boundedbuffer>** con lock e variabili condizione

Esercizio: Coke Machine

Implementare in Java una soluzione al problema di prelevare lattine di coca cola da una macchinetta e di rifornirla nel caso in cui rimanga vuota. In particolare:

- Definire le classi per i thread con ruolo Utente e Rifornitore.
- Definire la classe CokeMachine contenente lattine di coca cola e i metodi :

preleva (...), eseguito dal generico utente per prelevare una lattina dalla macchinetta

rifornisci (...), eseguito dal fornitore del servizio per caricare la macchinetta nel caso in cui rimane vuota

- Assumere che inizialmente la macchinetta è piena. A scelta , il primo utente a trovare la macchinetta vuota o l'utente che preleva l'ultima lattina deve segnalare al fornitore che la macchinetta è vuota; il fornitore a seguito di tale comunicazione provvederà al rifornimento.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class CokeMachine {

    private static final int N = 50; //Capacità della macchinetta

    private int count; //Numero effettivo di lattine presenti nella macchinetta
    private final Lock lock = new ReentrantLock(); //Lock per la mutua esclusione

    //Condition variables

    // .... <COMPLETARE>.... }

}
```


Esercizio contatore sincronizzato

Implementare un programma in Java che permetta a più thread di operare contemporaneamente su un contatore (oggetto condiviso di classe *Counter*) usando gli oggetti lock e le variabili condizione per la mutua esclusione e la cooperazione sull'oggetto condiviso.

In particolare:

- Un contatore (oggetto della classe *Counter*) è inizialmente a zero ed è in grado di contare fino a $N=10$;
- Thread differenti devono poter incrementare/decrementare di una unità dallo stesso “contatore” (stesso oggetto di classe *Counter*) invocando il metodo `increment()`/`decrement()` della classe *Counter*; le operazioni di incremento/decremento devono essere possibili solo se mantengono rispettivamente il conteggio non superiore a N /non negativo.
- Per testare il corretto funzionamento dell'oggetto di classe *Counter*, si preveda una classe con metodo `main()` che istanzi un certo numero di thread figli: alcuni thread con ruolo “TaskI” che all'interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e incrementare il contatore; e altri thread con ruolo “TaskD” che all'interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e decrementare il contatore.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercitazione: Sincronizzazione in Java mediante lock e variabili condizione (Parte 2)

- Esercizio (BAR)

Prof.ssa Patrizia Scandurra

Corso di laurea
in Ingegneria Informatica

Esercizio: BAR

- In uno stadio è 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 capacità massima **NMAX**, che esprime il numero massimo di persone (tifosi) che il bar può accogliere contemporaneamente.
- Per motivi di sicurezza, nel bar **non è consentita la presenza contemporanea di tifosi di squadre opposte**.
- Il bar è gestito da un **barista** che può decidere di chiudere il bar in qualunque momento per effettuare la **pulizia** del locale. Al termine dell'attività di pulizia, il bar viene riaperto.
- Durante il periodo di chiusura non è consentito l'accesso al bar a nessun cliente. Nella fase di chiusura, potrebbero essere presenti alcune persone nel bar: in questo caso il barista attende 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 tra i thread 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**.