

# **I Thread in Java**

## **parte 2:**

### **Sincronizzazione**

# Sincronizzazione: `wait` e `notify`

***wait set***: coda di thread associata ad ogni oggetto, inizialmente vuota.

- I thread entrano ed escono dal ***wait set*** utilizzando i metodi `wait()` e `notify()`.
- `wait` e `notify` possono essere invocati da un thread **solo all'interno di un blocco sincronizzato o di un metodo sincronizzato** (e` necessario il *possesso* del lock dell'oggetto).

# wait, notify, notifyall

**wait** comporta il rilascio del lock, la sospensione del thread ed il suo inserimento in **wait set**. Al risveglio (tramite notify o notifyall) il thread riacquisirà automaticamente il lock.

**(NB. throws InterruptedException)**

**notify** comporta l'estrazione di un thread da **wait set** ed il suo inserimento in **entry set**. [Cioè: il thread risvegliato riacquisce automaticamente il lock.]

**notifyall** comporta l'estrazione di **tutti** i thread da **wait set** ed il loro inserimento in **entry set**.

**NB:** **notify** e **notifyall** non provocano il rilascio del lock: → i thread risvegliati devono attendere che il lock sia libero (Politica **signal&continue**).

---

```
//Esempio: mailbox con capacita`=1
```

```
public class Mailbox{  
    private int contenuto;  
    private boolean pieno=false;  
  
    public synchronized int preleva()  
    {    try{  
        while (pieno==false)  
            wait();  
        pieno=false;  
        notify();  
    }catch(InterruptedException e){}  
    return contenuto;  
}
```

```
    public synchronized void deposita(int valore)  
    {    try{  
        while (pieno==true)  
            wait();  
        contenuto=valore;  
        pieno=true;  
        notify();  
    }catch(InterruptedException e){}  
    }  
}
```



**Quali  
problemi???**

# Problemi

- La presenza di una sola coda rende necessario dopo ogni inserimento ed ogni prelievo, il risveglio di TUTTI i processi (produttori e consumatori) sospesi.

```
//Esempio: mailbox con capacita`=1
public class Mailbox{
    private int contenuto;
    private boolean pieno=false;

    public synchronized int preleva()
    {    try{
        while (pieno==false)
            wait();
        pieno=false;
        notifyAll();
    }catch (InterruptedException e){}
    return contenuto;
}

    public synchronized void deposita(int valore)
    {    try{
        while (pieno==true)
            wait();
        contenuto=valore;
        pieno=true;
        notifyAll();
    }catch (InterruptedException e){}
}
}
```

---

```

//Mailbox di capacita` N
public class Mailbox {
    private int[] contenuto;
    private int contatore, testa, coda;

    public mailbox(){ //costruttore
        contenuto = new int[N];
        contatore = 0;
        testa = 0;
        coda = 0;
    }
    public synchronized int preleva () {
        int elemento;
        while (contatore == 0)
            wait();
        elemento = contenuto[testa];
        testa = (testa + 1)%N;
        --contatore;
        notifyAll();
        return elemento;
    }
    public synchronized void deposita (int valore){
        while (contatore == N)
            wait();
        contenuto[coda] = valore;
        coda = (coda + 1)%N;
        ++contatore;
        notifyAll();
    }
}

```

---

# wait&notify

## *Principale limitazione :*

- unica coda (wait set) per ogni oggetto sincronizzato

→ non e` possibile sospendere thread su code differenti!

Problema superato nelle versioni più recenti di Java (versione 5.0 ) tramite la possibilità utilizzare le **variabili condizione**.



# Variabili condizione

- Nelle versioni più recenti di Java (**Java™ 2 Platform Standard Ed. 5.0**) esiste la possibilità utilizzare le **variabili condizione**. Ciò è ottenibile tramite l'uso un'apposita interfaccia (definita in `java.util.concurrent.locks`) :

```
public interface Condition{  
    //Public instance methods  
    void await ()throws InterruptedException;  
    void signal();  
    void signalAll();  
}
```

- dove i metodi `await`, `signal`, e `signalAll` sono del tutto equivalenti ai metodi `wait`, `signal` e `signalAll` riferiti in genere alle variabili condizione
  - La semantica di `signal` è “**signal\_and\_continue**”
-

# Mutua esclusione: lock

- Oltre a metodi/blocchi synchronized, la versione 5.0 di Java prevede la possibilità di utilizzare esplicitamente il concetto di *lock*, mediante l'interfaccia (definita in `java.util.concurrent.locks`) :

```
public interface Lock{  
    //Public instance methods  
    void lock();  
    void unlock();  
    Condition newCondition();  
}
```

# Uso di Variabili Condizione

**Ad ogni variabile condizione deve essere associato un lock, che:**

- ▣ al momento della sospensione del thread mediante **await** il lock verra` liberato;
- ▣ al risveglio di un thread, il lock verra` automaticamente rioccupato.

➔ La creazione di una condition deve essere effettuata mediante in metodo **newCondition** del lock associato ad essa.

In pratica, **per creare un oggetto Condition** :

```
Lock lockvar=new Reentrantlock(); //Reentrantlock è una
                                   // classe che implementa
                                   // l'interfaccia Lock
Condition C=lockvar.newCondition(); //C è una condition
                                   // associata al lock lockvar
```

# Monitor

Con gli strumenti visti, possiamo quindi definire **classi** che rappresentano monitor:

## Dati:

- ❑ le **variabili condizione**
- ❑ **1 lock per la mutua esclusione** dei metodi "entry", da associare a tutte le variabili condizione
- ❑ variabili interne: stato delle risorse gestite

## Metodi:

- ❑ metodi public ("entry")
- ❑ metodi privati
- ❑ costruttore

# Esempio: gestione di buffer circolare

```
public class Mailbox
{ //Dati:
  private int[] contenuto;
  private int contatore, testa, coda;
  private Lock lock= new ReentrantLock();
  private Condition non_pieno=
    lock.newCondition();
  private Condition non_vuoto=
    lock.newCondition();

  //Costruttore:
  public Mailbox( ) {
    contenuto=new int[N];
    contatore=0;
    testa=0;
    coda=0;
  }
```

```
//metodi "entry":
```

```
public int preleva() throws InterruptedException
{ int elemento;
  lock.lock();
  try
  {   while (contatore== 0)
        non_vuoto.await();
      elemento=contenuto[testa];
      testa=(testa+1)%N;
      --contatore;
      non_pieno.signal ( );
  } finally{lock.unlock();}
  return elemento;
}
```

```
public void deposita (int valore) throws
    InterruptedException
{ lock.lock();
  try
  {   while (contatore==N)
        non_pieno.wait();
        contenuto[coda] = valore;
        coda=(coda+1)%N;
        ++contatore;
        non_vuoto.signal( );
    } finally{ lock.unlock();}
}
```

# Programma di test:

```
public class Produttore extends Thread
{   int messaggio;
    Mailbox m;
    public  Produttore(Mailbox M){this.m =M;}
    public void run()
    {   while(1)
        {   <produci messaggio>
            m.deposita(messaggio);
        }
    }
}
```

```
public class Consumatore extends Thread
{   int messaggio;
    Mailbox m;
    public  Consumatore(Mailbox M){this.m =M;}
    public void run()
    {   while(1)
        {   messaggio=m.preleva();
            <consuma messaggio>
        }
    }
}
```



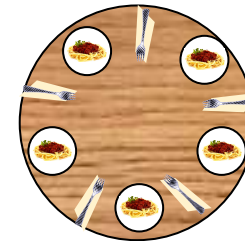
```
public class BufferTest{  
  
    public static void main(String args[])  
    {  
        Mailbox M=new Mailbox();  
        Consumatore C=new Consumatore(M);  
        Produttore P=new Produttore(M);  
        C.start();  
        P.start();  
        ...  
    }  
}
```

# **I filosofi a cena**

(E. Dijkstra, 1965)

# Il problema

- 5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette.
- Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:
  - una fase in cui **pensa**,
  - una fase in cui **mangia**.



Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock.

# Soluzione

Ogni filosofo verifica se entrambe le forchette sono disponibili:

- ❑ in caso affermativo, acquisisce le due forchette (in modo atomico);
- ❑ in caso negativo, aspetta.

[in questo modo non si puo` verificare deadlock: non c'e` possesso e attesa]

# Realizzazione

## Quali thread?

- ▣ filosofo

## Risorsa condivisa?

la tavola apparecchiata

-> definiamo la classe **tavola**, che  
rappresenta il monitor allocatore delle  
forchette

# Struttura Filosofo<sub>i</sub>

```
public class filosofo extends Thread
{   tavola m;
    int i;
    public  filosofo(tavola M, int id){this.m =M;this.i=id;}

    public void run()
    {   try{
        while(true)
        {   System.out.print("Filosofo "+ i+" pensa....\n");
            m.prendiForchette(i) ;
            System.out.print("Filosofo "+ i+" mangia....\n");
            sleep(8) ;
            m.rilasciaForchette(i) ;
            sleep(100) ;
        }
    }catch(InterruptedException e){}

    }
}
```

# Monitor

```
public class tavola
{ //Costanti:
    private final int NF=5;      // costante: num forchette/filosofi
    //Dati:
    private int []forchette=new int[NF]; //num forchette disponibili
    per ogni filosofo i
    private Lock lock= new ReentrantLock();
    private Condition []codaF= new Condition[NF]; //1 coda per ogni
    filosofo i
    //Costruttore:
    public tavola( ) {
        int i;
        for(i=0; i<NF; i++)
            codaF[i]=lock.newCondition();
        for(i=0; i<NF; i++)
            forchette[i]=2;
    }
    // metodi public e metodi privati:...}
```

# Metodi public

```
public void prendiForchette(int i) throws  
    InterruptedException  
{ lock.lock();  
    try  
    {  
        while (forchette[i] != 2)  
            codaF[i].await();  
  
        forchette[sinistra(i)]--;  
        forchette[destra(i)]--;  
  
    } finally{ lock.unlock();}  
    return;  
}
```



```
public void rilasciaForchette(int i) throws
    InterruptedException
{ lock.lock();
  try
  {
    forchette[sinistra(i)]++;
    forchette[destra(i)]++;
    if (forchette[sinistra(i)]==2)
        codaF[sinistra(i)].signal();
    if (forchette[destra(i)]==2)
        codaF[destra(i)].signal();

    } finally{ lock.unlock();}
  return;
}
```

## Metodi privati

```
int destra(int i)
{ int ret;
  ret=(i==0? NF-1:(i-1));
  return ret;
}
```

```
int sinistra(int i)
{ int ret;
  ret=(i+1)%NF;
  return ret;
}
```

# Programma di test

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

public class Filosofi {
    public static void main(String[] args) {
        int i;
        tavola M=new tavola();
        filosofo []F=new filosofo[5];
        for(i=0;i<5;i++)
            F[i]=new filosofo(M, i);
        for(i=0;i<5;i++)
            F[i].start();
    }
}
```