Esercitazione 4
Programmazione Concorrente in Java

I threads in Java

- Ogni programma Java contiene almeno un *singolo thread*, corrispondente all' esecuzione del metodo *main()* sulla *JVM*.
- E' possibile creare dinamicamente nuovi thread *attivando concorrentemente* le loro esecuzioni all' interno del programma.

Due possibilita` di creazione:

1. Thread come oggetti di sottoclassi della classe **`Thread`**
2. Thread come oggetti di classi che implementano l' interfaccia **`runnable`**

Primo metodo:
Thread come oggetti di sottoclassi della classe `Thread`

- I threads sono oggetti che derivano dalla **classe Thread** (fornita dal package **java.lang).**

- Il metodo **run** della classe di libreria **Thread** definisce *l' insieme di statement Java* che ogni thread (oggetto della classe) eseguirà *concorrentemente* con gli altri thread.

- Nella classe **Thread** l' implementazione del suo metodo **run** è vuota.

- In ogni sottoclasse derivata da **Thread** deve essere ridefinito (*override*) il metodo **run** in modo da fargli eseguire ciò che è richiesto dal programma.

Possibile schema

```
class AltriThreads extends Thread {
  public void run() {
    <corpo del programma eseguito>
    <da ogni thread di questa classe>
    }
}

public class EsempioConDueThreads
{     public static void main (string[] args)
      {  AltriThreads t1=new AltriThreads();
         t1.start();//attivazione del thread t1
         <resto del programma eseguito
            dal thread main>
      }
}
```

- La classe **AltriThread** (estensione di Thread) implenta i nuovi thread *ridefinendo il metodo* **run**.
- La classe EsempioConDueThreads fornisce il **main** nel quale viene creato il thread **t1** come oggetto derivato dalla classe **Thread**.
- Per **attivare** il thread deve essere chiamato il metodo **start()** che invoca il metodo **run()** (il metodo **run()** non può essere chiamato direttamente, ma solo attraverso **start()**).

➔ In questo modo abbiamo creato *due thread concorrenti*:

  - il thread principale, associato al main;
  - il thread t1 creato dinamicamente dal precedente, con l'esecuzione dello statement t1.start() che lancia in concorrenza l'esecuzione del metodo run() del nuovo thread.

E se occorre definire thread che non siano necessariamente sottoclassi di Thread?

Secondo metodo:
Thread come classe che implementa l'interfaccia `runnable`

Interfaccia Runnable: maggiore **flessibilità** → thread come sottoclasse di qualsiasi altra classe

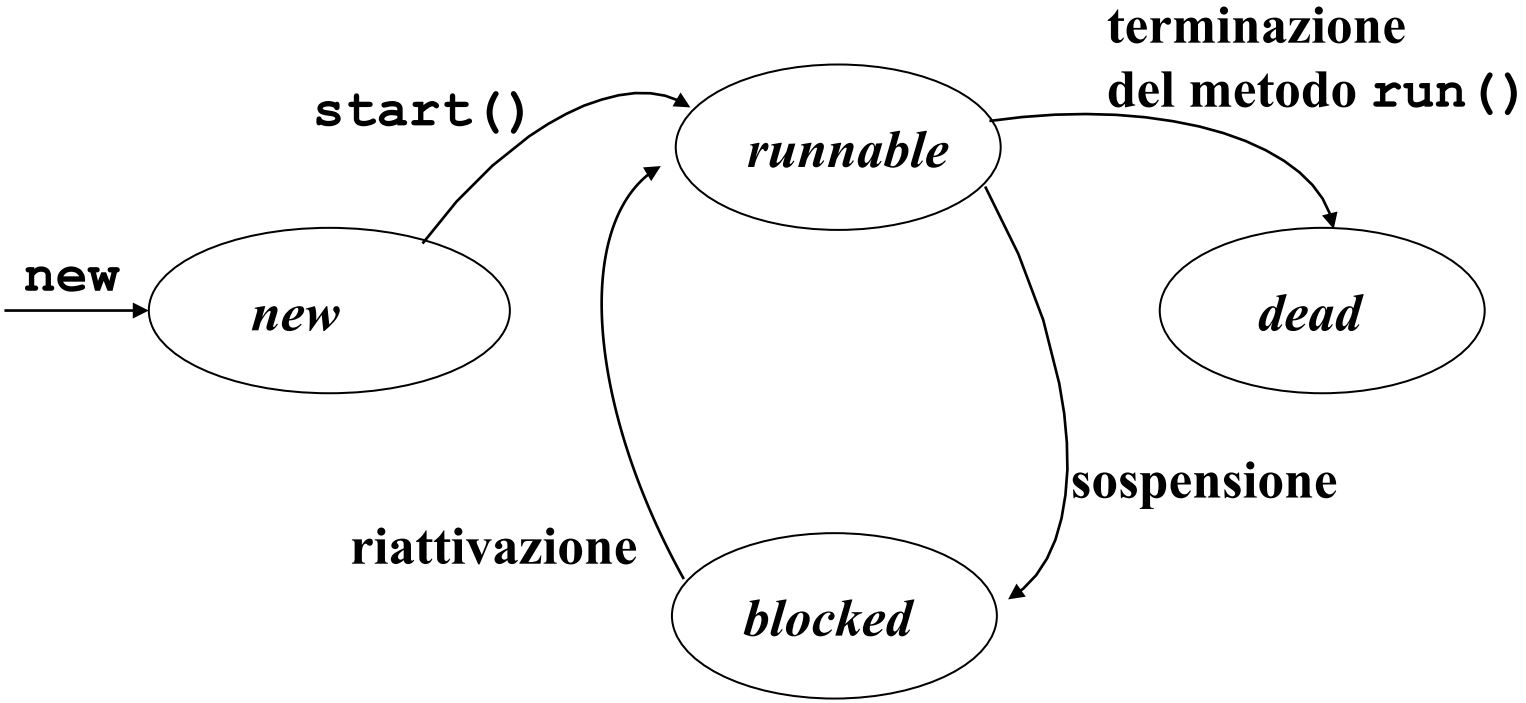
- implementare il metodo **`run()`** nella classe che implementa l'interfaccia **`Runnable`**
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando come parametro l'istanza della classe che si è creata
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Esempio di classe `EsempioRunnable` che *implementa l'interfaccia Runnable* ed è sottoclasse di `MiaClasse`:

```
class EsempioRunnable extends MiaClasse implements Runnable
{
   // non e' sottoclasse di Thread
   public void run()
   {
        for (int i=1; i<=10; i++)
             System.out.println(i + " " + i*i);
   }
}

public class Esempio
{
   public static void main(String args[])
   {
        EsempioRunnable e = new EsempioRunnable();
        Thread t = new Thread(e);
        t.start();
   }
}
```

Grafo di stato di un thread



Priorità e scheduling

- **Preemptive** priority scheduling con priorità fisse (crescenti verso l'alto).
- MIN-PRIORITY, MAX-PRIORITY: costanti definite nella classe thread.
- Ogni thread eredita, all'atto della sua creazione, la priorità del processo padre.
- Metodo **`set-priority`** per modificare il valore della priorità

JVM esegue l'algoritmo di scheduling:

- quando il thread correntemente in esecuzione esce dallo stato *runnable* (sospensione o terminazione);
- quando diventa *runnable* un thread a priorità più alta (preemption).
- La presenza di Time Slicing dipende dall'implementazione.

Metodi per il controllo di thread

- `start()` fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato
- `stop()` ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`
- `suspend()` ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di `resume()`. Non libera le risorse impegnate dal thread (lock inclusi)
- `resume()` ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

- `sleep(long t)` blocca per un ***tempo specificato*** (`time`) l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.
- `join()` ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***
- `yield()` ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in ***coda***

Il problema di `stop()` e `suspend()`

`stop()` e `suspend()` rappresentano azioni "brutali" sul ciclo di vita di un thread → rischio di determinare situazioni **inconsistenti** o di blocco critico (***deadlock***)

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l' interruzione può condurre ad uno ***stato inconsistente*** del sistema

➔ JDK 1.5, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi **`stop()`**, **`suspend()`** **`e`** **`resume()`** (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite gli stumenti specifici per la sincronizzazione (object locks, **`wait()`**, **`notify()`**, **`notifyAll()`** e ***variabili condizione***)

Strumenti di sincronizzazione nel linguaggio Java

Sincronizzazione in Java

Modello a memoria comune:

I threads di una applicazione condividono lo spazio di indirizzamento.

➔ Ogni tipo di interazione tra thread avviene tramite *oggetti* comuni:

- Interazione di tipo ***competitivo*** (*mutua esclusione*): meccanismo degli **objects locks**.
- Interazione di tipo ***cooperativo***:
  - meccanismo **wait-notify**.
  - **variabili condizione**

Mutua esclusione

- Ad ogni oggetto viene associato dalla JVM un **lock** (analogo ad un ***semaforo di mutua escusione***).
- E' possibile denotare alcune sezioni di codice che operano su un oggetto come *sezioni critiche* tramite la parola chiave **`synchronized`**.

➔ Il compilatore inserisce :

- un prologo in testa alla sezione critica per l'**acquisizione del lock** associato all'oggetto.
- un epilogo alla fine della sezione critica per **rilasciare il lock**.

Blocchi synchronized

Con riferimento ad un oggetto x si può definire un blocco di statement come una sezione critica nel seguente modo (**synchronized blocks**):

```
synchronized (oggetto x) {<sequenza di statement>;}
```

Esempio:

```
Object mutexLock= new Object;
   …..
   public void  M( ) {
        <sezione di codice non critica>;
        synchronized (mutexlock){
        < sezione di codice critica>;
        }
      <sezione di codice non critica>;
   }
```

- all'oggetto **`mutexLock`** viene implicitamente associato un lock, il cui valore puo` essere:
  - **libero**: il thread può eseguire la sezione critica
  - **occupato**: il thread viene sospeso dalla JVM in una coda associata a **`mutexLock`** *(entry set)*.

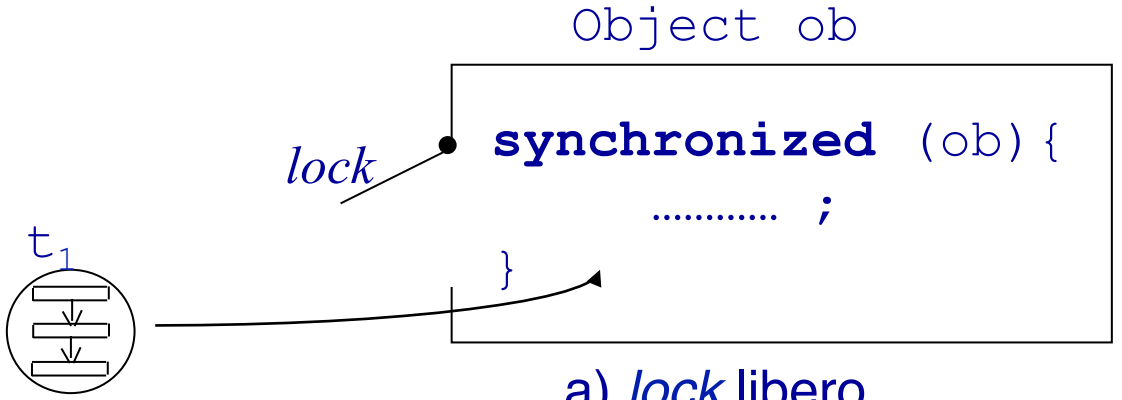
Al termine della sezione critica:

- se *non ci sono thread in attesa*: il lock viene reso libero .
- se *ci sono thread in attesa*: il lock rimane occupato e viene scelto uno di questi .

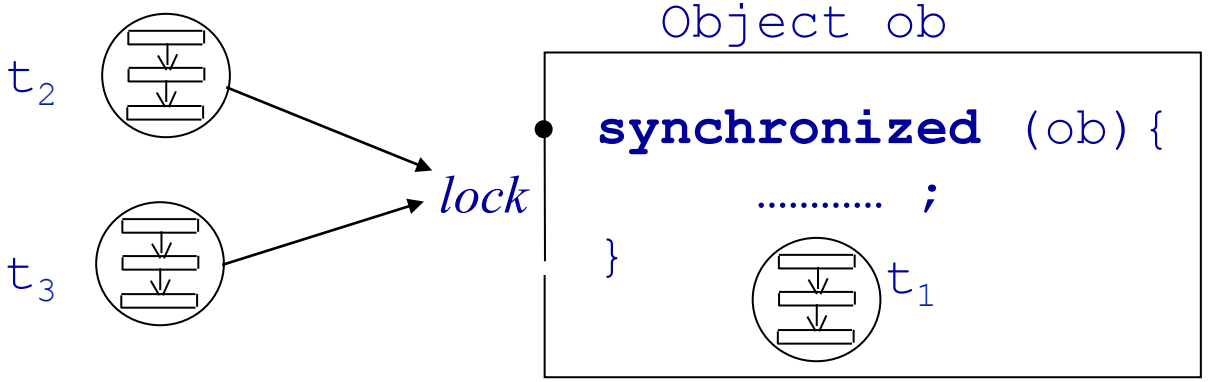
synchronized block

- esecuzione del blocco **mutuamente esclusiva** rispetto:
  - ad altre esecuzioni dello *stesso blocco*
  - all' esecuzione di *altri blocchi* sincronizzati sullo stesso oggetto

Entry set di un oggetto



a) *lock* libero



a) *lock* occupato: t2
e t3 vengono inseriti
nell'entry set di ob

Metodi synchronized

- **Mutua esclusione** tra i metodi di una classe

```
public class intVar {
private int i=0;
public synchronized void incrementa()
{ i ++; }
public synchronized void decrementa()
{i--; }
}
```

- Quando un metodo viene invocato per operare su un oggetto della classe, l'esecuzione del metodo avviene in **mutua esclusione utilizzando il *lock dell'oggetto***.

Esempio: accesso concorrente a un contatore

```
public class competingproc extends Thread
{  contatore r; /* risorsa condivisa */
   int T; // incrementa se tipo=1; decrementa se tipo=-1

   public  competingproc(contatore R,  int tipo)
   {    this.r=R;
        this.T=tipo;
   }

   public void run()
   {    try{
        while(true)
        {  if (T>0)   r.incrementa();
           else if (T<0)  r.decrementa();
        }
        }catch(InterruptedException e){}
   }
}
```

```
public class contatore {
    private int C;

    public contatore(int i)
    {     this.C=i;}

    public synchronized void incrementa()
    {     C++;
          System.out.print("\n eseguito incremento: valore    attuale del contatore:
    "+ C+" ....\n");
    }

    public synchronized void decrementa()
    {     C--;
          System.out.print("\n eseguito decremento: valore
          attuale del contatore: "+ C+" ....\n");
    }
}
```

```
import java.util.*;

public class prova_mutex{ // test

    public static void main(String args[]) {
        final int NP=30;
        contatore C =new contatore(0);
        competingproc []F=new competingproc[NP];
        int i;
        for(i=0;i<(NP/2);i++)
          F[i]=new competingproc(C, 1); // incrementa
        for(i=(NP/2);i<NP;i++)
                F[i]=new competingproc(C, -1); // decrementa
        for(i=0;i<NP;i++)
                F[i].start();
    }

}
```

Semafori in Java

- Nelle versioni precedenti alla 5.0 Java non prevedeva i semafori (tuttavia essi potevano essere facilmente costruiti mediante i meccanismi di sincronizzazione standard *wait* e *notify*).
- Dalla versione 6, è disponibile la classe Semaphore:

```
import java.util.concurrent.Semaphore;
```

Tramite la quale si possono creare semafori, sui quali è possibile operare tramite i metodi:

- `acquire();` `// implementazione di p()`
- `release();` `// implementazione di v()`

Uso di oggetti Semaphore:

Inizializzazione ad un valore K dato:

```
Semaphore s=new Semaphore(k);
```

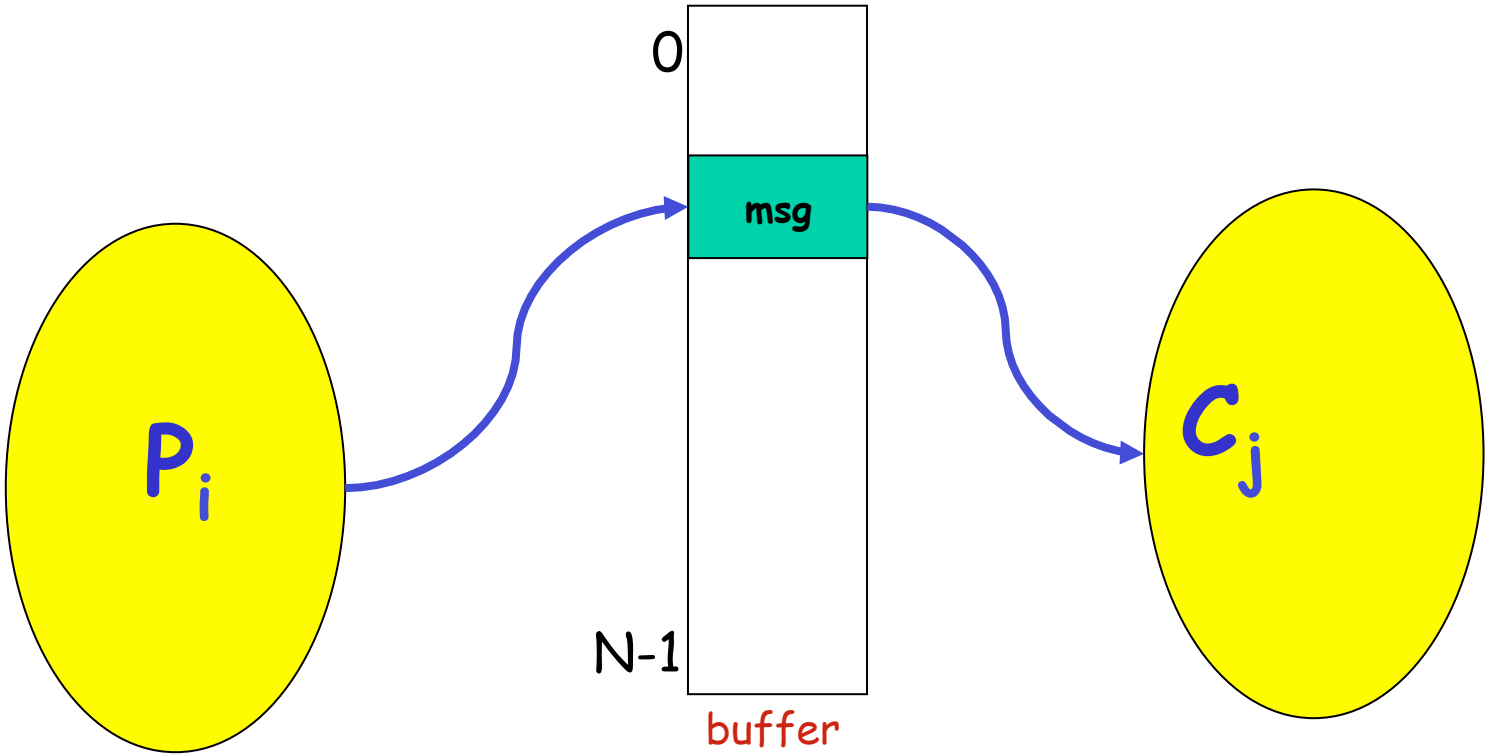
Operazioni: stessa semantica di p e v

```
s.acquire(); // esecuzione di p() su s
```

```
s.release(); // esecuzione di v() su s
```

NB Esistono altre operazioni che estendono la semantica tradizionale del semaforo..

Esempio: produttori e consumatori con buffer di capacità N



HP: Buffer (*mailbox*) limitato di dimensione N

Soluzione con semafori RISORSA

```
public class threadP extends Thread{ //produttori
   int i=0;
   risorsa r; //oggetto che rappresenta il buffer condiviso

   public  threadP(risorsa R)
   {   this.r=R;
   }

   public void run()
   {   try{  System.out.print("\nThread PRODUTTORE: il mio
             ID è: "+getName()+"..\n");
             while (i<100)
             { sleep(100);
               r.inserimento(i);
               i++;
               System.out.print("\n"+ getName() +":
               inserito messaggio " +i+ "\n");
             }
       }catch(InterruptedException e){}
   }
}
```

```
public class threadC extends Thread{ //consumatori
   int msg;
   risorsa r;

   public  threadC(risorsa R)
   {   this.r=R;
   }

   public void run()
   {   try{  System.out.print("\nThread CONSUMATORE: il mio
             ID è: "+getName()+"..\n");
             while (true)
             {     msg=r.prelievo();
                   System.out.print("\n"+getName()+"
                   consumatore ha letto il messaggio "+msg
                   + "...\n");
             }
       }catch(InterruptedException e){}
   }
}
```

```
import java.util.concurrent.Semaphore;
public class risorsa { // definizione buffer condiviso
   final int N = 30;     // capacità del buffer
   int lettura, scrittura;//indice di lettura
   int []buffer;
   Semaphore sP; /* sospensione dei Produttori; v.i. N*/
   Semaphore sC; /* sospensione dei Consumatori v.i. 0*/
   Semaphore sM; // semaforo di mutua esclusione v.i. 1

   public  risorsa() // costruttore
   {   lettura=0;
       scrittura=0;
       buffer= new int[N];
       sP=new Semaphore(N); /* v.i. N*/
       sC=new Semaphore(0); /* v.i. 0*/
       sM=new Semaphore(1); // semaforo di mutua esclusione
   }  //continua..
```

```
// ..continua classe risorsa

public void inserimento(int M)
{   try{ sP.acquire();
         sM.acquire(); //inizio sez critica
         buffer[scrittura]=M;
         scrittura=(scrittura+1)%N;
         sM.release(); //fine sez critica
         sC.release();
    }catch(InterruptedException e){}
}

//continua..
```

```
//... Continua

public int prelievo()
  {  int messaggio=-1;
     try{ sC.acquire();
          sM.acquire(); //inizio sez critica
          messaggio=buffer[lettura];
          lettura=(lettura+1)%N;
          sM.release(); //fine sez critica
          sP.release();
     }catch(InterruptedException e){}
     return messaggio;
  }
} // fine classe risorsa
```

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


  public class prodcons{

   public static void main(String args[]) {

     risorsa R = new risorsa();// creaz. buffer
     threadP TP=new threadP(R);
     threadC TC=new threadC(R);

     TC.start();
     TP.start();
   }
}
```

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***.
(NB. throws InterruptedException)

`notify` comporta l'estrazione di un thread da **wait set** ed il suo inserimento in ***entry set***.

`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.

➔ Politica **signal&continue**: il rilascio del lock avviene al completamento del *blocco o del metodo sincronizzato* da parte del thread che ha eseguito la notify.

```
//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){}
   }
}
```

```
//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***.

Monitor in Java: le 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 possibilita` 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();
```

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();
      ...
  }
```

Esercizio 1: semafori Java

Si consideri una **piscina** in un centro termale.

L'acqua della piscina ha proprietà terapeutiche e pertanto è frequentata da persone affette da patologie. In particolare sono previsti due tipi di pazienti:

- Pazienti ortopedici, cioè con patologie ortopediche;
- Pazienti infettivi, cioè con malattie infettive.

I pazienti accedono singolarmente alla piscina, vi permangono per una quantità di tempo arbitraria e successivamente escono dalla piscina.

La piscina ha **una capacità massima MAXP**, che esprime il numero massimo di pazienti che può contenere.

Il regolamento della piscina **vieta la presenza contemporanea** nella piscina di pazienti ortopedici e pazienti infettivi.

Si sviluppi un'applicazione concorrente in Java, che rappresenti pazienti e fisioterapisti con thread concorrenti.

L'applicazione deve, utilizzando i **semafori**, realizzare una politica di sincronizzazione che soddisfi i vincoli dati e che, inoltre, nell'accesso alla piscina **favorisca i pazienti ortopedici rispetto ai pazienti infettivi**.

Impostazione

- 1 piscina
- 2 classi di thread:
    1. Ortopedici
    2. Infettivi

```
public class Ortopedico extends Thread
{  Piscina p
   public  Ortopedico(Piscina P){this.p=P;}
   public void run()
   {    p.entraO();
        <..>
        p.esceO();
   }
}
public class Infettivo extends Thread
{  Piscina p
   public  Infettivo (Piscina P){this.p=P;}
   public void run()
   {    p.entraI();
        <..>
        p.esceI();
   }
}
```

Come usare i semafori?

- Per sincronizzare gli accessi al gestore della piscina (mutua esclusione) -> semaforo SM
- Per mettere in attesa i thread che non possono entrare: semafori privati
  - Semaforo SI per gli infettivi
  - Semaforo SO per gli ortopedici
  - Quale schema di sincronizzazione:
    - Attesa circolare?
    - Passaggio del testimone?

Risorsa condivisa

- Piscina:

```
public class Piscina
{  //Dati:
   private …; // stato della piscina
   private Semaphore sO; /* sospensione Ortopedici*/
   private Semaphore sI; /* sospensione Infettivi*/
   private Semaphore sM; /* per mutua esclusione*/

   public  Piscina() // costruttore
   {    sO=new Semaphore(..); /* v.i */
        sI=new Semaphore(..); /* v.i.*/
        sM=new Semaphore(1); // mutex
        …
   }

   <definizione metodi ingresso e uscita:  entraO, entraI, esceO,
   esceI >
}
```

Classe main

```
public class Test{

  public static void main(String args[])
  {   Piscina P=new Piscina();
      Random rand = new Random();
      Infettivo TI[] = new Infettivo[100];
      Ortopedico TO[] = new Ortopedico[100];
      for (int i = 0; i < 100; i++)
            TI[i] = new Infettivo(P);

      for (int i = 0; i < 100; i++)
            TO[i] = new Ortopedico(P);
  }
```

Esercizio 2

Risolvere l'esercizio 1 utilizzando un metodo basato sul **monitor** (lock e variabili condizione).