I Thread in Java
parte 1

Thread

Un thread è un ***singolo flusso sequenziale*** di controllo all'interno di un processo(task)

Un thread (o processo leggero) è un'unità di esecuzione che ***condivide codice e dati*** con altri thread ad esso associati

Un ***thread***

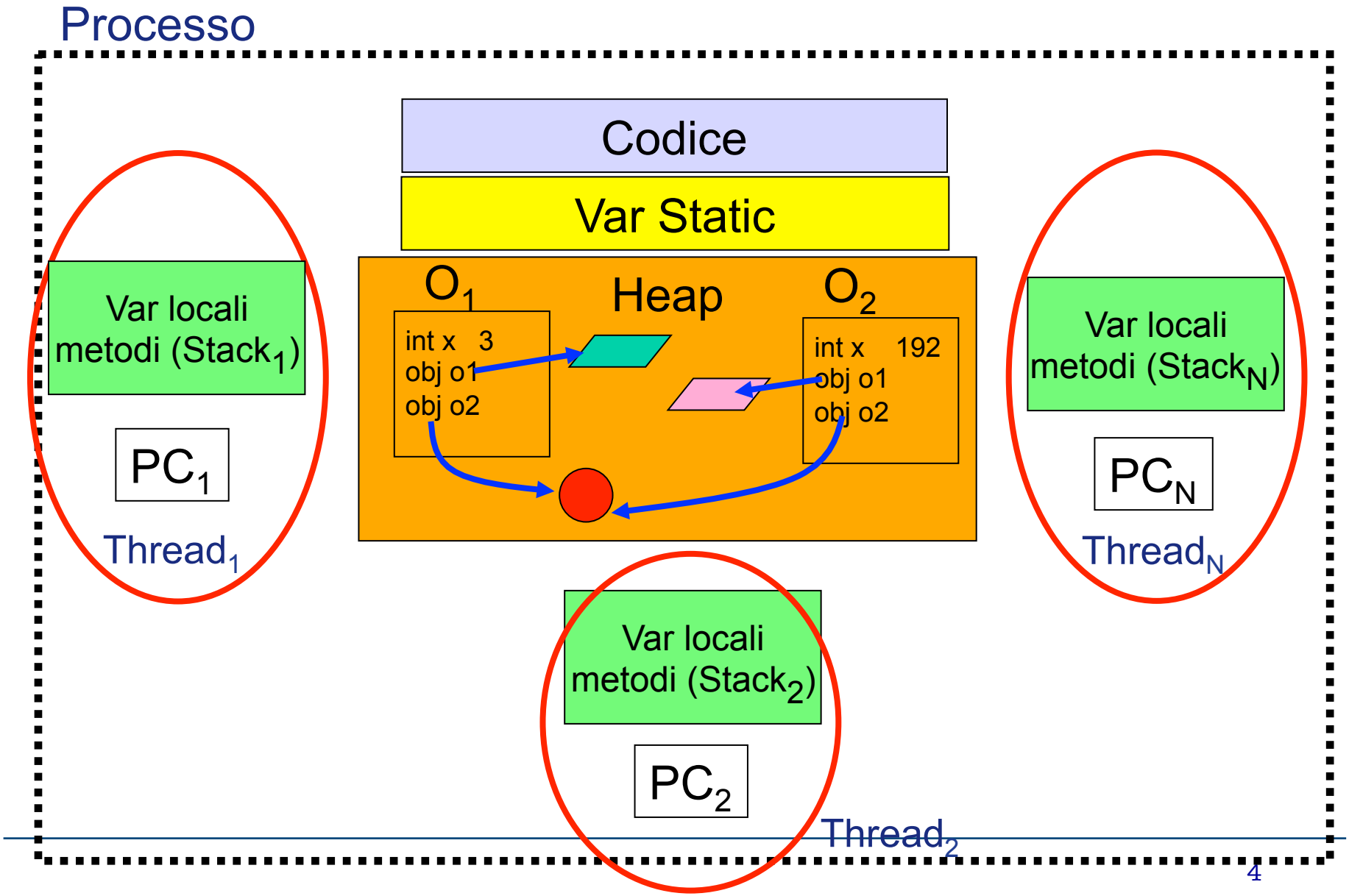
- ***NON*** ha spazio di memoria riservato per dati e heap: tutti i thread appartenenti allo stesso processo condividono lo ***stesso spazio di indirizzamento***
- ha ***stack*** e ***program counter privati***

I threads in Java

- All'esecuzione di ogni programma Java corrisponde un task che 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.

Java Thread


Processo
Codice
Var Static
Heap
O_1
int x 3
obj o1
obj o2
O_2
int x 192
obj o1
obj o2
Var locali metodi ($Stack_1$)
PC_1
$Thread_1$
Var locali metodi ($Stack_N$)
PC_N
$Thread_N$
Var locali metodi ($Stack_2$)
PC_2
$Thread_2$

Java Thread: programmazione

Due modalità per implementare i thread in Java:

1. ***estendendo la classe*** `Thread`
2. ***implementando l'interfaccia*** `Runnable`

1. Thread come oggetti di sottoclassi della classe `Thread`

- I thread sono oggetti che derivano dalla **classe Thread** (fornita dal package **java.lang).**
- Il metodo **run()** della classe di libreria **Thread** definisce *l'insieme di istruzioni Java* che ogni thread (oggetto della classe) eseguirà. (NB: nella classe **Thread** l'implementazione del metodo **run** è vuota).
- In ogni sottoclasse derivata da **Thread** il metodo **run** deve essere **ridefinito** (*override*) specificando all'interno di esso cosa far eseguire ai thread di quella classe.
- Per creare un thread, si deve creare un' istanza della classe che lo definisce tramite **new;** dopo la new il thread esiste, ma **non è ancora attivo**.
- Per attivare un thread si deve invocare il metodo **start()** che a sua volta invoca il metodo **run**()

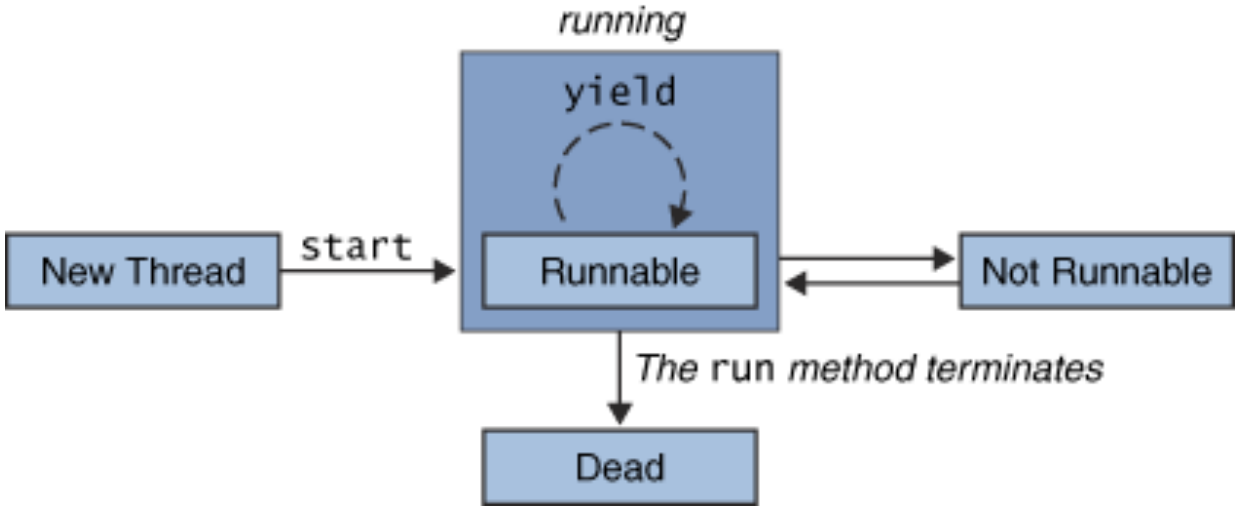
Possibile schema

```
class SimpleThread extends Thread {
  public void SimpleThread()
  { <costruttore> }

  public void run() {
    <sequenza di istruzioni eseguita>
    <da ogni thread di questa classe>
    }
}

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

- La classe **SimpleThread** (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()**).

➔ Abbiamo creato *due thread concorrenti*: il thread principale associato al **main** ed il thread **t1.**

Ciclo di vita di un thread



New Thread

- Subito dopo l'istruzione **new**
- Il costruttore alloca e inizializza le variabili di istanza

Runnable

- Il thread è eseguibile ma potrebbe non essere in esecuzione

Ciclo di vita di un thread

Not Runnable

- Il thread non può essere messo in esecuzione
- Entra in questo stato, ad esempio, quando è in attesa della terminazione di un' operazione di I/O, oppure cerca di accedere ad un metodo "synchronized" di un oggetto bloccato, o dopo aver invocato uno dei seguenti metodi: sleep(), wait(), suspend()
- Esce da questo stato quando si verifica la condizione complementare

Dead

- Il thread giunge a questo stato per "morte naturale" o perché un altro thread ha invocato il suo metodo stop()

Esempio: primo metodo

```
public class SimpleThread extends Thread{

      public SimpleThread(String str)
      {super(str);}

      public void run() {
            for(int i=0; i<10; i++)
            { System.out.println(i+ " " +getName());
              try{
               sleep((int)Math.random()*1000);
               } catch (InterruptedException e){}
            }
              System.out.println("DONE! "+getName());
            }
      }


}
```

Java Thread

```
public class EsempioConDueThreads
{     public static void main(String[] args)
  {   SimpleThread st1= new SimpleThread(”Pippo");
      st1.start();
  }
}
```

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

2. Thread come classi che implementano Runnable

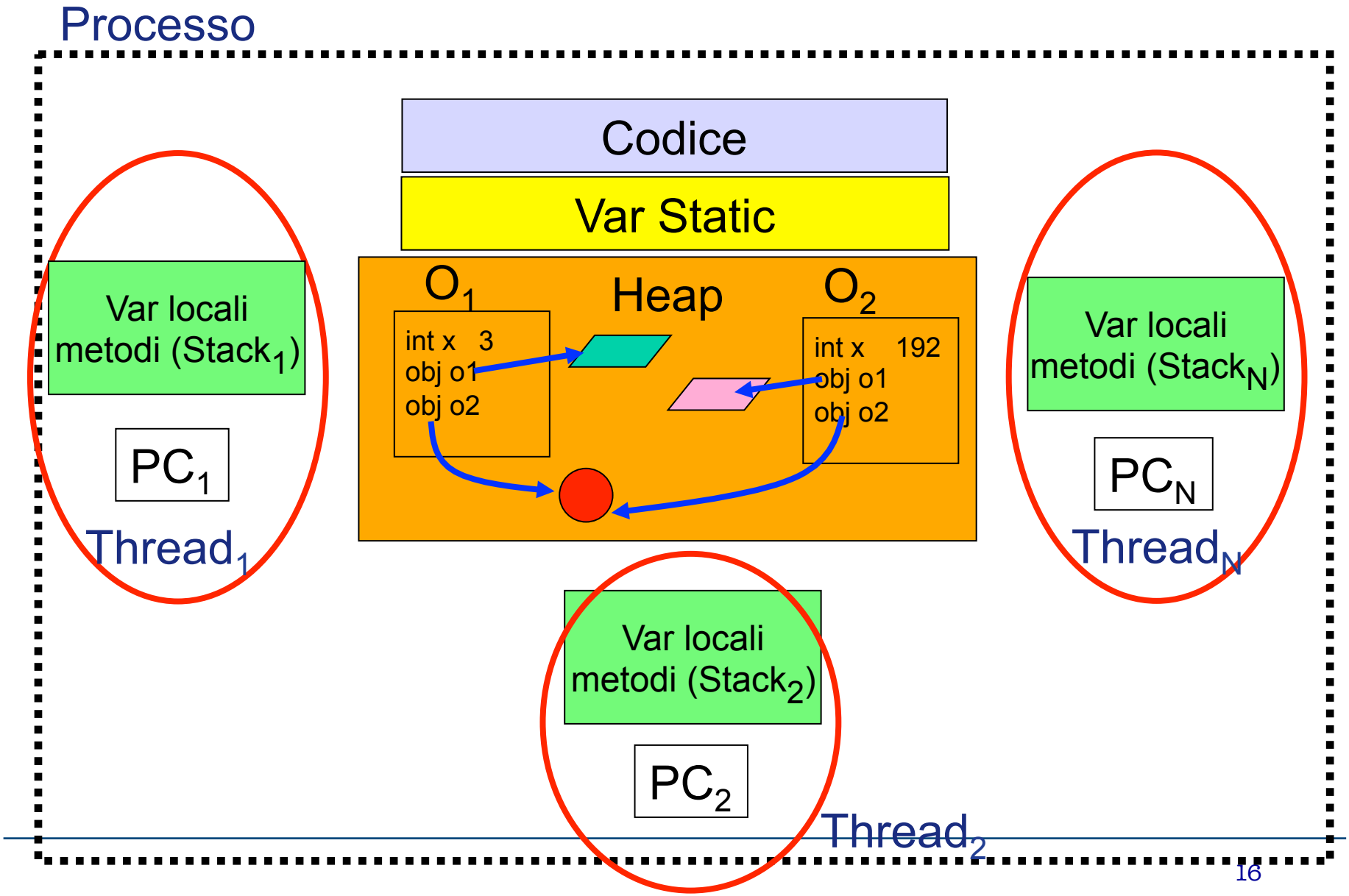
Definizione di thread come classe che implementa **interfaccia `Runnable`**

1. la classe deve ***ridefinire il metodo `run()`***
2. si crea un'istanza di tale classe tramite **`new`**
3. si crea ***un'istanza della classe Thread*** con **`new`**, passandole come ***parametro l'oggetto che implementa `Runnable`***
4. si esegue il thread invocando il metodo **`start()`** ***sull'oggetto con classe Thread creato***

Java Thread

```
class EsempioRunnable extends MiaClasse
                      implements Runnable {
   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();
   }
}
```

Thread


Processo
Codice
Var Static
Heap
O1
O2
int x 3
obj o1
obj o2
int x 192
obj o1
obj o2
Var locali metodi (Stack1)
PC1
Thread1
Var locali metodi (StackN)
PCN
ThreadN
Var locali metodi (Stack2)
PC2
Thread2

Sincronizzazione di thread

Differenti thread ***condividono lo stesso spazio di memoria (heap)***

- è possibile che più thread accedano **contemporaneamente** a uno stesso oggetto, ***invocando un metodo che modifica lo stato dell'oggetto***
- stato **finale** dell'oggetto sarà ***funzione dell'ordine*** con cui i thread accedono ai dati

➡ Servono meccanismi di ***sincronizzazione***

Sincronizzazione in Java

Modello ad ambiente globale
(o a memoria comune)

- 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 -> semafori**
    - **[variabili condizione]**

Mutua esclusione

- Ad ogni oggetto viene associato automaticamente dalla JVM un **lock**, che rappresenta lo stato dell'oggetto (libero/occupato).
- 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**)*.

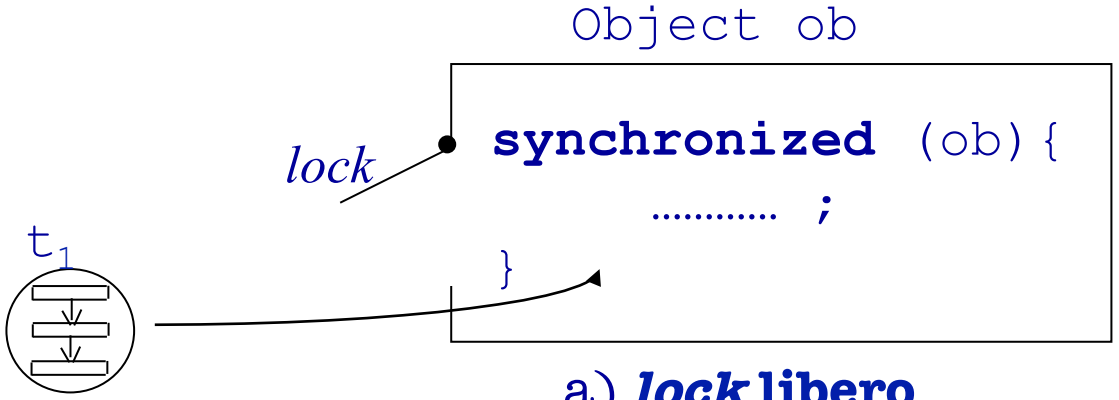
Al termine della sezione critica:

- 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 .

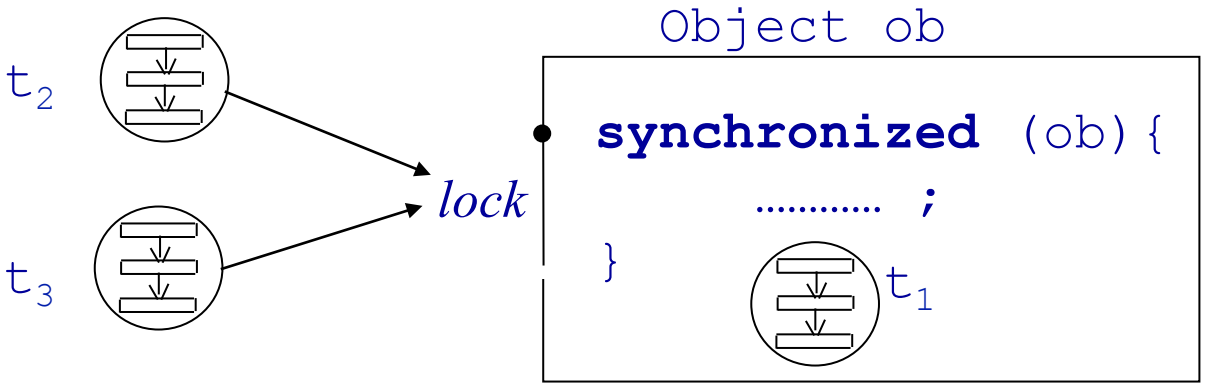
synchronized block

- 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 contatore {
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 5.0, 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:

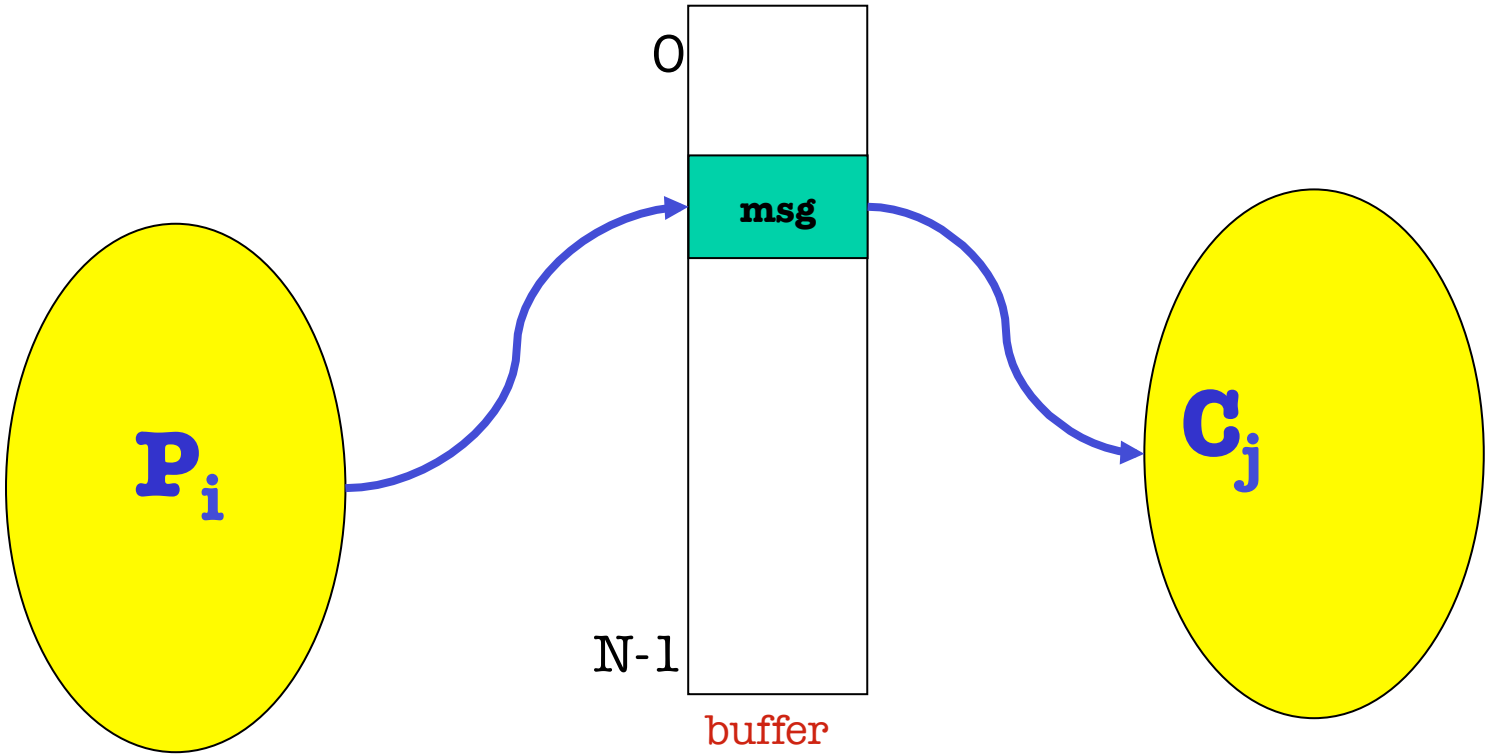
```
Inizializzazione ad un valore K dato:

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. (http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Semaphore.html)

Esempio: produttori e consumatori con buffer di capacità N



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

```
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();
    }
}
```

Esempio: la Catena di Montaggio

Un'azienda elettronica produce schede a microprocessore. La produzione di ogni scheda avviene in due fasi diverse:

1) **Assemblaggio**: in questa fase avviene l'assemblaggio automatico dei diversi componenti;
2) **Inscatolamento**: le schede assemblate vengono introdotte in scatole di capacita` N.

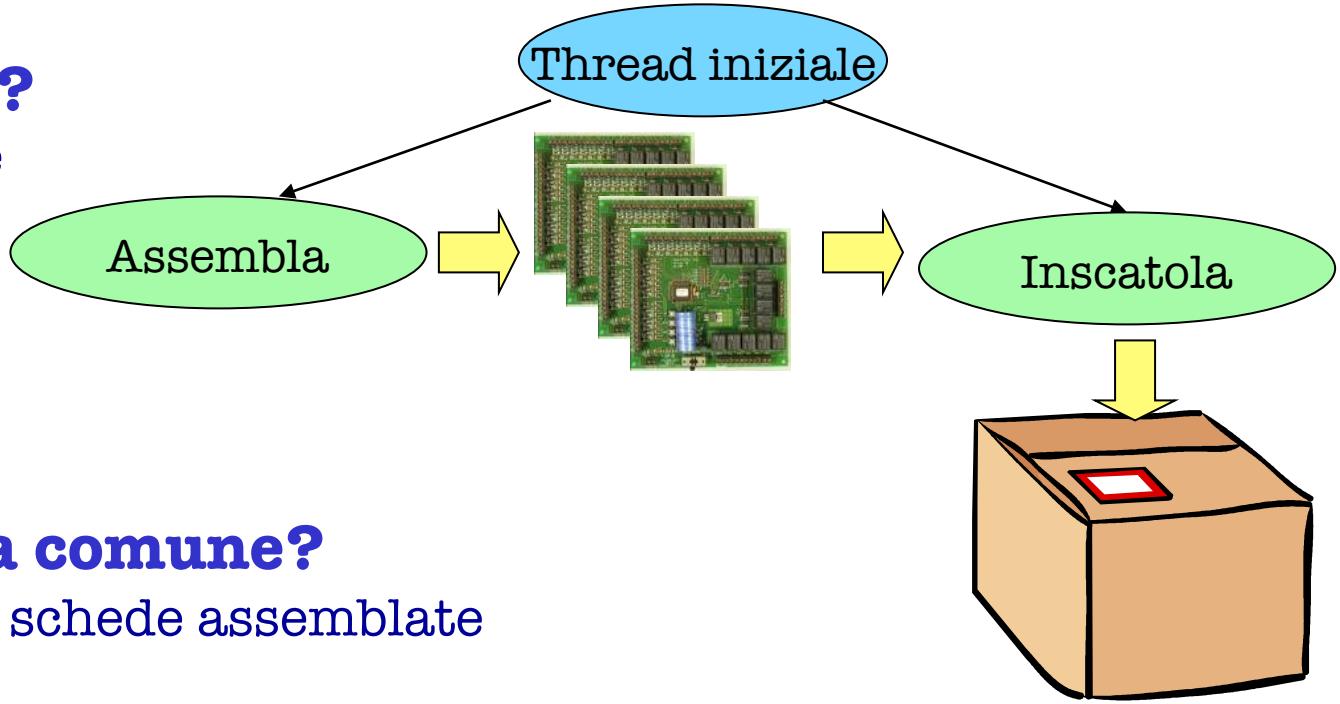
Si supponga di affidare **ognuna delle 2 fasi a un thread distinto** incaricato di controllare la macchina automatica dedicata alla realizzazione di quella particolare fase.
Utilizzando i **semafori**, si realizzi una politica di sincronizzazione che tenga conto dei seguenti vincoli:
- l'inscatolamento puo` essere attivato soltanto quando N nuovi prodotti sono stati assemblati.

Si supponga inoltre che il thread dedicato all'assemblaggio non possa effettuare una nuova fase di assemblaggio se vi sono ancora **MAX** schede da inscatolare (MAX > N).
La soluzione dovra` consentire un soddisfacente grado di concorrenza tra i threads.

Spunti e suggerimenti (1)

Quali thread?

- thread iniziale
- **Assembla**
- **Inscatola**



Quale risorsa comune?

- l'insieme delle schede assemblate

Il thread iniziale crea i 2 thread ***Assembla***, ***Inscatola***, ognuno con struttura ciclica:

- **Assembla**: al termine di un assemblaggio, soltanto dopo aver confezionato l'N-esimo prodotto, il thread Assembla attiva il thread *Inscatolamento*; puo` sospendersi se ci sono MAX schede ancora da inscatolare.
- **Inscatola**: una volta attivato provvede ad eseguire la fase di inscatolamento.

Spunti e suggerimenti (2)

Sincronizzazione:

- **mutua esclusione nell'accesso alle risorse condivise (assemblati):**
  - **definiamo un semaforo di mutua esclusione sM**
- **sospensione del processo che Assembla:**
  - ***definiamo un semaforo sA (v.i.=max)***
- **sospensione del processo che Inscatola:**
  - ***definiamo un semaforo sI (v.i.=0)***

```
public class threadA extends Thread{ // def. Assemblatore
   int i=0;
   int  da_produrre;
   risorsa r;

   public  threadA(risorsa R, int pezzi)
   {    this.r=R;
          this.da_produrre=pezzi;
   }

   public void run()
   {    try{
        System.out.print("\nThread ASSEMBLAGGIO: il mio ID
                                  è: "+getName()+"..\n");
        while (i<30)
                {       sleep(100);
                        r.nuovoA();
                        i++;
                        System.out.print("\n"+ getName() +":
                                nuovo assemblato...." +i+ "\n");
                }
        }catch(InterruptedException e){}

   }}
```

```
  public class threadS extends Thread{ //thread che inscatola

  int i, scatole=0;
  risorsa r;

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

  public void run()
  {   try{   System.out.print("\nThread INSCATOLAMENTO:
                     il mio ID è: "+getName()+"..\n");
             while (true)
             {       r.nuovaS();
                     sleep(100);/*durata inscatolamento...*/
                     scatole++;
                     System.out.print("\n"+getName()+"
                            inscatolamento "+scatole + ...\n");
             }
      }catch(InterruptedException e){}
  } //chiude run
}
```

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

public class risorsa {
      final int max=15;
      final int N = 4;  // capacità della scatola
      int pronti; // assemblati pronti (al massimo MAX)

      int i;
      Semaphore sA; // per la sospensione di TA; v.i max
      Semaphore sI; // v.i. 0 per la sospensione di TS
      Semaphore sM; // semaforo di mutua esclusione

  public  risorsa(int pronti)
  {   sA=new Semaphore(max);  //rappresenta lo spazio
                               //disponibile
      sI=new Semaphore (0);   // rappresenta il numero di
                              //scatole che possono essere
                              //confezionate
      sM=new Semaphore (1);   // mutua esclusione
      pronti =0;
  }
```

```
public void nuovoA()//deposito assemblato
   {   try{  sA.acquire();
             sM.acquire(); //inizio sez critica
             pronti=pronti+1;
             if (pronti%N==0)
                   sI.release();
             sM.release(); //fine sez critica
         }catch(InterruptedException e){}
   }

public void nuovaS() //preleva N assemblati
   {   try{  sI.acquire();
             sM.acquire();
             pronti=pronti-N;
             for(i=0; i<N; i++)
                   sA. release();
             sM.release();
         }catch(InterruptedException e){}
   }

}
```

```
public class supplychain{

 public static void main(String args[]) {

    risorsa R=new risorsa(0);
    threadA TA=new threadA(R, 1000);
    threadS TS=new threadS(R);

    TA.start();
    TS.start();

            }

}
```

Java Thread: Alcune considerazioni al contorno: i problemi di `stop()` e `suspend()`

`stop()`

- forza la ***terminazione*** di un thread
- tutte le ***risorse utilizzate vengono immediatamente liberate*** (lock compresi)

Se il ***thread interrotto*** stava compiendo un insieme di operazioni da eseguirsi in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente del sistema.***

Per questo motivo il metodo stop() è "deprecated".

I problemi di `stop()` e `suspend()`

`suspend()`

- ***blocca l'esecuzione di un thread***, in attesa di una successiva invocazione di **`resume()`**
- non libera le risorse impegnate dal thread (***non rilascia i lock***)

Se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva*** (ad esempio sospeso durante l'esecuzione di un metodo synchronized), tale ***risorsa rimane bloccata.***

Per questo motivo il metodo suspend() è "deprecated".

Altri metodi di interesse per Java thread

- **`sleep(long ms)`**
  - sospende thread per il # di ms specificato
- **`interrupt()`**
  - invia un evento che produce l'interruzione di un thread
- **`interrupted()/isInterrupted()`**
  - verificano se il thread corrente è stato interrotto
- **`join()`**
  - ***attende la terminazione del thread specificato***
- **`isAlive()`**
  - true se thread è stato avviato e non è ancora terminato
- **`yield()`**
  - ***costringe il thread a cedere il controllo della CPU***