Progettazione del Software
Thread: concetti di base

Giuseppe De Giacomo, Massimo Mecella
Dipartimento di Informatica e Sistemistica
SAPIENZA Università di Roma

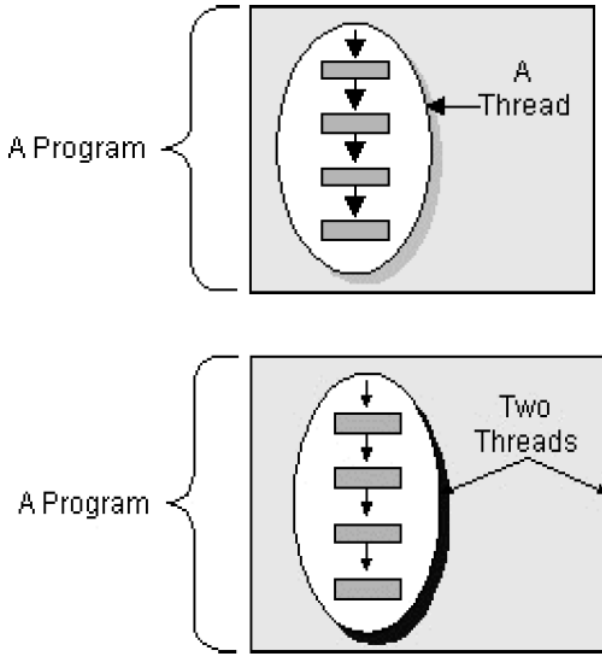
Slide adattate da Stefano Millozzi – Progetto di Reti 2008/09

Alcuni concetti

1. ***Monotasking:***
   - *elaborazione batch in* rigorosa sequenza
2. ***Multitasking:***
   - Elaborazione "contemporanea" di più compiti diversi (task)
   - Ogni task è tipicamente un processo separato
3. ***Multithreading:***
   - Elaborazione "contemporanea" di più istanze dello stesso task
   - Ogni task è un thread;
     - più thread fanno parte dello stesso processo

Thread

1. Un thread è un flusso di controllo sequenziale nell'ambito di un programma



Thread

1. Flusso di esecuzione indipendente nel contesto di un programma
   - Talvolta chiamati lightweight process o execution context
2. Differenze rispetto ai processi:
   - le uniche risorse specifiche del thread sono quelle necessarie a garantire un flusso di esecuzione indipendente (contesto, stack,…)
   - tutte le altre risorse, incluso lo spazio di indirizzamento,
   - sono condivise con gli altri thread

Thread

Flusso di esecuzione indipendente

- "illusione" che ogni flusso abbia un "processore" dedicato all'esecuzione del proprio codice
- Come questo venga effettivamente realizzato in elaboratori in cui il numero dei processori sia (molto) inferiore a quello dei thread dipende dai meccanismi propri del sistema operativo e della JVM. Esula dagli scopi del corso, cfr. i corsi di "Sistemi Operativi" e di "Progetto di Reti di Calcolatori e Sistemi Informatici"

Utilizzo dei Thread

1. Per migliorare l'interazione con l'utente
   - mantenendo rapide tutte le interazioni con l'utente
   - le operazioni lunghe vengono svolte da thread dedicati senza compromettere la reattività del thread che gestisce l'interazione con l'utente
2. Per simulare attività simultanee
3. Per sfruttare sistemi multi-processore
   - al fine di dedicare più processori all'esecuzione dello stesso programma

java.lang.Thread

1. I thread della JVM sono associati ad istanze della classe **`java.lang.Thread`**

2. Gli oggetti istanza di **`Thread`** svolgono la funzione di interfaccia verso la JVM che è *l'unica capace di creare effettivamente nuovi thread*

3. Attenzione a non confondere il concetto di thread con gli oggetti istanza della classe **`java.lang.Thread`**
   - tali oggetti sono solo lo strumento con il quale è possibile comunicare alla JVM
     - di creare nuovi thread
     - di interrompere dei thread esistenti
     - di attendere la fine di un thread (join)
     - …

Implementazione dei Thread in Java

1. Due modi per creare nuovi thread:
   - Instanziare classi che derivano da **`java.lang.Thread`**
     - più semplice
     - ma non è possibile derivare da altre classi

     Naive

   - Instanziare direttamente **`Thread`** passando al suo costruttore un oggetto di interfaccia **`Runnable`**

     OK!

     - bisogna implementare l'interfaccia **`Runnable`** e creare esplicitamente l'istanza di **`Thread`**
     - ma la classe rimane libera di derivare da una qualsiasi altra classe

2. Le istruzioni da eseguire devono essere inserite nel metodo **`run()`**, chiamato da automaticamente **`Thread`**
   - Metodo **`run()`**: ereditato da **`Thread`** o offerto da **`Runnable`**

Runnable

```
interface Runnable {
    public void run();
}
```

Quando un oggetto che implementa l'interfaccia **`Runnable`** (e che quindi offre il metodo **`run()`**) viene usato per creare un thread, il thread appena lanciato manda appunto in esecuzione il metodo **`run()`**

- *… quindi il metodo **`run()`** è il codice indipendente di cui il thread è il esecutore "virtuale" dedicato*

Java.lang.Thread

Constructor Summary
`Thread``()` Allocates a new `Thread` object.
`Thread``(Runnable target)` Allocates a new `Thread` object.

Method Summary	
`void`	**`run`**`()` If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.
`void`	**`start`**`()` Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

Java.lang.Runnable

Method Summary
void **run**()

Method Detail

run

`public void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also: `Thread.run()`

Esempio 1 (soluzione naive)

1. La classe e' un'istanza di Thread (run "overidden")
2. Possibile SOLO nel caso di classe "non figlia"

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

Nel cliente

```
new SimpleThread("Jamaica").start();
```

Esempio 2 (soluzione da preferire)

1. La classe e'inserita in un'istanza di Thread
2. Unica possibilità nel caso di classe "figlia" (vedi classi UML)

```
public class SimpleThreadRunnable extends Dummy implements Runnable {
…
  public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try {
                        sleep((long)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }
      System.out.println("DONE! " + getName());
}
```

Nel cliente

```
SimpleThreadRunnable Ja = new SimpleThreadRunnable("Jamaica");
…
new Thread(Ja).start();
```


Esempio

```
public class MioSempliceThread extends C
                          implements Runnable {
    …
    public void run() {
          //faccio qualcosa
          …
    }
}
```

C'è un solo flusso di esecuzione

```
MioSempliceThread mst = new MioSempliceThread(…);
…
new Thread(mst).start();
…
```

Da qui in poi ci sono due flussi

Thread

```
public class Java.lang.Thread {
        public Thread Thread(){…}
        public Thread Thread(Runnable target){…}
        public void run(){…}
        public void start(){…}
        public static void sleep(long millisec) {…}
        …
   }
```

Istanzia un oggetto `Thread`

Istanzia un oggetto `Thread`, passandogli un `target` che offre il metodo `run()`

Avvia l'esecuzione del thread; la JVM invoca il metodo `run()`

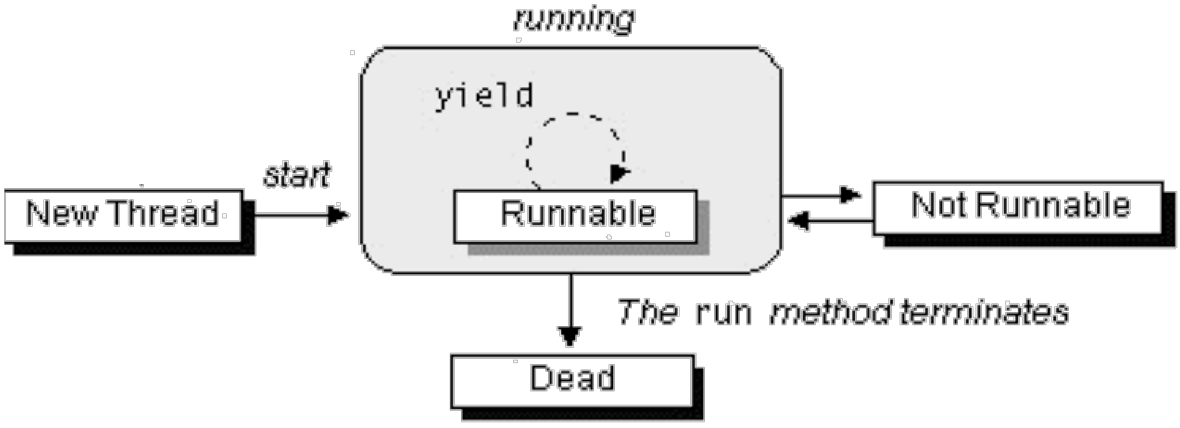
Mette il thread in pausa per il tempo specificato

IF il `Thread` è stato creato passandogli un `target`
THEN chiama il suo metodo `run()`, di fatto mandando in esecuzione nel nuovo thread appena creato il metodo offerto da `target`
ELSE non fa nulla e ritorna immediatamente

Ciclo di vita di un thread

Stati in cui si può trovare un thread

1. New Thread
2. Invocazione del metodo start()
3. Attivo
   - Running
   - Not Running
4. Dead

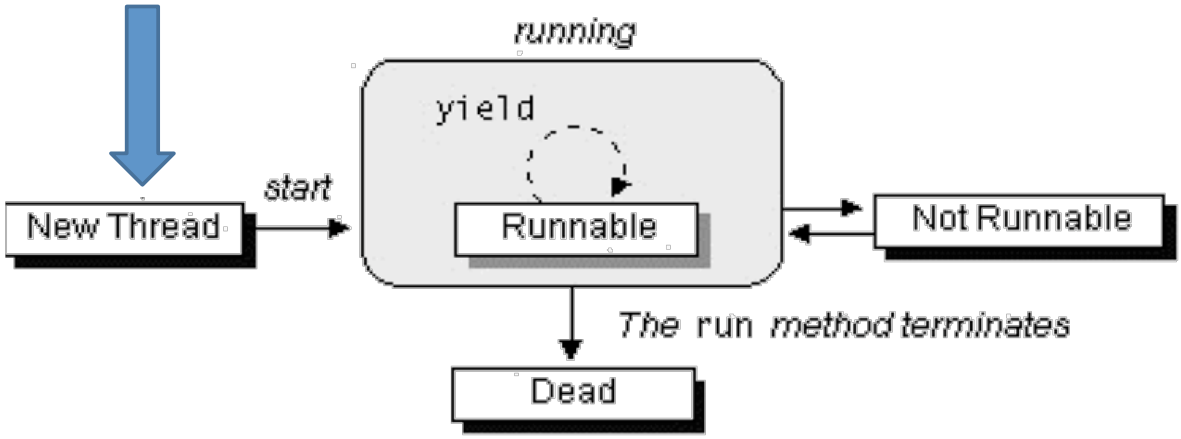


Ciclo di vita di un thread

Creazione di un nuovo oggetto Thread

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

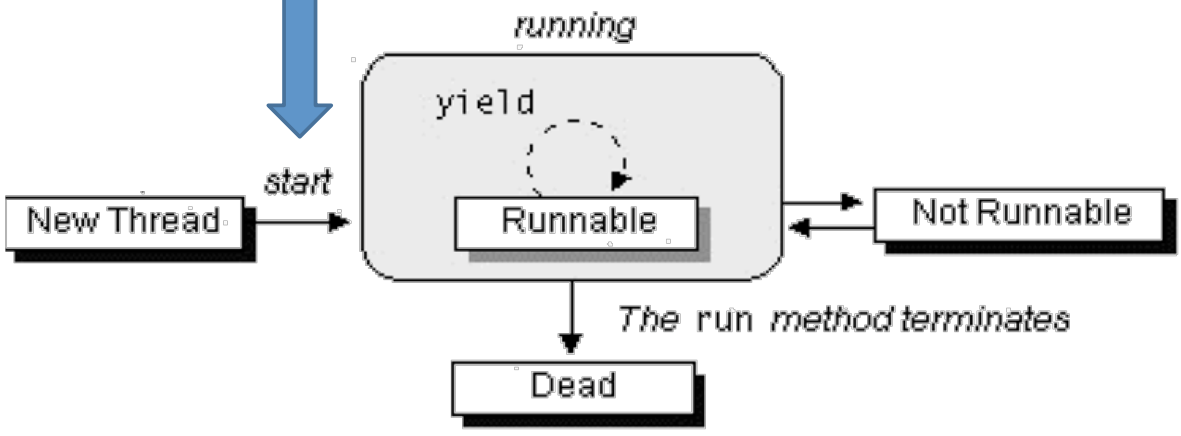
```
SimpleThreadRunnable Ja = new SimpleThreadRunnable("Jamaica");
Thread aux = new Thread(Ja);
```



Ciclo di vita di un thread

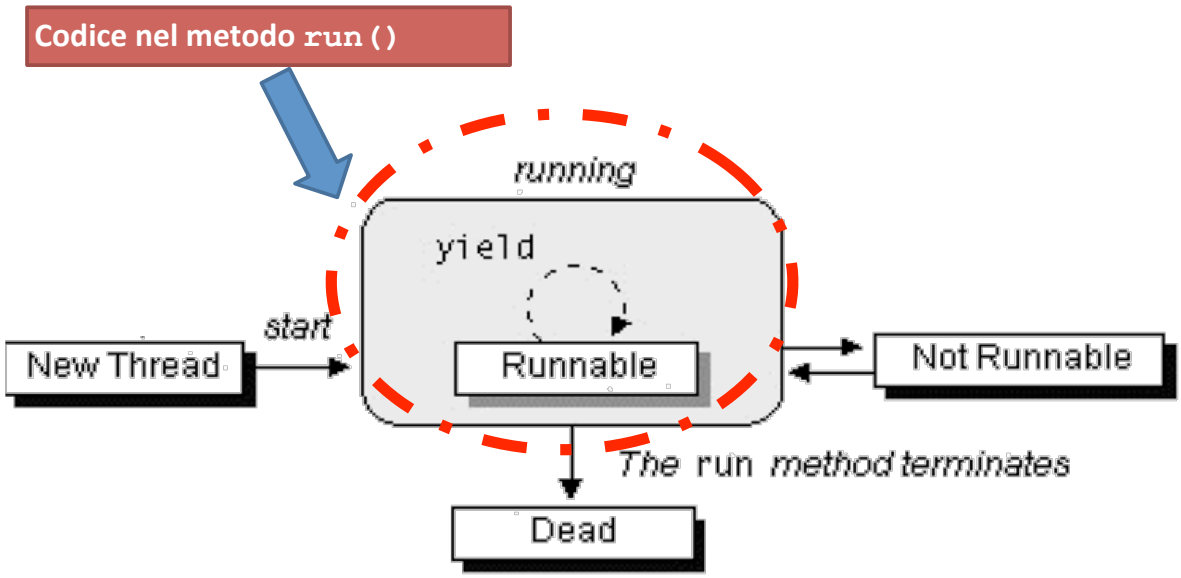
Il metodo **`start()`** crea le risorse di sistema necessarie ad eseguire il thread, schedula il thread per l'esecuzione, ed invoca il metodo **`run()`**

```
aux.start();
```



Ciclo di vita di un thread

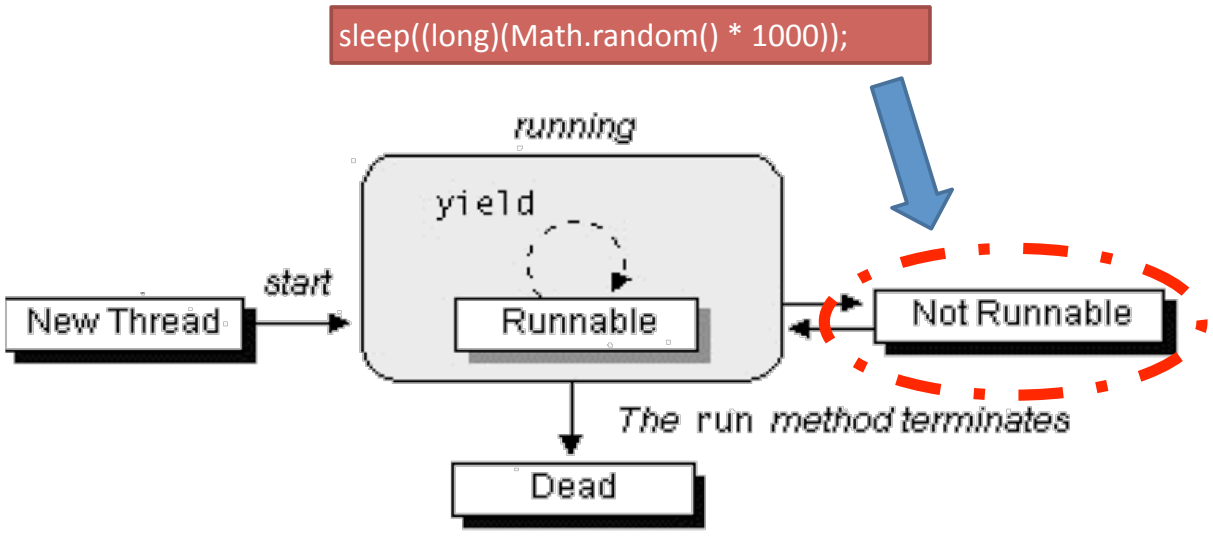
Un thread appena avviato è nello stato ***Runnable***



Ciclo di vita di un thread

Un thread diventa ***Not Runnable*** quando si verifica uno di questi eventi:

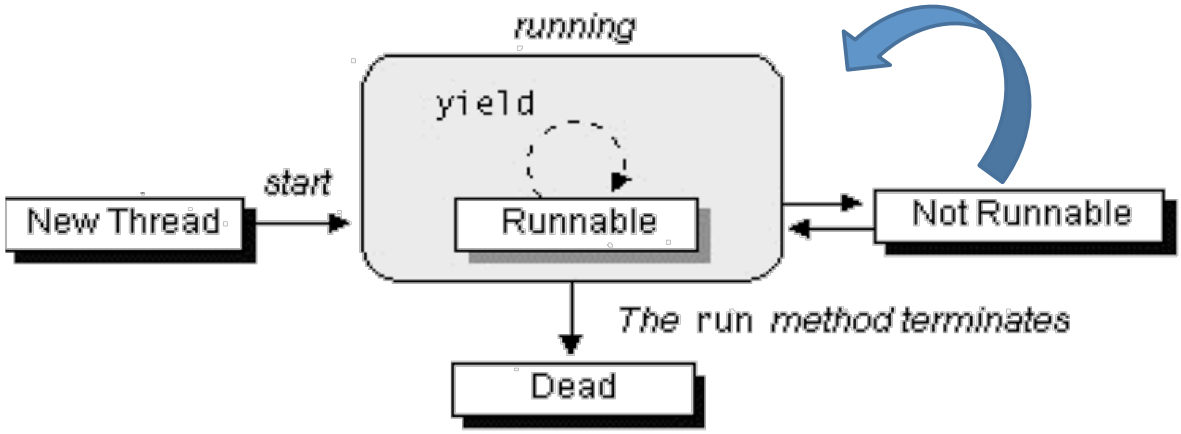
- È invocato il suo metodo **`sleep()`**
- Il thread chiama il metodo **`wait()`** per attendere che sia soddisfatta una specifica condizione -- vedere dopo
- il thread è in attesa di **I/O**



Ciclo di vita di un thread

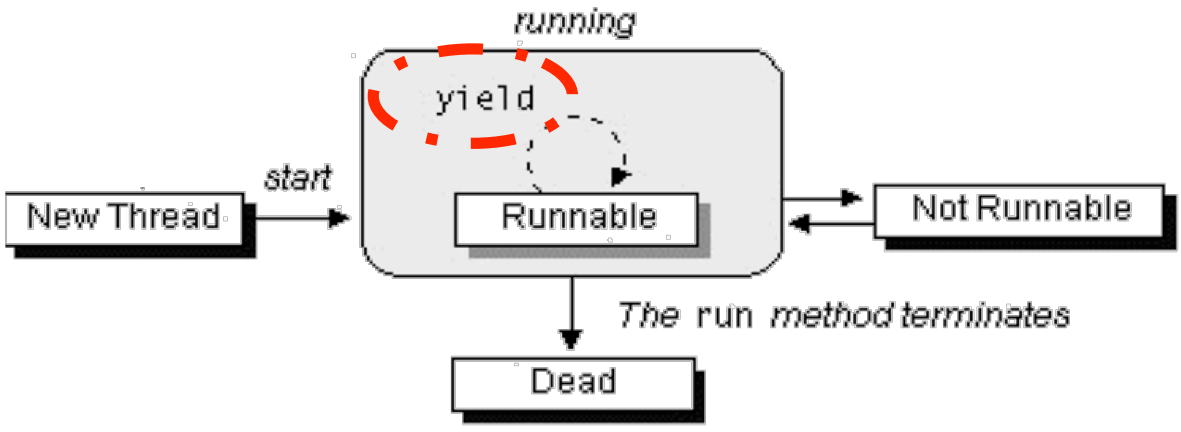
Per ogni ingresso nello stato ***Not Runnable***, c'è una precisa e definita situazione che lo riporta in ***Runnable***

- Se il thread è stato messo in sleep, allora il numero specificato di millisecondi deve passare
- Se il thread è in waiting su una condizione, allora un altro oggetto deve notificare ad esso il cambiamento di condizione, chiamando **`notify()`** o **`notifyAll()`**
- Se il thread è in attesa di **I/O**, allora l'I/O deve completare



Ciclo di vita di un thread

L'ulteriore metodo **`yield()`** mette in pausa temporaneamente il thread, permettendo ad altri di passare all'esecuzione

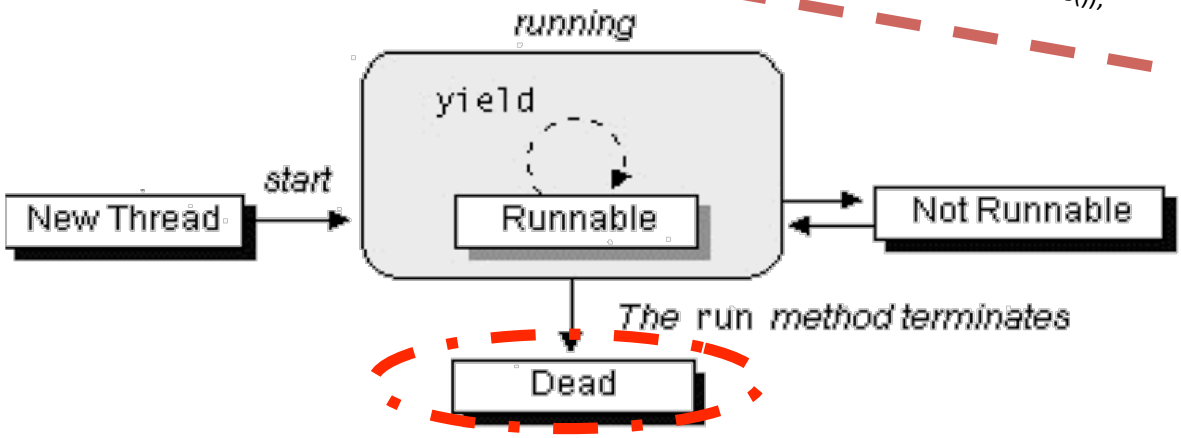


Ciclo di vita di un thread

Il thread muore quando il metodo **`run()`** termina la sua esecuzione

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

Il metodo run() termina

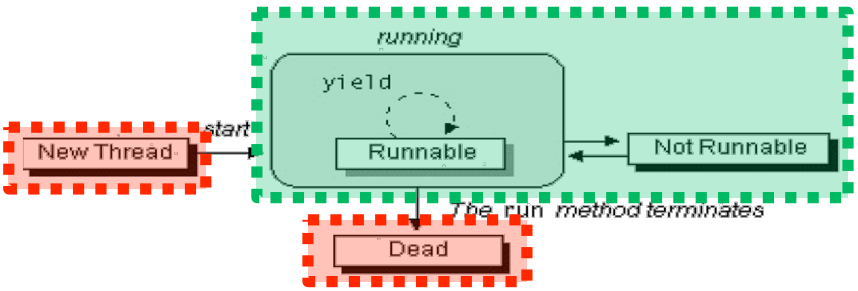


Ciclo di vita di un thread

`public boolean isAlive()` in Thread

- Ritorna **`false`** se il thread è nuovo (appena creato) o è DEAD
- Ritorna **`true`** se il thread è stato avviato e non fermato (quindi è RUNNABLE o NOT RUNNABLE)

Quindi NON si può distinguere precisamente tra i vari stati



Alcuni metodi della classe Thread

1. **`public void start()`**
   - Avvia il thread: attenzione invocando il metodo **`run()`** non viene avviato un nuovo thread ma semplicemente il metodo **`run()`** viene eseguito nel thread corrente

2. **`public static void sleep(long millisec)`**
   - Il thread passa nello stato NOT RUNNABLE per il tempo specificato, dopo di che torna nello stato RUNNABLE

3. **`public void yield()`**
   - Il thread cessa di essere RUNNING. Se vi sono molti thread RUNNABLE in attesa, il metodo garantisce il passaggio ad un thread RUNNABLE diverso solo se questo ha priorità maggiore o uguale
     - Lo vedremo in dettaglio di seguito

Esercizio

1. Realizzare una classe java Runnable che rappresenta un thread per stampare (senza andare a capo) su output 100 volte un carattere passato come parametro.
   - Scrivere un main in cui vengono creati avviati due thread, la prima che stampa il carattere '0' e la seconda '1'
   - Eseguire più volte il programma e verificare se le sequenze di 0 e 1 generate sono uguali
2. Soluzione:
   - esercizioCaratteri01

Gestione dei Thread

1. **Non preemptive**
   - Il gestore non interrompe mai un thread in esecuzione
   - E' ciascun thread che cede il controllo della CPU in maniera esplicita (`I/O`, `sleep()`, `yield()`, coordinamento tra thread)

2. **Preemptive**
   - Il gestore pone in pausa un thread:
     - A suddivisione di tempo (periodo di tempo di CPU allocato ad ogni thread)
     - Non a suddivisione di tempo (priorità dei thread)

Gestione dei Thread in JAVA

(Idealmente secondo specifiche)

1. **Preemptive**
   - A selezione di thread RUNNABLE con priorità più alta
   - SOLO UN THREAD eseguito tra thread di pari priorità (scelta round-robin)

2. Non c'è, in generale, suddivisione di tempo nei thread implementati nella Java VM (classe Thread)

Nota sulla suddivisione di tempo

1. Specifiche della Java Virtual Machine non indicano come i thread Java debbano essere mappati su thread nativi del s.o.

3. Può esistere l'implementazione di thread nativi di sistema a suddivisione di tempo
   - In generale dipendenza dalla piattaforma
   - Es. in WindowsNT a ciascun thread un processo di sistema gestito a suddivisione di tempo

4. Non confidare nella suddivisione di tempo!
   - Adottare tecniche "sagge" (vedi dopo!)

Priorità dei Thread

1. `public final static int MAX_PRIORITY = 10`
2. `public final static int NORM_PRIORITY = 5`
3. `public final static int MIN_PRIORITY = 1`

5. `public final int getPriority()`
6. `public final void setPriority(int newPriority)`
   - Valori ammessi tra 1 e 10

Nota sulla priorità

1. Mapping priorità thread Java su priorità thread nativi del s.o. dipendente da piattaforma
2. Possono esistere implementazioni di thread nativi che non tengono conto delle priorità dei thread Java (es. Linux in alcune modalità)
3. Adottare tecniche "sagge" per garanzia di portabilità (vedi dopo!)

Thread: alcune tecniche

1. Non usare cicli infiniti o prevedere nel ciclo:
   - operazioni di *I/O*
   - fasi di *sleep*
   - operazioni di *coordinamento tra thread*
2. Richiamare il metodo *yield nel caso di* thread con operazioni CPU-bound
   - rilascio volontario di CPU
3. Mai fare affidamento completo sul timeslicing nè solo sulla priorità per conseguire portabilità

Scope delle variabili nei thread

1. Variabili di metodo (inclusi parametri) sono locali ad un thread
   - Allocate nello stack locale di ciascun thread
   - Modifiche fatte a variabili locali di un thread non sono fatte a variabili locali di un altro thread
2. Variabili di istanza sono comuni a tutti i thread che accedono a quell'istanza
3. Variabili di classe sono comuni a tutti i thread nel runtime

5. Attenzione ad accedere in maniera concorrente ad una stessa variabile da parte di più thread
   - Occorrono regole di sincronizzazione!
   - Lo vedremo in seguito

Esempio

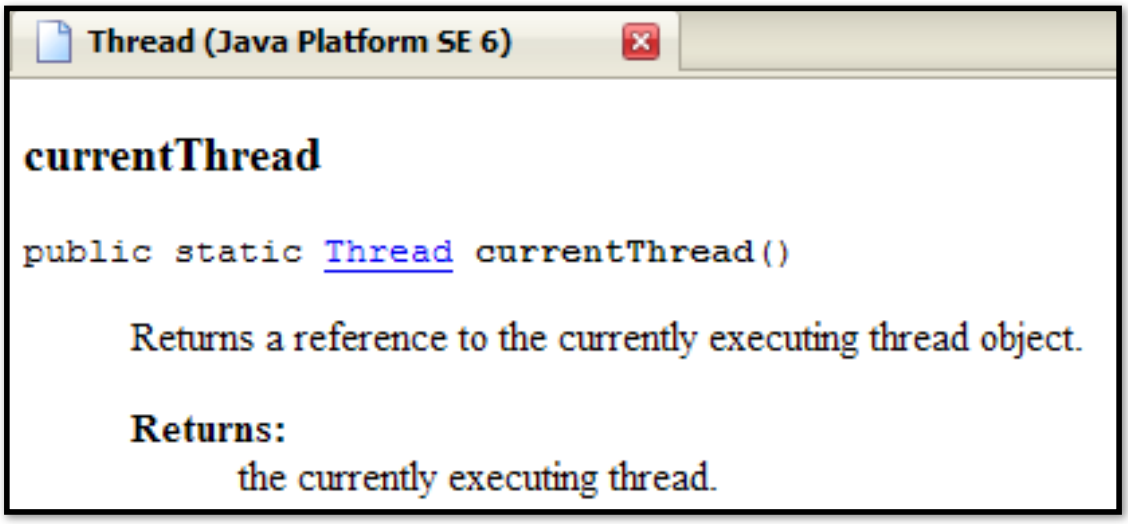
```
public class Test {

      static int staticVAR;        // tutti i thread nel runtime
      Object instanceVAR;          // tutti i thread che accedono
alla
                                   // stessa istanza di Test


      void methodFOO(int paramP) {
           int localVAR = paramP;  // ogni thread ha il suo
          …
      }
}
```

Esempio thread con var locali

1. Esempio esempioSemplice
   - Due thread corrono concorrentemente

Metodo currentThread

1. La classe **`Thread`** contiene il metodo static **`currentThread()`** che ritorna il **`Thread`** corrente

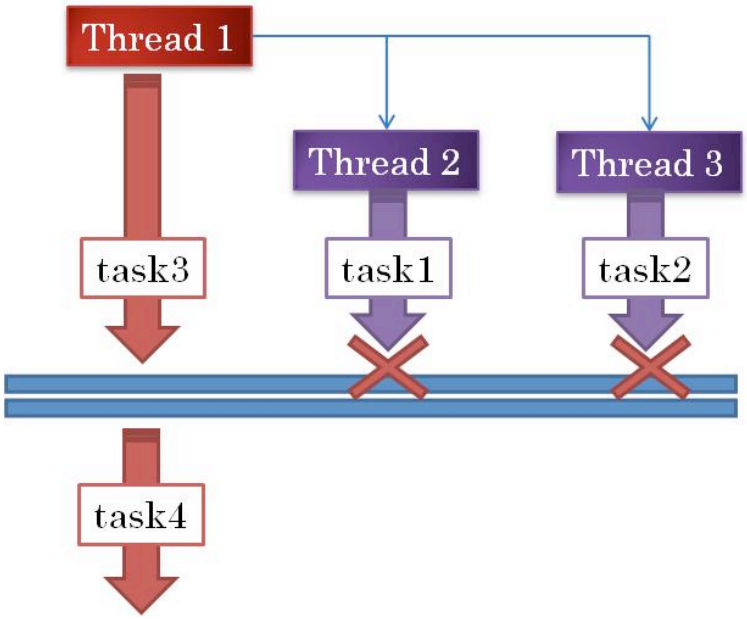


SINCRONIZZAZONE

Sincronizzazione e cooperazione tra thread

1. Sincronizzazione
   - Un thread può interagire con altri thread sincronizzando le proprie azioni in funzioni degli altri thread, per il raggiungimento di un fine comune
     - Ad esempio un thread potrebbe mettersi in attesa che un altro thread termini per poter proseguire l'elaborazione
     - Viene utilizzata per controllare che il flusso delle elaborazioni parallele avvenga senza problemi
2. Cooperazione
   - Si tratta di un passo ulteriore in cui i thread si sincronizzano per potersi scambiare dati
     - Un tipico esempio è il paradigma Produttore/Consumatore in cui l'output di un thread diventa input per un altro

Sincronizzazione

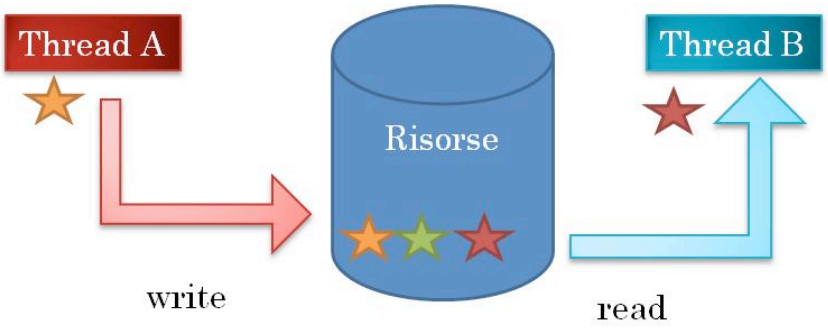


Il thread1 avvia altri task paralleli per poi mettersi in attesa del loro completamento per continuare il proprio task

Cooperazione

I thread condividono risorse comuni su cui lavorare

- I thread oltre a sincronizzarsi si scambiano messaggi
- L'output di un thread può essere l'input per un altro e occorrono strutture condivise che permettano la comunicazione tra thread



Thread in Java: Creazione e Terminazione

1. Creazione di thread
   - si creano oggetti istanze della classe **`java.lang.Thread`**
     - Il thread viene effettivamente creato dalla JVM non appena si richiama il metodo **`start()`**
     - Il nuovo thread esegue il metodo **`run()`**

2. Terminazione di thread
   - i thread terminano quando
     - finisce l'esecuzione del metodo **`run()`**
     - sono stati interrotti con il metodo **`interrupt()`**
     - eccezioni ed errori

3. Join con thread
   - richiamando il metodo **`join()`** su un oggetto **`Thread`** si blocca il thread corrente sino alla terminazione del thread associato a tale oggetto

Interruzione di Thread

1. Un thread può interrompere un altro thread invocando il metodo **`interrupt()`** del corrispondente oggetto **`Thread`**

2. In realtà il thread interrompente segnala la richiesta di interruzione settando un flag in tale oggetto

3. In generale il thread interrotto non serve la richiesta di interruzione immediatamente

4. Invece solleva una eccezione in alcuni momenti della sua esecuzione

java.lang.InterruptedException

1. Il thread interrotto servirà la richiesta di interruzione in opportuni momenti; tra i quali
   - durante l'esecuzione di una **`sleep()`**
   - al momento dell'invocazione di una **`sleep()`**
   - durante l'esecuzione di una **`wait()`**

2. Il thread interrotto solleva una **`java.lang.InterruptedException`**
   - in seguito l'interruzione si considera servita ed il flag di interruzione viene resettato
3. In questo modo si possono programmare thread che siano interrotti solo nei punti voluti, senza "traumi"

4. *N.B. le vecchie implementazioni della JVM hanno mostrato tutti i problemi delle soluzioni precedenti*

SleepInterrupt.java (1/2)

```
public class SleepInterrupt implements Runnable {
    public void run() {
        try {
            System.out.println("in run()");
            Thread.sleep(20000);
            System.out.println("in run() - woke up");
        } catch ( InterruptedException x ) {
            System.out.println("in run() - interrupted while sleeping");
            return;
        }
        System.out.println("fine");
    }
}
```

SleepInterrupt.java (2)

```
public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();

        // Be sure that the new thread gets a chance to run for a while
        try { Thread.sleep(2000); }
        catch ( InterruptedException x ) { }

        System.out.println("in main() - interrupting other thread");
        t.interrupt();
        System.out.println("in main() - leaving");
}
```


Join

`void join()`

- Attende che il thread muoia

`void join(long mills)`

- Attende al più **`mills`** millisecondi la morte del thread

join() di un Thread

java.lang
Class Thread
java.lang.Object
└**java.lang.Thread**
All Implemented Interfaces: Runnable

Method Summary

void	**join**() Waits for this thread to die.
void	**join**(long millis) Waits at most `millis` milliseconds for this thread to die.
void	**join**(long millis, int nanos) Waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.

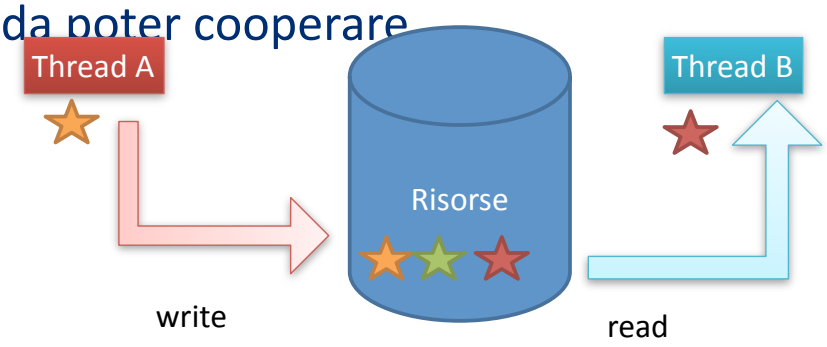
Esempio Join

1. Esempio esempioJoin
   - Illustra uso Join()

Cooperazione tra thread

1. Ci sono molte situazioni in cui diversi thread concorrenti in esecuzione condividono le medesime risorse. Ogni thread deve tener conto dello stato e delle attività degli altri.

2. Visto che i thread condividono una risorsa comune, devono essere sincronizzati in qualche modo in modo da poter cooperare

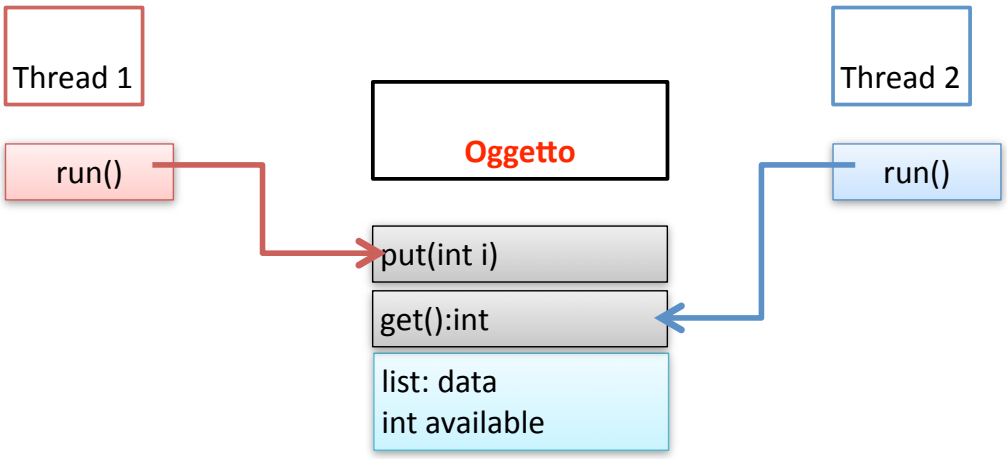


Esempio accesso a dati condivisi

1. Esempio Contatore
   - non synchronized
   - synchronized

Accesso concorrente ad un oggetto

1. Due thread che utilizzano un oggetto intermedio per scambiarsi dati, usando i metodi get e put per leggere/scrivere dati



Lock di oggetti (synchronized)

1. Un oggetto condiviso tra un thread Produttore (thread 1) ed uno Consumatore (thread 2)
   - Sincronizzazione basata su acquisizione e rilascio di un lock su un oggetto
   - Solo il thread in possesso del lock può usare l'oggetto intermedio
2. Risoluzione delle "race conditions" tra thread (-> data integrity)

4. Acquisizione/rilascio lock di un oggetto effettuata automaticamente ed atomicamente dal Java Runtime Environment (JRE) (-> Oggetti dotati di monitor)
   - Lock basato sulla keywork synchronized

5. **Vediamo in dettaglio → Monitor**

Monitor/1

1. "Custode" di un oggetto che determina l'accesso a suoi comportamenti (metodi e sezioni di codice)
2. Mutua esclusione su gruppi di procedure
   - Un thread alla volta in esecuzione (altri thread sospesi)
3. In Java la parola chiave è ***synchronized***
   - Nella signature del metodo di un oggetto (non solo...)
   - Su esecuzione di "synchronized", verifica da parte del monitor sull'uso dell'oggetto ed accesso garantito o bloccato al thread
   - Coda di thread per l'accesso al metodo *synchronized di* un oggetto
4. Monitor oggetto rilasciato dal thread a fine esecuzione di metodo (o sezione) synchronized

Monitor/2

Cosa sincronizziamo

1. Metodi di istanza (**public synchronized…**)
   - I metodo di istanza synchronized può essere attivo in un determinato momento
2. Metodi statici di classi (**public static synchronized…**)
   - monitor per classe che regola l'accesso a tutti i metodi static di quella classe
   - Solo un metodo static synchronized può essere attivo in un determinato momento
3. Singoli blocchi di istruzione, controllati da una variabile oggetto

   synchronized (oggetto) {Istruzione}
   - Espressione restituisce un oggetto, NON un tipo semplice
   - Usata per sincronizzare metodi non sincronizzabili (ad es. controllo accesso agli oggetti array)
   - La sincronizzazione viene fatta in base all'oggetto e un solo blocco synchronized su un particolare oggetto può essere attivo in un determinato momento
     - NB: se la sincronizzazione viene fatta su istanze differenti, i due blocchi non saranno mutuamente esclusivi!

Schemi di utilizzo

```
synchronized void metodo1() {…}
synchronized void metodo2() {…}
```

```
static synchronized void metodo1() {…}
static synchronized void metodo2() {…}
```

```
void metodo1(String param) {
        synchronized(param){
            …
        }
}
```


Metodi non sincronizzati

1. Non interpellano il monitor di un oggetto
2. Ignorano il monitor e vengono eseguiti regolarmente da qualsiasi thread ne faccia richiesta
3. Da ricordare:
   - un solo thread alla volta può eseguire metodi *synchronized*
   - un qualsiasi numero di thread può eseguire metodi non *synchronized*

Esempio coda limitata condivisa

1. Esempio coda limitata condivisa con errori
2. Esempio con aggiunta di parti synchronized

COORDINAMENTO

Limiti della sola sincronizzazione

1. L'uso della sincronizzazione permette di gestire l'accesso multiplo ad ai metodi di un oggetto in maniera safe
   - Come gestire invece l'interazione di tipo Produttore/ Consumatore?
     - In particolare quali meccanismi posso usare per bloccare ad esempio il Consumatore quando non ci sono dati disponibili?
     - Analogamente come blocco il produttore se non sono disponibili aree in cui salvare i dati?

2. Ho bisogno di primitive aggiuntive:
   - **`notify()`**, **`notifyAll()`**, **`wait()`**

4. Si tratta di primitive che ricordano *wait* e *signal* dei *semafori* (vedi corso Sistemi Operativi) e che permettono in java di rilasciare il lock su una risorsa sincronizzata e di

Coordinamento

1. Condizioni all'interno di un metodo sincronizzato

```
while (!la condizione che desidero) {
    wait();
}
```

2. Attesa
   - Thread posto in pausa nella coda di attesa del monitor dell'oggetto (stato NOT RUNNABLE)
3. Notifica
   - Thread rimosso dalla coda di attesa del monitor dell'oggetto e posto in esecuzione con la segnalazione che la condizione desiderata è vera

wait, notify, notifyAll della classe Object

1. `public final void wait()`
2. `public final void wait(long millisec)`
3. `public final void wait(long millisec, int nanosec)`
   - Determinano l'attesa di un thread fino all'invocazione di **`notify()`**, **`notifyAll()`**, alla scadenza del timeout o alla sua interruzione attraverso l'invocazione di interrupt()
4. `public final void notify()`
   - Risveglia un thread specifico in attesa sulla coda del monitor dell'oggetto
5. `public final void notifyAll()`
   - Risveglia tutti i thread in attesa sulla coda del monitor dell'oggetto

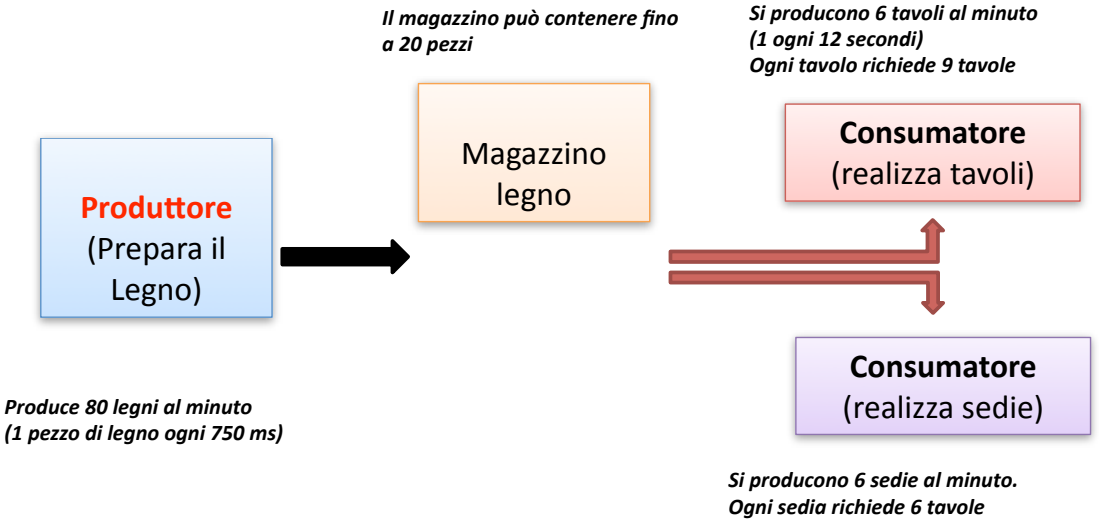
6. I metodi **`wait`** sono invocabili sono all'interno di un blocco sincronizzato

Method Summary

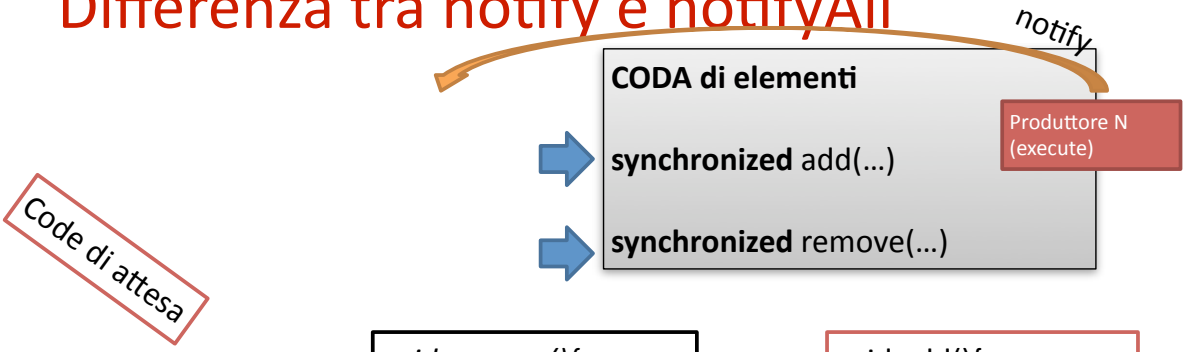
1. void **notify**()
   Wakes up a single thread that is waiting on this object's monitor.
2. void **notifyAll**()
   Wakes up all threads that are waiting on this object's monitor.
3. void **wait**()
   Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
4. void **wait**(long timeout)
   Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
5. void **wait**(long timeout, int nanos)

Esempio

1. Simulare l'andamento di una catena di montaggio per la lavorazione del legno e la conseguente produzione di tavoli e sedie
   - Disponibilità di un magazzino a capacità limitata
     - Cambiare le velocità di lavorazione per vedere le produzioni come cambiano



Differenza tra notify e notifyAll




```
void remove(){
  while(!vuota){wait();}
  rimuovi alla lista
  notify()
}
```

```
void add(){
  while(piena){wait();}
  aggiungi alla lista
  notify()
}
```

Supponiamo che dopo l'esecuzione del metodo **add** da parte del "***produttore N***" la coda si riempia, viene eseguito **notify** e risvegliato un solo thread (supponiamo Produttore 3).
Il metodo **add** viene eseguito ma la coda è piena e quindi il thread torna in **wait**.
→Il problema è che non viene avviato nessun altro thread e la situazione si blocca
→Se si utilizzasse **notifyAll** verrebbero risvegliati tutti i thread, uno dopo l'altro, e quindi prima o poi toccherebbe anche ad un thread consumatore

Esempio coda limitata condivisa

1. Esempio coda limitata condivisa con uso di **`wait()`** e **`notifyAll()`**

Deadlock

```
public class Deadlock implements Runnable {
      Deadlock grabIt;
      public synchronized void run() {
            try {Thread.sleep(2000);} catch (InterruptedException e) {}
            grabIt.syncMethod();
      }
      public synchronized void syncMethod() {
            try {Thread.sleep(2000);} catch (InterruptedException e) {}
            System.out.println("Sono in syncMethod");
      }
}

   public static void main(String[] args) {
      Deadlock d1 = new Deadlock();
      Deadlock d2 = new Deadlock();
      Thread t1 = new Thread(d1);
      Thread t2 = new Thread(d2);
      d1.grabIt = d2;
      d2.grabIt = d1;
      t1.start();
```

t1 chiama il metodo synchronized run su d1 che, dopo lo sleep, chiama syncMethod su d2; ma il monitor di d2 è in possesso di t2 che lo ha avviato invocando il metdodo synchronized run su d2 che, dopo lo sleep, chiama syncMethod su d1 il cui monitor è in possesso di t1 che lo ha avviato. t1 aspetta t2 che aspetta t1 !!!

I principali metodi dei Thread

static Thread **currentThread**()
Returns a reference to the currently executing thread object.

void **destroy**()**Deprecated**.

long **getId**()Returns the identifier of this Thread.

String **getName**() Returns this thread's name.

int **getPriority**() Returns this thread's priority.

ThreadGroup **getThreadGroup**() Returns the thread group to which this thread belongs.

void **interrupt**() Interrupts this thread.

boolean **isAlive**() Tests if this thread is alive.

I principali metodi dei Thread/2

void **join**() Waits for this thread to die.

void **join**(long millis) Waits at most millis milliseconds for this thread to die.

void **resume**()**Deprecated**.

void **run**()

static void **sleep**(long millis)

void **start**()

void **stop**() **Deprecated**.

void **stop**(Throwable obj) **Deprecated**.

void **suspend**() **Deprecated**.

static void **yield**() Causes the currently executing thread object to temporarily pause and allow other threads to execute.