Progettazione del Software
Thread: concetti di base

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

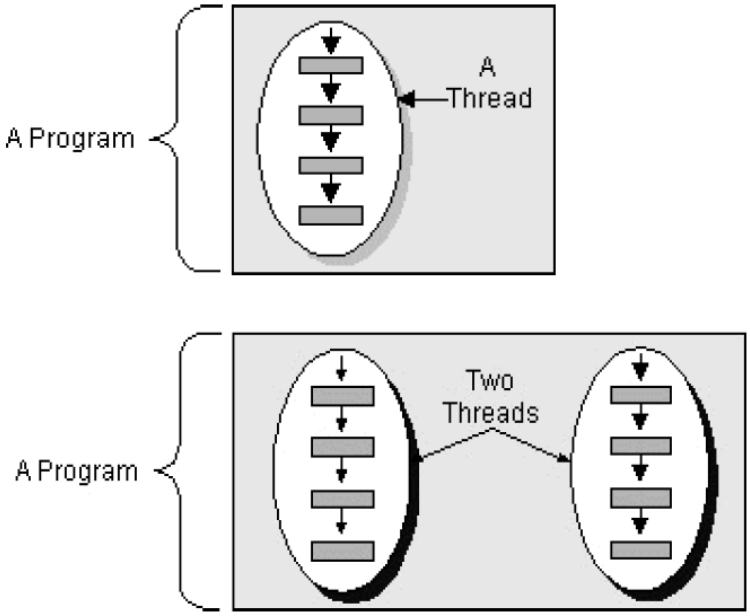
Queste slide fanno uso del materiale di Stefano Millozzi – Progetto di Reti

Alcuni concetti

- ***Monotasking:***
  - *elaborazione batch in* rigorosa sequenza
- ***Multitasking:***
  - Elaborazione "contemporanea" di più compiti diversi (task)
  - Ogni task è tipicamente un processo separato
- ***Multithreading:***
  - Elaborazione "contemporanea" di più istanze dello stesso task
  - Ogni task è un thread;
    - più thread fanno parte dello stesso processo – es: la Java Virtual Machine (JVM)

I thread

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



I thread

- Flusso di esecuzione indipendente nel contesto di un programma
  - Talvolta chiamati lightweight process o execution context
- Differenze rispetto ai processi:
  - le uniche risorse specifiche del thread sono quelle necessarie a garantire un flusso di esecuzione **indipendente** (contesto, **stack**,…)
  - lo spazio di indirizzamento, inclusa la **memoria dinamica** (heap), e più in generale tutte le altre risorse sono **condivise** con gli altri thread
- NB:
  - Creazione del thread più semplice del processo (più rapida)
  - Memoria condivisa tra tutti i thread

I 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

- Per **simulare attività simultanee**

- Per **sfruttare sistemi multi-processore**
  - al fine di dedicare più processori all’esecuzione dello stesso programma

- Per gestire in maniera efficiente le``**risorse lente**’’
  - Accesso al disco
  - Interazione con l’utente

- 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

java.lang.Thread

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

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

- 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 si comunicare alla JVM
    - di creare nuovi thread
    - di interrompere dei thread esistenti
    - di attendere la fine di un thread (join)
    - …
- NB: Creando un oggetto Thread non si crea un thread del SO

Implementazione dei thread in Java

- L'uso dei thread in Java è basato su
  - una classe predefinita **`java.lang.Thread`**
  - una interfaccia predefinita **`java.lang.Runnable`**

- Due modi per creare un nuovo thread:

  - Instanziare un oggetto **`Thread`** passando al suo costruttore un oggetto con interfaccia **`Runnable`**
    - bisogna implementare l'interfaccia **`Runnable`** e creare esplicitamente l'istanza di **`Thread`**
    - ma la classe rimane libera di derivare da una qualsiasi altra classe

    OK!

  - Instanziare classi derivate da **`Thread`** (che implementa lei stessa l'interfaccia **`Runnable`**)
    - più semplice,
    - ma non è possibile derivare da altre classi

    Naive

- *NB: entrambi i modi realizzano esattamente lo stesso funzionamento, descritto nel seguito: il primo esplicitamente, il secondo implicitamente,* ***noi useremo sempre il primo modo.***

Implementazione dei thread in Java

- L'oggetto che implementa l'interfaccia **`Runnable`** funge da "**eseguibile**":
  - L'operazione da eseguire viene inserita nel metodo **`run()`** richiesto da **`Runnable`**

- L'oggetto **`Thread`** funge da "**esecutore**" il cui compito è eseguire su un thread separato un eseguibile:
  - All'oggetto **`Thread`** è passato un **`Runnable`** (eseguibile) nel costruttore;
  - L'oggetto **`Thread`** usa il suo metodo **`start()`** per chiamare ed eseguire (in un nuovo thread) il metodo **`run()`** del **`Runnable`**.

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 da preferire)

- La classe **`MiaClasseRunnable`** implementa **`Runnable`**.
- Uno oggetto di questa (l'eseguibile) viene passato al costruttore di una istanza di **`Thread`** (l'esecutore), poi viene dato **`start()`**, mandando l'eseguibile in esecuzione sull'esecutore.

```
public class MiaClasseRunnable implements Runnable {
        private String nome;

        MiaClasseRunnable(String n) {
                nome = n;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println(nome + ": " + i);
                }
                System.out.println(nome + ": DONE! ");
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                MiaClasseRunnable ja = new MiaClasseRunnable("Jamaica");
                Thread mioThread = new Thread(ja);
                mioThread.start();
        }
}
```

Esempio 2 (soluzione naive)

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

```
public class MiaClasseThread extends Thread {
        private String nome;

        MiaClasseThread(String n) {
//              super();
                nome = n;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println(nome + ": " + i);
                }
                System.out.println(nome + ": DONE! ");
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                MiaClasseThread jat = new MiaClasseThread("Jamaica");
                jat.start();
        }
}
```

Esempio

```
public class MiaClasseRunnable implements Runnable {
        …
        public void run() {
                //faccio qualcosa
                …
        }
}
```

```
MiaClasse e = new MiaClasseRunnable(…);
…
Thread mioThread = new Thread(e);
mioThread.start();
…
```

C'è un solo flusso di esecuzione

Da qui in poi ci sono due flussi

Esercizio

- Realizzare una classe java Runnable che rappresenta un eseguibile 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
- Soluzione:
  - esercizioCaratteri01

Soluzione

```
public class PrintThread implements Runnable {
    private final char c;

    public PrintThread(char c) {
        this.c = c;
    }

    public void run() {
        for (int i=0;i<100;i++)
            System.out.print(c);
    }

}

public class Main {

    public static void main(String[] args) {
        Thread t1 = new Thread(new PrintThread('0'));
        Thread t2 = new Thread(new PrintThread('1'));
        t1.start();
        t2.start();
    }
}
```

Esecuzione

Output:

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000001111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
```

Cosa è successo? sono stati eseguiti in sequenza?

- I due thread sono stati eseguiti concorrentemente *non* is sequenza, ma nella particolare implementazione della JVM l'esecuzione del programma è stata troppo veloce: il primo thread è terminato prima che potesse esseredato accesso all'output al secondo thread.

Per mostrare tale interleaving facciamo "dormire" (con **`Theread.sleep()`**) i thread di tanto in tanto …

Soluzione con sleep

```
public class PrintThread implements Runnable {
    private final char c;

    public PrintThread(char c) {
        this.c = c;
    }

    public void run() {
        for (int i=0;i<100;i++){
            System.out.print(c);
            try {
                    Thread.sleep((long)(Math.random() * 10));
            } catch (InterruptedException e) {}
        }
    }

}

public class Main {

    public static void main(String[] args) {
        Thread t1 = new Thread(new PrintThread('0'));
        Thread t2 = new Thread(new PrintThread('1'));
        t1.start();
        t2.start();
    }
}
```

Esecuzione

Output:

```
01000100101010101101001100011010100100100101101010100101100100110110010
11110100100101100110011010110100110011001010001110110100111101101010010
10110110011001010000101011001110011101101010101101001100
```

Output:

```
01000101010001001010011000101010101100100011110001110010011000100010110
10111011010011101010110101100101100010100010101000101011100000010111001
01101010110010010100100110101101010110110010111111111111
```

Output:

```
01010100010110011010011100000111001100100101101011101010010101000100100
10110101010010100110011100010110010100100110101000101011000011010101011011
10001110001100011011000110011100100111010010011111111111
```

Cosa è successo?

- Ora ad ogni esecuzione la JVM ha l'occasione di fare interleaving dei due threads in modo diverso.

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()`

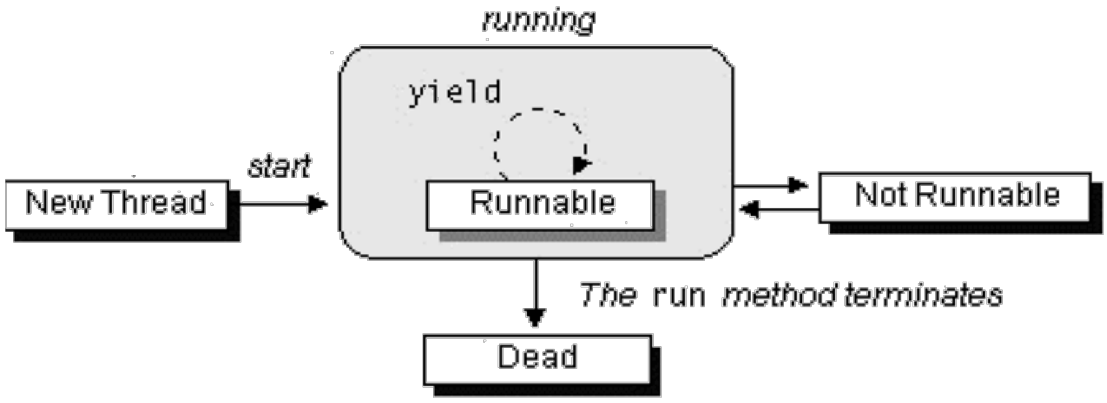
Mette il thread in pausa per il tempo specificato

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

- New Thread
- Invocazione del metodo start()
- Attivo
  - Running
  - Not Running
- Dead

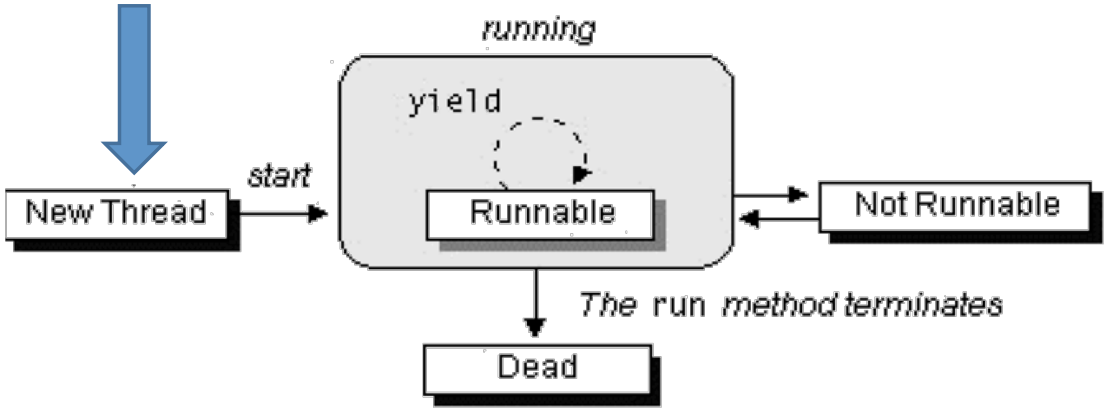


Ciclo di vita di un thread

Creazione di un nuovo oggetto Thread

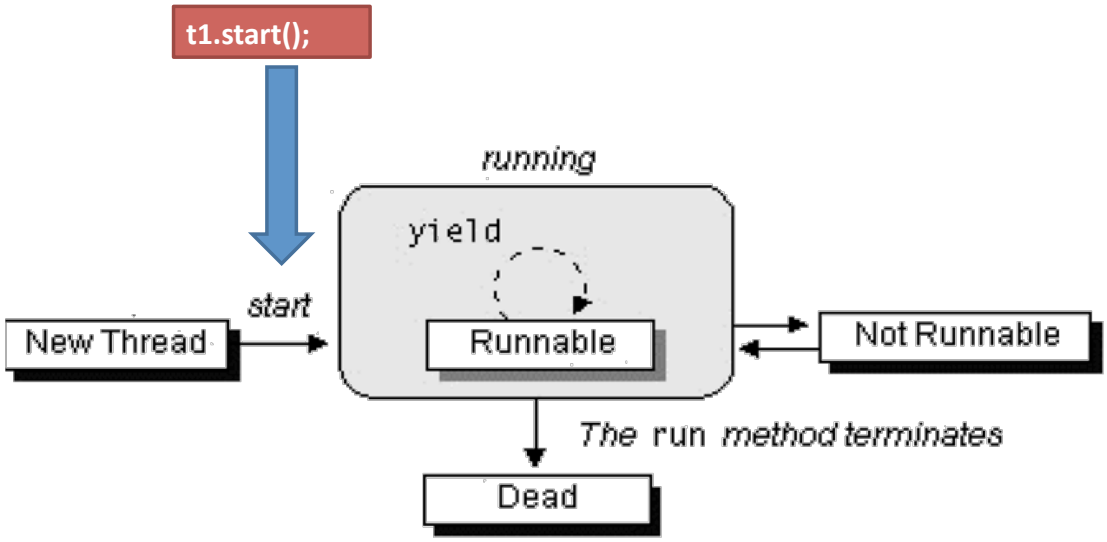
```
public void run() {
  for (int i=0;i<100;i++){
    System.out.print(c);
    try {
      Thread.sleep((long)(Math.random() * 10));
    } catch (InterruptedException e) {}
  }
}
```

```
Thread t1 = new Thread(new PrintThread('0'));
```



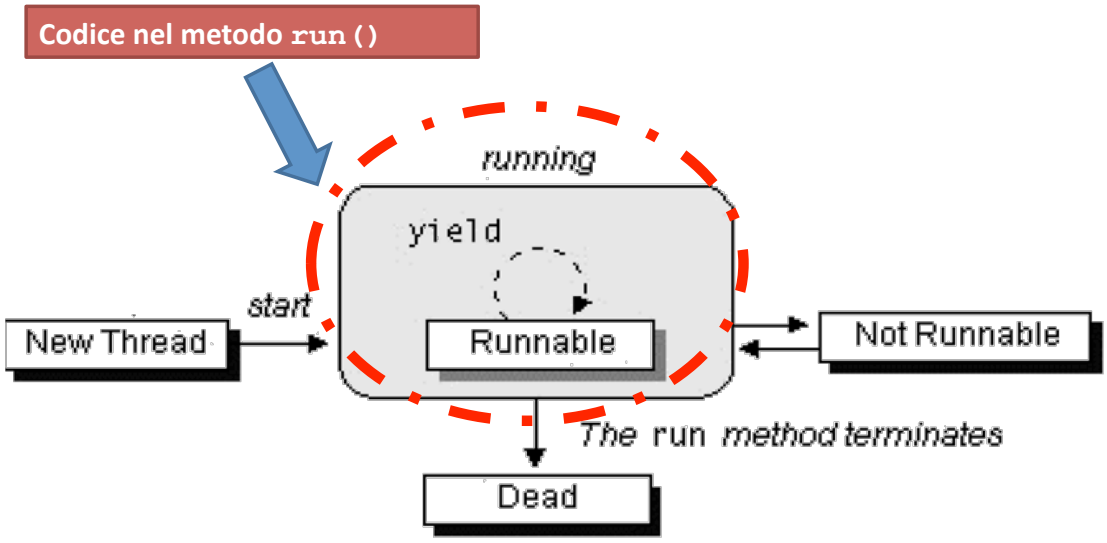
Ciclo di vita di un thread

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



Ciclo di vita di un thread

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

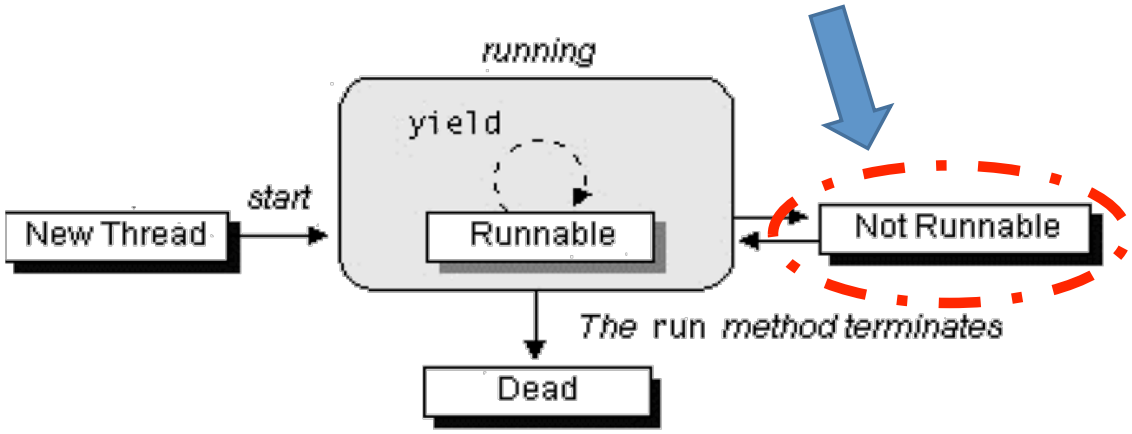


Ciclo di vita di un thread

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**

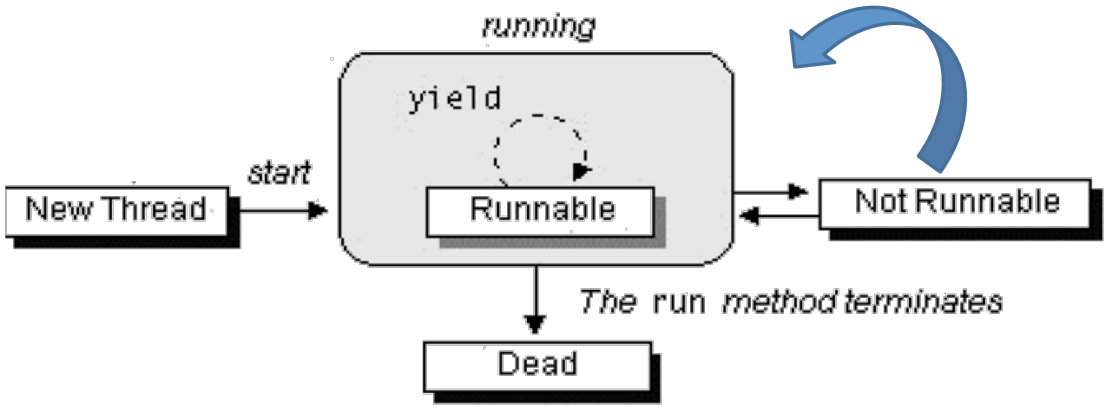
```
Thread.sleep((long)(Math.random() * 10));
```



Ciclo di vita di un thread

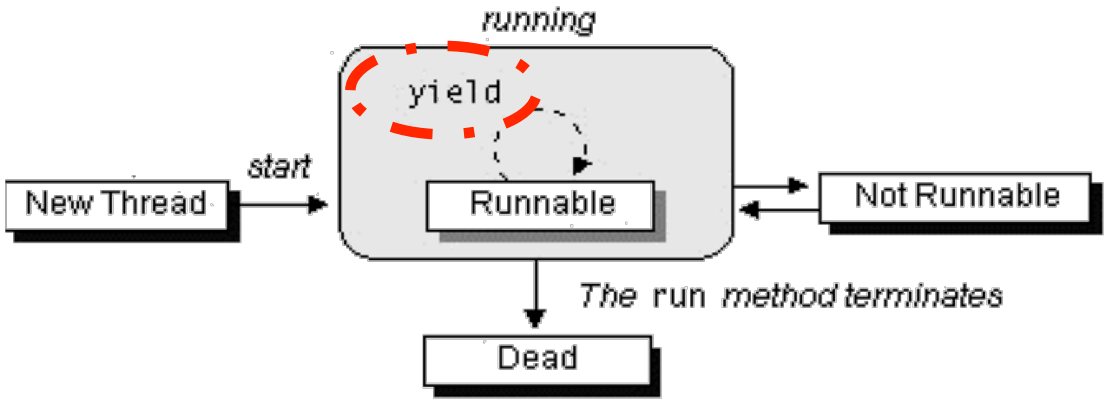
Per ogni ingresso nello stato ***Not Runnable***, c'è una precisa e definita situazione che lo riporta in ***Runnable***

- 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

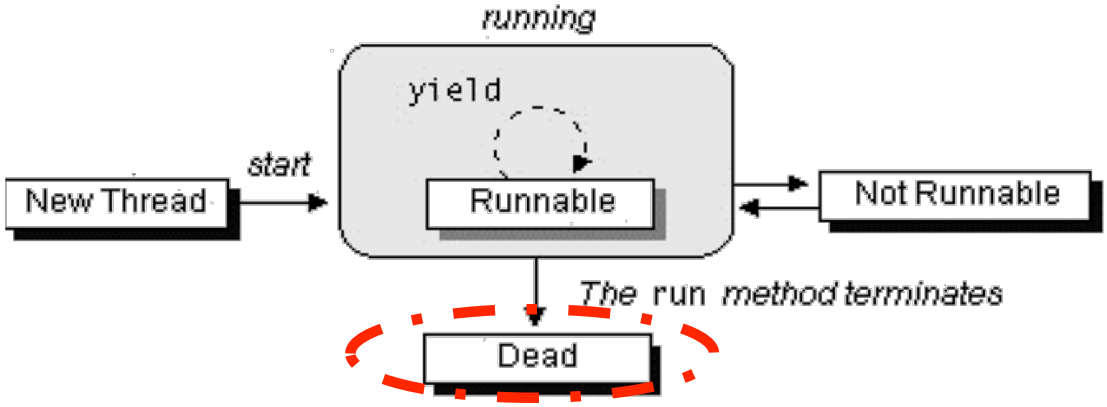


Ciclo di vita di un thread

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

```
public void run() {
  for (int i=0;i<100;i++){
    System.out.print(c);
    try {
      Thread.sleep((long)(Math.random() * 10));
    } catch (InterruptedException e) {}
  }
}
```

Il metodo run() termina

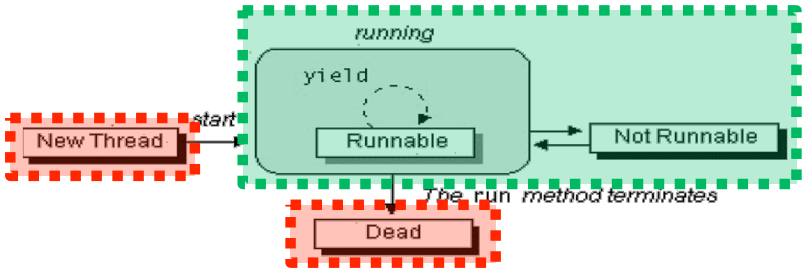


Ciclo di vita di un thread

`public boolean isAlive()` in Thread

- Ritorna **`false`** se il thread è non è mai stato avviato (attraverso **`start()`**) 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

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

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

- **`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

Gestione dei thread

- **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)
- **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)

- **In Java si ha una gestione preemptive**
  - A selezione di thread RUNNABLE con priorità più alta
  - SOLO UN THREAD eseguito tra thread di pari priorità (scelta round-robin)
  - SUDDIVISIONE DI TEMPO nei thread implementati nella Java VM non è specificata ma si appoggia al sistema operativo

Nota sulla suddivisione di tempo

- Le specifiche della Java Virtual Machine non indicano come i thread Java debbano essere mappati su thread nativi del s.o.

- 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

Priorità dei thread

- `public final static int MAX_PRIORITY = 10`
- `public final static int NORM_PRIORITY = 5`
- `public final static int MIN_PRIORITY = 1`

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

Nota sulla priorità

- La corrispondenza tra le priorità dei thread Java e le priorità thread nativi del s.o. dipendente da piattaforma
- Possono esistere implementazioni di thread nativi che non tengono conto delle priorità dei thread Java (es. Linux in alcune modalità)
- Adottare tecniche “sagge” per garanzia di portabilità (vedi dopo!)

Thread: alcune tecniche

- Non usare cicli infiniti o prevedere nel ciclo:
  - operazioni di *I/O*
  - fasi di *sleep*
  - operazioni di *coordinamento tra thread*

- Richiamare il metodo *yield nel caso di* thread con operazioni CPU-bound
  - rilascio volontario di CPU

- Mai fare affidamento completo sul timeslicing nè solo sulla priorità per conseguire portabilità

Scope delle variabili nei thread

- 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
- Variabili di istanza sono comuni a tutti i thread che accedono a quell'istanza
- Variabili di classe sono comuni a tutti i thread nel runtime

- 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
      …
  }
}
```

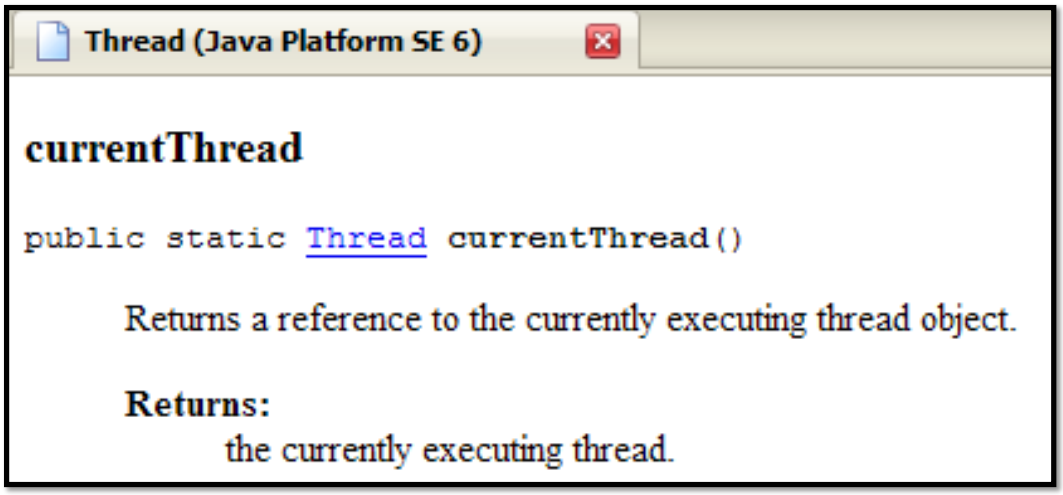
Esempio thread con var locali

```
public class RunThreads implements Runnable {

   public void run() {
        for (int i = 0; i < 25; i++) {
                String threadName = Thread.currentThread().getName();
                System.out.println(threadName + " is running");
        }
   }
}
public class Main {
   public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()
                        + " is running");
        Thread alpha = new Thread(new RunThreads());
        Thread beta = new Thread(new RunThreads());
        alpha.setName("Alpha thread");
        beta.setName("Beta thread");
        alpha.start();
        beta.start();
        System.out.println(Thread.currentThread().getName()
                        + " stops running");
   }
}
```

Metodo currentThread

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

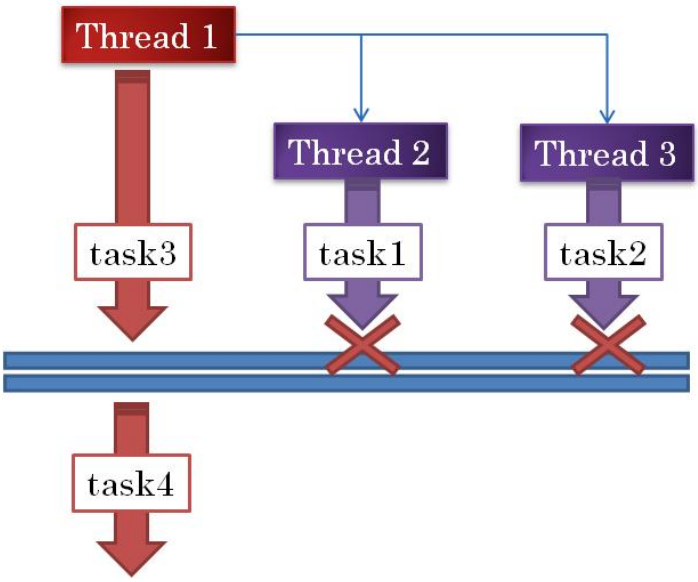


SINCRONIZZAZONE

Sincronizzazione tra thread

- **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

Sincronizzazione



Il thread 1

- avvia i thread 2 e 3 (che eseguono rispettivamente il task1 e il task2),
- esegue il task 3,
- e si mette in attesa della terminazione dei thread 2 e 3, prima di continuare con il task 4

Join

Il metodo fondamentale per la sincronizzazione tra thead in Java è **`join()`:**

`void join()`

- Quando il thread corrente **`t1`** invoca **`t2.join()`** si mette in attesa che il thread associato all'oggetto **`t2`** termina (passo allo stato dead); quando **`t2`** termina allora **`t1`** riparte proseguendo con la sua istruzione successiva.

`void join(long mills)`

- Attende al più **`mills`** millisecondi la morte del thread generato dall' oggetto di invocazione

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 (1/3)

Consideriamo il seguente classe

```
public class CountDown implements Runnable {
    private String id;
    private int countDown;

    public CountDown(String nome, int countDownPartenza) {
          id = nome;
          countDown = countDownPartenza;
    }

    public String status() {
          return id + ": " + (countDown > 0 ? countDown : "Go!") + "\n";
    }

    public void run() {
          while (countDown >= 0) {
                    System.out.print(status());
                    countDown--;
                    try {
                              Thread.sleep(100);
                    } catch (InterruptedException e) {
                    }
          }
    }
}
```

Esempio Join (2/3)

Consideriamo un main sequenziale (cioé con un unico thread)

```
public class MainSequenziale {
    public static void main(String[] args) {
          CountDown h = new CountDown("Huston", 10);
          CountDown c = new CountDown("CapeCanaveral", 10);
          h.run(); // NB invoco h.run() sul thread del main
          c.run(); // NB invoco c.run() sul thread del main
          System.out.println("Si parte!!!");
    }
}
```

Esempio Join (3/3)

Una esecuzione di `MainSequenziale`:

- Huston: 10
- Huston: 9
- Huston: 8
- Huston: 7
- Huston: 6
- Huston: 5
- Huston: 4
- Huston: 3
- Huston: 2
- Huston: 1
- Huston: Go!
- CapeCanaveral: 10
- CapeCanaveral: 9
- CapeCanaveral: 8
- CapeCanaveral: 7
- CapeCanaveral: 6
- CapeCanaveral: 5
- CapeCanaveral: 4
- CapeCanaveral: 3
- CapeCanaveral: 2
- CapeCanaveral: 1
- CapeCanaveral: Go!
- Si parte!!!

Esempio Join (2/3b)

Consideriamo un main concorrente ma senza sincronizzazione (no join)

```
public class MainConcorrenteNoJoin {
   public static void main(String[] args) {
        CountDown h = new CountDown("Huston", 10);
        CountDown c = new CountDown("CapeCanaveral", 10);
        Thread t1 = new Thread(h);
        Thread t2 = new Thread(c);
        t1.start(); // NB invoco h.run() sul thread t1
        t2.start(); // NB invoco c.run() sul thread t2
        System.out.println("Si parte!!!");
   }
}
```

Esempio Join (3/3b)

Una esecuzione di `MainConcorrenteNoJoin`:

```
Huston: 10
CapeCanaveral: 10
Si parte!!!
Huston: 9
CapeCanaveral: 9
CapeCanaveral: 8
Huston: 8
CapeCanaveral: 7
Huston: 7
CapeCanaveral: 6
Huston: 6
CapeCanaveral: 5
Huston: 5
CapeCanaveral: 4
Huston: 4
CapeCanaveral: 3
Huston: 3
CapeCanaveral: 2
Huston: 2
CapeCanaveral: 1
Huston: 1
CapeCanaveral: Go!
Huston: Go!
```

Esempio Join (2/3c)

Consideriamo infine un main concorrente con sincronizzazione (join)

```
package countdown;

public class MainConcorrenteJoin {
   public static void main(String[] args) {
         CountDown h = new CountDown("Huston", 10);
         CountDown c = new CountDown("CapeCanaveral", 10);
         Thread t1 = new Thread(h);
         Thread t2 = new Thread(c);
         t1.start(); // NB invoco h.run() sul thead t1
         t2.start(); // NB invoco c.run() sul thead t2
         try {
                  t1.join();
                  t2.join();
         } catch (InterruptedException e) {
                  // qui va codice da eseguire
                  // se il main viene risvegliato da un interrupt
                  // prima di aver fatto join
         }
         System.out.println("Si parte!!!");
   }
}
```

Esempio Join (3/3c)

Una esecuzione di `MainConcorrenteJoin`:

```
Huston: 10
CapeCanaveral: 10
Huston: 9
CapeCanaveral: 9
Huston: 8
CapeCanaveral: 8
Huston: 7
CapeCanaveral: 7
Huston: 6
CapeCanaveral: 6
Huston: 5
CapeCanaveral: 5
Huston: 4
CapeCanaveral: 4
Huston: 3
CapeCanaveral: 3
Huston: 2
CapeCanaveral: 2
Huston: 1
CapeCanaveral: 1
Huston: Go!
CapeCanaveral: Go!
Si parte!!!
```

Interruzione di thread

- Un thread **`t1`** può mandare una richiesta di interruzione ad un altro thread t2 invocando il metodo **`t2.interrupt()`** della classe **`Thread`**
- In realtà il thread **`t1`** eseguendo l'istruzione **`t2.interrupt()`** segnala la richiesta di interruzione al thread **`t2`** settando uno specifico flag in tale oggetto
- In generale il thread interrotto **`t2`** non serve la richiesta di interruzione immediatamente:
  - Se è in esecuzione (running) può testare il flag attraverso l'istruzione **`Thread.interrupted()`** che verifica appunto se il flag è settato a **`true.`** Si noti che se non fa un test esplicito allora continua senza accorgersi dell'interruzione.
  - Se invece è in attesa (non running) –o appena passa in questo stato– per esempio sta eseguendo una **`sleep()`** o una **`wait()`**, allora solleva automaticamente una eccezione **`java.lang.InterruptedException`**, reseta il flag a **`false`**, e passa a gestire il flusso d'errore (nota tutte le operazioni che mettono nello stato di attesa richiedono la gestione di **`InterruptedException`**
- Si noti che questo comportamento induce ad usare gli **`interrupt()`** per la gestione di errori, e non per la sincronizzazione nel flusso corretto d'esecuzione.

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 (2/2)

```
public class Main {
   public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();
        try {
               Thread.sleep(2000);
        } catch (InterruptedException x) {
        }

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

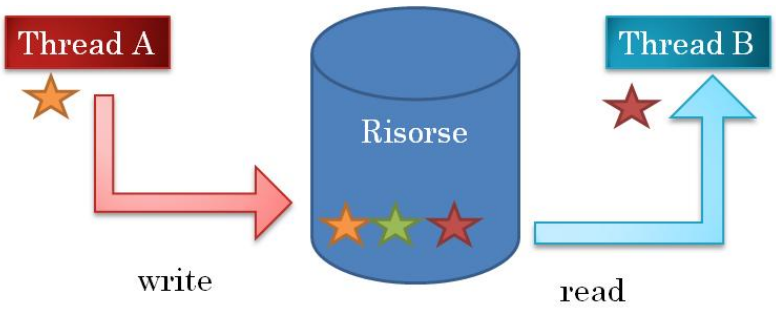
Cooperazione tra thread

- **Cooperazione:** si tratta di una forma di sincronizzazione 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

Cooperazione

I thread condividono risorse comuni su cui lavorare

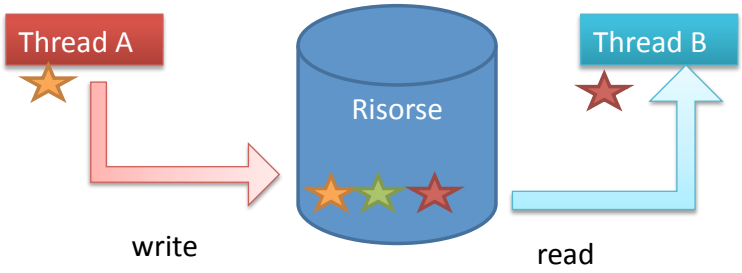
- I thread oltre a sincronizzarsi si scambiano messaggi
- L'output di un thread può essere l'input per un altro e occorrono strutture condivise che permettano la comunicazione tra thread



Cooperazione

- 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.

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



Esempio accesso a dati condivisi (1/4)

Consideriamo la seguente classe Contatore

```
public class Contatore {
   private int valore;

   public void incrementa() {
        int tmp = valore;
        try { Thread.sleep(10);
        } catch (InterruptedException ex) {}
        valore = tmp + 1;
        System.out.println(
                Thread.currentThread().getName() +
                                ": il contatore segna " + valore);

   }

   public int getValore() {
        return valore;
   }

}
```

Esempio accesso a dati condivisi (2/4)

Consideriamo poi il seguente processo che conta usando un Contatore

```
public class ProcessoConta implements Runnable {
   private Contatore contatore;

   public ProcessoConta(Contatore c) {
       contatore = c;
   }

   public void run() {
       for (int i = 0; i < 10; i++) {
              contatore.incrementa();
              try { Thread.sleep(10);
              } catch (InterruptedException ex) {}

       }
   }
}
```

Esempio accesso a dati condivisi (3/4)

Consideriamo infine il seguente main che genera due processi conta che condividono il contatore

```
public class Main {
   public static void main(String[] args) {
       Contatore cnt = new Contatore();
       Thread a = new Thread(new ProcessoConta(cnt));
       Thread b = new Thread(new ProcessoConta(cnt));
       a.start();
       b.start();
   }
}
```

Esempio accesso a dati condivisi (4/4)

Eseguendo il main otteniamo il seguente risultato! Il contatore sbaglia!!!

- Thread-0: il contatore segna 1
- Thread-1: il contatore segna 1
- Thread-0: il contatore segna 2
- Thread-1: il contatore segna 2
- Thread-1: il contatore segna 3
- Thread-0: il contatore segna 3
- Thread-1: il contatore segna 4
- Thread-0: il contatore segna 4
- Thread-1: il contatore segna 5
- Thread-0: il contatore segna 5
- Thread-1: il contatore segna 6
- Thread-0: il contatore segna 6
- Thread-1: il contatore segna 7
- Thread-0: il contatore segna 7
- Thread-1: il contatore segna 8
- Thread-0: il contatore segna 8
- Thread-1: il contatore segna 9
- Thread-0: il contatore segna 9
- Thread-1: il contatore segna 10
- Thread-0: il contatore segna 10

Esempio accesso a dati condivisi (1/4b)

Consideriamo la seguente classe Contatore

```
public class Contatore {
   private int valore;

   public synchronized void incrementa() {
        int tmp = valore;
        try { Thread.sleep(10);
        } catch (InterruptedException ex) {}
        valore = tmp + 1;
        System.out.println(
                Thread.currentThread().getName() +
                        ": il contatore segna " + valore);

   }

   public synchronized int getValore() {
        return valore;
   }

}
```

Esempio accesso a dati condivisi (4/4b)

Eseguendo il main otteniamo il seguente risultato! Ora con il contatore sychronized i risultati sono corretti!!!

- Thread-0: il contatore segna 1
- Thread-1: il contatore segna 2
- Thread-0: il contatore segna 3
- Thread-1: il contatore segna 4
- Thread-0: il contatore segna 5
- Thread-1: il contatore segna 6
- Thread-0: il contatore segna 7
- Thread-1: il contatore segna 8
- Thread-0: il contatore segna 9
- Thread-1: il contatore segna 10
- Thread-0: il contatore segna 11
- Thread-1: il contatore segna 12
- Thread-0: il contatore segna 13
- Thread-1: il contatore segna 14
- Thread-0: il contatore segna 15
- Thread-1: il contatore segna 16
- Thread-0: il contatore segna 17
- Thread-1: il contatore segna 18
- Thread-0: il contatore segna 19
- Thread-1: il contatore segna 20

Monitor (1/2)

- "Custode" di un oggetto che determina l'accesso a suoi comportamenti (metodi e sezioni di codice)
- Mutua esclusione su gruppi di procedure
  - Un thread alla volta in esecuzione (altri thread sospesi)
- 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
- Monitor oggetto rilasciato dal thread a fine esecuzione di metodo (o sezione) synchronized

Monitor (2/2)

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) {…}`
   - 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!

Forme di codice synchronized

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

Guarda il monitor su oggetto `this`

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

Guardano il monitor su oggetto `.class` associato alla classe

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

Guarda il monitor su oggetto `ogg`

Metodi non synchronized

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

Esempio accesso a dati condivisi (ancora)

- Consideriamo ancora l'esempio del Contatore. Ma questa volta nel metodo **`run()`** del **`ProcessoConta`** si accede due volte all'oggetto contatore con le due seguenti invocazioni:
  - `contatore.incrementa()`
  - `contatore.getValore()`

Esempio accesso a dati condivisi (1/4c)

Consideriamo la seguente classe Contatore

```
public class Contatore {
   private int valore;

   public void incrementa() {
        int tmp = valore;
        try { Thread.sleep(10);
        } catch (InterruptedException ex) {}
        valore = tmp + 1;
   }

   public int getValore() {
        return valore;
   }

}
```

Esempio accesso a dati condivisi (2/4c)

Consideriamo poi il seguente processo che conta usando un Contatore

```
public class ProcessoConta implements Runnable {
   private Contatore contatore;

   public ProcessoConta(Contatore c) {
        contatore = c;
   }

   public void run() {
        for (int i = 0; i < 10; i++) {
                contatore.incrementa();
            try { Thread.sleep(10);
              } catch (InterruptedException ex) {}
                System.out.println(
   Thread.currentThread().getName() +
   ": il contatore segna " + contatore.getValore()
                                    );

        }
   }
}
```

Esempio accesso a dati condivisi (4/4c)

Cosa è successo? L'incremento e la stampa non sono eseguite consecutivamente!

- Thread-0: il contatore segna 2
- Thread-1: il contatore segna 3
- Thread-0: il contatore segna 4
- Thread-1: il contatore segna 5
- Thread-0: il contatore segna 6
- Thread-1: il contatore segna 7
- Thread-0: il contatore segna 8
- Thread-1: il contatore segna 9
- Thread-0: il contatore segna 10
- Thread-1: il contatore segna 11
- Thread-0: il contatore segna 12
- Thread-1: il contatore segna 13
- Thread-0: il contatore segna 14
- Thread-1: il contatore segna 15
- Thread-0: il contatore segna 16
- Thread-1: il contatore segna 17
- Thread-0: il contatore segna 18
- Thread-1: il contatore segna 19
- Thread-0: il contatore segna 20
- Thread-1: il contatore segna 20

Deadlock

- La concorrenza può portare a diversi problemi che non si manifestano in applicazioni sequenziali. Il più impartante tra questi è il deadlock
- **Deadlock**: due thread devono accedere a due oggetti ma ciascuno acquisisce il lock su uno dei due e poi aspetta il rilascio dell'altro oggetto da parte dell'altro thread, il che non avverrà mai perché a sua volta l'altro thread è in attesa del rilascio del primo oggetto da parte del primo thread.

- Nel seguito si mostra un esempio di deadlock in Java

Esempio deadlock (1/3)

Consideriamo la seguente risorsa. Si noti che in maxVal this accede a un'altra risorsa ra (nel thread corrente)

```
public class Risorsa {
    private String nome;
    private int valore;

    Risorsa(String id, int v) {
          nome = id;
          valore = v;
    }

    public synchronized int getValore() {
          return valore;
    }

    public synchronized String getNome() {
          return nome;
    }

    public synchronized Risorsa maxVal(Risorsa ra) {
          try {
                Thread.sleep(100);
          } catch (InterruptedException e) {
          }
          int va = ra.getValore();
          if (valore >= va)
                return this;
          else
                return ra;
    }
}
```


Esempio deadlock (2/3)

Confronto è un Runnable che in run() invoca maxVal() sulla r1, e tale metodo a sua volta invoca getValore() su r2.

```
public class Confronto implements Runnable {
    Risorsa r1;
    Risorsa r2;

    Confronto(Risorsa r1, Risorsa r2) {
          this.r1 = r1;
          this.r2 = r2;
    }

    public void run() {
          System.out.println(Thread.currentThread().getName() + "
    begins");
          Risorsa rm = r1.maxVal(r2);
          String rn = rm.getNome();
          System.out.println(Thread.currentThread().getName()
                                      + ": la risorsa più grande è " + rn);
          System.out.println(Thread.currentThread().getName() + " ends");
    }

}
```

Esempio deadlock (3/3)

```
public class MainSingoloConfronto {
    public static void main(String[] args) throws InterruptedException {
          Risorsa r1 = new Risorsa("alpha",5);
          Risorsa r2 = new Risorsa("beta",20);
          System.out.println(Thread.currentThread().getName() + " begins");
          Confronto c1 = new Confronto(r1, r2);
          Thread t1 = new Thread(c1);
          t1.start();
          Thread.sleep(10000); //aspetta 10 sec
          System.out.println(Thread.currentThread().getName() + " ends");
    }
}
```

Consideriamo prima un main sequenziale. Tutto va come previsto.

Output:

```
main begins
Thread-0 begins
Thread-1 begins
main ends
```


Esempio deadlock (3/3bis)

Consideriamo ora un main multithread. Abbiamo deadlock!

```
public class MainDeadlock {
    public static void main(String[] args) throws InterruptedException {
          Risorsa r1 = new Risorsa("alfa",5);
          Risorsa r2 = new Risorsa("beta", 20);
          System.out.println(Thread.currentThread().getName() + " begins");
          Confronto c1 = new Confronto(r1,r2);
          Confronto c2 = new Confronto(r2,r1);
          Thread t1 = new Thread(c1); //t1.setDaemon(true);
          Thread t2 = new Thread(c2); //t2.setDaemon(true);
          t1.start();
          t2.start();
          Thread.sleep(10000);
          System.out.println(Thread.currentThread().getName() + " ends");
    }
}
```

Output:

```
main begins
Thread-0 begins
Thread-1 begins
main ends
I due thread Thread-0 e Thread-1 sono in deadlock
```

Esempio deadlock (discussione)

Vediamo come si genera il deadlock nel main multithread.!

- **`t1`** chiama il metodo **`synchronized maxRis`** su **`r1`**, prendendo il monitor di r1. Intanto t2 chiama **`maxRis`** su **`r2`** prendendo il monitor di **`r2`**.

- Il thread **`t1`**, eseguendo **`maxRis`** su **`r1`**, invoca il metodo **`synchornized getValore`** e su **`r2`**. Ma il monitor di r2 è in possesso di t2, quindi t1 si mette in attesa.

- Intanto **`t2`**, eseguendo **`maxRis`** su **`r2`**, invoca il metodo **`synchornized getValore`** su **`r1`**. Ma il monitor di **`r1`** è in possesso di **`t1`**, quindi anche **`t2`** si mette in attesa.

- **`t1`** aspetta **`t2`** che aspetta **`t1`**. Deadlock !!!

Esempio deadlock (discussione)

- Si noti che il **`main`** dopo 10 sec termina, mentre t1 e t2 rimangono running ma bloccati sui monitor di r2 e r1 rispettivamente.

- *NB: Se vogliamo che al termine del main terminino forzatamente anche t1 e t2, nel main dopo la creazione dei thread e prima di dargli lo start dobbiamo dichiararli come "demoni" attraverso l'istruzione setDeamon.Cioè nel main scriviamo:*
  - *`t1.setDaemon(true);`*
  - *`t2.setDaemon(true);`*

  I thread dichiarati demoni terminano forzatamente dopo la terminazione dei thread non demoni.

COORDINAMENTO

Limiti della sola sincronizzazione

- 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?

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

- 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 coordinarne l'utilizzo

Coordinamento

- Condizioni all'interno di un metodo sincronizzato

```
while (!condizione desiderata) {
    wait();
}
```

  Si noti l'uso del **`while`**: quando il thread riceve una **`notify`** non è detto che la condizione desiderata sia soddisfatta, in caso non lo sia il thread si rimette in **`wait`**
- Attesa
  - Thread posto in pausa nella coda di attesa del monitor dell'oggetto (stato NOT RUNNABLE)
- 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

- **`public final void wait()`**
  - Determinano l'attesa del thread che detiene il monitor dell'oggetto di invocazione del **`wait`** fino all'invocazione di **`notifyAll()`** o **`notify()`**, su tale oggetto (alla sua interruzione attraverso l'invocazione di **`interrupt()`**). Nel mettersi in attesa il thread rilascia il monitor.
- **`public final void notifyAll()`**
  - Risveglia tutti i thread in attesa sul monitor dell'oggetto di invocazione di **`notifyAll()`**. Tale thread si attivano e si mettono ad aspettare il proprio turno sul monitor di tale oggetto di invocazione
- **`public final void notify()`**
  - Si comporta come **`notifyAll()`**, ma risvegliando a caso solo uno dei thread in attesa sul monitor dell'oggetto di invocazione

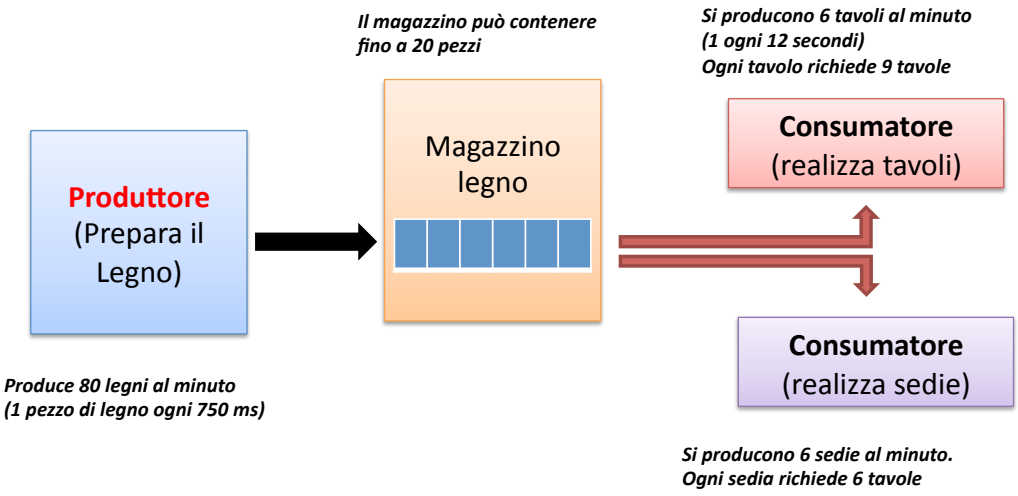
- *I metodi **`wait, notifyAll, notify,`** sono invocabili sono all'interno di un blocco sincronizzato poiché operano sul monitor dell'oggetto di invocazione sul quale hanno correntemente il lock.*

Method Summary

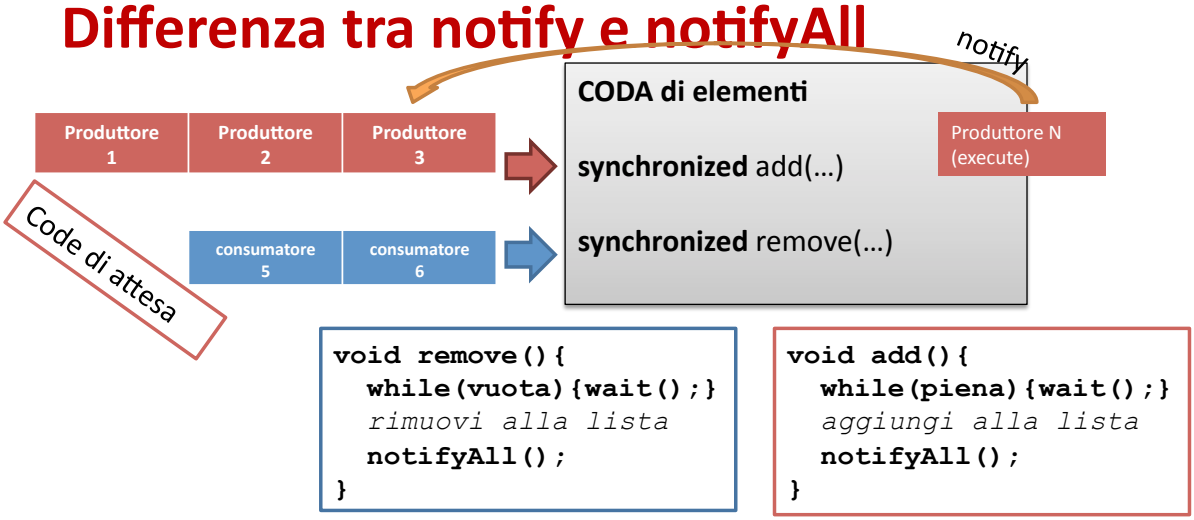
- void **notify**()
  Wakes up a single thread that is waiting on this object's monitor.
- void **notifyAll**()
  Wakes up all threads that are waiting on this object's monitor.
- void **wait**()
  Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- 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.
- void **wait**(long timeout, int nanos)

Esempio

- 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



Supponiamo che dopo l'esecuzione del metodo **add** da parte del "***produttore N***" la coda si riempia, viene eseguito **notify** **(**non un **notifyAll)** 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 utilizza **notifyAll** vengono risvegliati tutti i thread, uno dopo l'altro, e quindi prima o poi toccherà anche ad un thread consumatore

Esempio coda limitata condivisa

- Consideriamo una coda limitata (bounded queue che permetta l'accesso concorrente.
- In rimozione del primo elemento qualora la coda sia vuota mette il thread corrente in **`wait`**
- Similmente all'aggiunta di un elemento in coda se la coda stessa è piena allora mette il thread corrente in **`wait`**
- Ogni volta che si completa una operazione di inserimento o rimozione viene dato il **`notifyAll`**
- Faremo uso della coda attraverso processi **produttore** e processi **consumatore**.

Esempio coda limitata condivisa (1/4)

```
public class BoundedQueue<T>
    private List<T> elements;
    private int maxSize;
    public BoundedQueue(int capacity) {
          elements = new ArrayList<T>();
          maxSize = capacity;
    }

    public synchronized T remove() throws InterruptedException {
          while (elements.size() == 0)
                   wait();
          T r = elements.remove(0);
          notifyAll();
          return r;
    }

    public synchronized void add(T newElem) throws InterruptedException {
          while (elements.size() == maxSize)
                   wait();
          elements.add(newElem);
          notifyAll();
    }
}
```

Qusto è il codice della coda limitata

*Si noti che la coda è **generic** rispetto al tipo degli elementi.*

Esempio coda limitata condivisa (2/4)

```
public class Producer implements Runnable {

    private String greeting;
    private BoundedQueue<String> queue;
    private int greetingCount;

    private static final int DELAY = 10;

    public Producer(String aGreeting, BoundedQueue<String> aQueue, int count) {
          greeting = aGreeting;
          queue = aQueue;
          greetingCount = count;
    }

    public void run() {
          try {
                   int i = 1;
                   while (i <= greetingCount) {
                            queue.add(i + ": " + greeting);
                            i++;
                            Thread.sleep((int) (Math.random() * DELAY));
                   }
          } catch (InterruptedException exception) {
          }
    }
}
```

Codice del **produttore**: produce stringhe (greetings, cioè frasi di saluto)

Esempio coda limitata condivisa (3/4)

```
public class Consumer implements Runnable {

   private BoundedQueue<String> queue;
   private int greetingCount;

   private static final int DELAY = 10;

   public Consumer(BoundedQueue<String> aQueue, int count) {
         queue = aQueue;
         greetingCount = count;
   }

   public void run() {
         try {
                  int i = 1;
                  while (i <= greetingCount) {
                           Object greeting = queue.remove();
                           System.out.println(greeting);
                           i++;
                           Thread.sleep((int) (Math.random() * DELAY));
                  }
         } catch (InterruptedException exception) {
         }
   }
}
```

Codice del **consumatore**: consuma stringhe (ancora greetings)

Esempio coda limitata condivisa (4/4)

```
public class ThreadTester {
   public static void main(String[] args) {
         BoundedQueue<String> queue = new BoundedQueue<String>(10);
         final int GREETING_COUNT = 1000;

         Runnable run1 = new Producer("Hello, World!", queue,    GREETING_COUNT);
         Runnable run2 = new Producer("Goodbye, World!", queue, GREETING_COUNT);
         Runnable run3 = new Consumer(queue, GREETING_COUNT);
         Runnable run4 = new Consumer(queue, GREETING_COUNT);

         Thread thread1 = new Thread(run1);
         Thread thread2 = new Thread(run2);
         Thread thread3 = new Thread(run3);
         Thread thread4 = new Thread(run4);

         thread1.start();
         thread2.start();
         thread3.start();
         thread4.start();
   }
}
```

Il **main** crea una coda, genera due produttori e due consumatori e li esegue concorrentemente

Esempio coda limitata condivisa

- *NB: il Java Collection Framework prevede versioni delle classiche strutture dati che gestiscono l'accesso concorrente proprio usando forme di* **`wait/notify`**.
- *Qui non ne abbaimo fatto uso per motivi didattici.*

I principali metodi dei Thread (1/2)

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 (2/2)

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.