CEFRIEL
Centro per la Formazione e la Ricerca
in Ingegneria dell'Informazione

Politecnico
di Milano

Java Multithreading

Programmazione concorrente in Java

Docente:
Diego Peroni
peroni@cefriel.it

Indice

- Definizione di thread
- Proprietà dei thread in Java
- Gestione delle priorità nei programmi multithreaded
- Gruppi di thread
- Accesso a variabili condivise
  - problematiche di sincronizzazione
  - paradigma dei monitor in Java
  - problemi connessi:
    - starvation
    - deadlock

Tasking e Threading

- In origine i calcolatori erano ***single-tasking***.
  - Veniva eseguito un job alla volta in modo *batch*.
- In seguito i S.O. sono divenuti ***multitasking***.
  - Multitasking: capacità di eseguire più job contemporaneamente.
  - Su un sistema desktop, ad esempio, possibilità di avere più programmi in esecuzione contemporaneamente (es. Netscape che scarica dei file, Word che stampa, il Solitario in esecuzione, …).
- I principali sistemi multitasking sono anche ***multithreading***.
  - Multithreading: possibilità di avere più flussi di controllo nell'ambito dello stesso processo.
  - Su un sistema desktop, un programma può eseguire più compiti contemporaneamente (es. Netscape che stampa una pagina, mentre ne scarica una seconda e mi fa compilare un form in una terza).

Processi e Thread

Processo (o task)

Programma in esecuzione, con il suo spazio di indirizzamento ed il suo stato.

Un processo è ad esempio un'istanza di Word o Netscape.

Detto anche "processo pesante", in riferimento al *contesto* (spazio di indirizzamento, stato) che si porta dietro.

Thread

Singolo flusso sequenziale di controllo all'interno di un processo.

Un processo può contenere più thread.

Tutti i thread di un processo condividono lo stesso spazio di indirizzamento.

Detto anche "processo leggero", perché ha un contesto semplice.

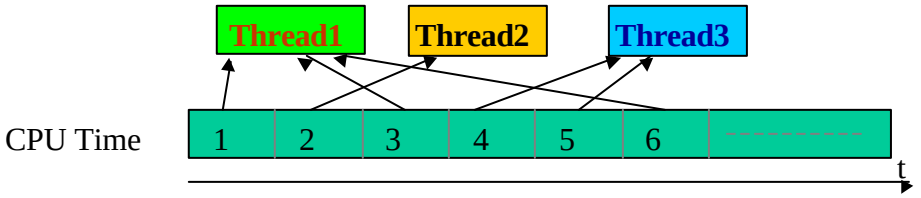
Parallelismo

- **Sistema parallelo**
  - Architettura in cui sono presenti più unità di elaborazione (CPU) sulle quali sono in esecuzione processi e thread.
  - In ogni istante di tempo, posso avere più di un processo o più di un thread fisicamente in esecuzione.
- **Sistema monoprocessore time-sliced**
  - Vi è una sola unità di elaborazione.
  - Il parallelismo di processi e thread viene simulato (*concorrenza*) allocando a ciascuno una frazione del tempo di CPU (*time slice*).
  - Allo scadere di ogni unità di tempo, il S.O. opera un cambio di contesto (*context switch*).
  - I thread sono processi leggeri perché il cambio di contesto è veloce.

Preemption

- In un **sistema non preemptive**, il cambio di contesto avviene quando il processo o il thread interrompe la propria esecuzione o volontariamente, o perché in attesa di un evento (input, output).
- In un **sistema preemptive**, allo scadere del time slice il processo o il thread viene forzatamente interrotto e viene operato il cambio di contesto.
- Es. Tutte le versioni di Unix, Windows 95/98 e Windows NT sono sistemi multitasking, multithreading, preemptive. Windows 3.1 è un sistema non preemptive.
- La *schedulazione* dei processi e dei thread può avvenire secondo diversi algoritmi (round-robin, con priorità, …).

Es.: Multithreading time-sliced



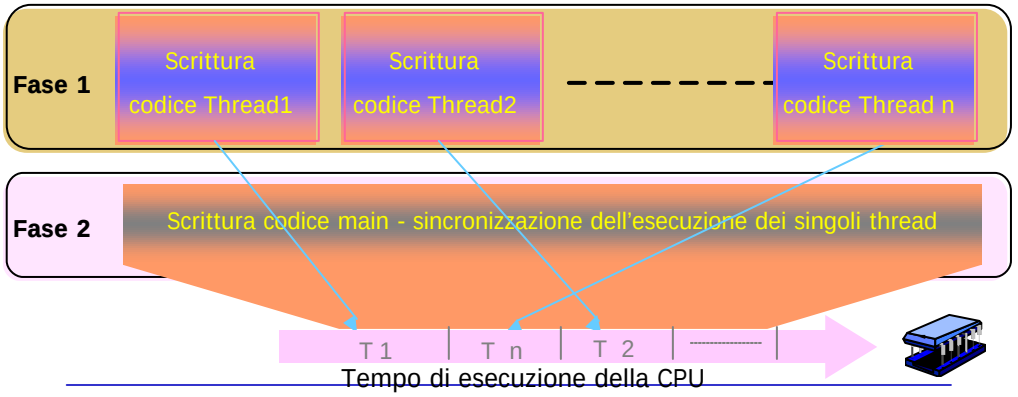
- Un programma Java deve prescindere dall'algoritmo di schedulazione sottostante.
- Il flusso di esecuzione viene controllato con meccanismi di *sincronizzazione*.

Perché i Thread ?

- Nella programmazione object oriented le classi strutturano il programma in sezioni indipendenti.
- È spesso necessario dividere il programma anche in "sotto-compiti" indipendenti.
  - Un programma può avere diverse funzioni concorrenti:
    - operazioni ripetute nel tempo ad intervalli regolari (es. animazioni);
    - esecuzione di compiti laboriosi senza bloccare la GUI del programma;
    - attesa di messaggi da un altro programma;
    - attesa di input da tastiera o dalla rete.
    - Es.: Si pensi ai compiti svolti da un web browser.
- L'alternativa al multithreading è il *polling*.
  - Il polling, oltre che scomodo da implementare, consuma molte risorse di CPU. È quindi estremamente inefficiente.

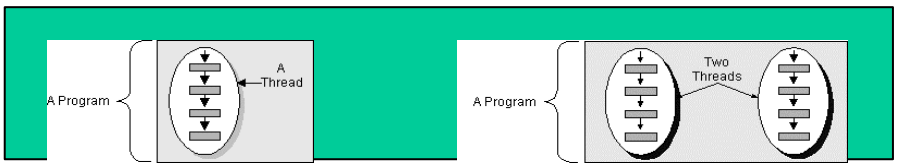
Scrittura di codice multithreaded

- Ci si concentra su due problemi distinti:
  - 1) Scrittura del codice di ciascun thread.
  - 2) Scrittura del codice del thread principale (main) che farà partire gli altri thread.



Definizione di Thread

- **Un thread è un singolo flusso di controllo sequenziale all'interno di un programma.**



  - Sequenziale: ogni thread ha un inizio, una fine, una sequenza ed ad ogni istante un solo punto di esecuzione;
  - Thread=lightweight process: utilizza il contesto del processo.
  - Execution context: il contesto di un singolo thread prevede inoltre, ad esempio, un proprio stack ed un proprio program counter.

Nuovo thread

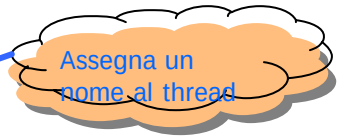
- Costruzione di una classe Thread (che estende la classe di base *java.lang.Thread*) che includa un ciclo for in grado di stampare il nome con accanto un numero progressivo.
- Esempio di output di un singolo thread (nome “Jamaica”):

0 Jamaica
1 Jamaica
2 Jamaica
...
9 Jamaica
DONE! Jamaica

Costruzione di una classe che istanzi e lanci due thread del tipo appena creato per vedere gli effetti di un’esecuzione concorrente.

Classe Thread

```
class ThreadSemplice extends Thread {
   public ThreadSemplice (String str) {
     super(str);
   }
   public void run() {
     for (int i = 0; i < 10; i++) {
       System.out.println( i + " " + getName() );
       try {
           Thread.currentThread().sleep((int)(Math.random()*1000));
       } catch (InterruptedException e) {
            System.out.println(“InterruptedException in ThreadSemplice: “ + e);
       }
     } // end for
     System.out.println("DONE! " + getName());
   }
}
```



Metodo main

```
class TwoThreadsTest {
   public static void main (String[] args) {
      // Istanzazione dei due thread
      ThreadSemplice ts1 = new
                  ThreadSemplice (“Jamaica”);
      ThreadSemplice ts2 = new
                  ThreadSemplice (“Fiji”);
      // Lancio dei due thread
      ts1.start();
      ts2.start();
   }
}
```

Output

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```


Attributi dei thread

- I thread Java sono implementati dalla classe *java.lang.Thread.*
- Le API di gestione dei thread sono *indipendenti dal particolare sistema operativo sottostante.*
  - Alcune implementazioni Java simulano i thread, altre invece si appoggiano ai thread nativi del sistema operativo.
- Ogni thread è caratterizzato da:
  - un corpo (thread body);
  - uno stato (thread state);
  - una priorità (thread priority);
  - un gruppo di appartenenza (thread group).
- Un thread può inoltre essere “Daemon Thread”.

Thread Body

- Il corpo di un thread è specificato dalle operazioni presenti nel metodo ***run()***.
  - Quando un thread viene avviato (metodo start()), avviene la biforcazione del flusso e viene invocato il metodo run().
- Java dispone di due modi diversi per creare un thread:
  - Creare una sottoclasse della classe *java.lang.Thread* e sovrascrivere il metodo run() (vuoto di default).
    - Vedi esempio precedente.
  - Creare una classe che implementa l'interfaccia *java.lang.Runnable*, e creare un nuovo Thread fornendo come target il riferimento ad un'istanza della classe creata.
    - Esempio successivo.

Interfaccia Runnable

- Interfaccia java.lang.Runnable

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

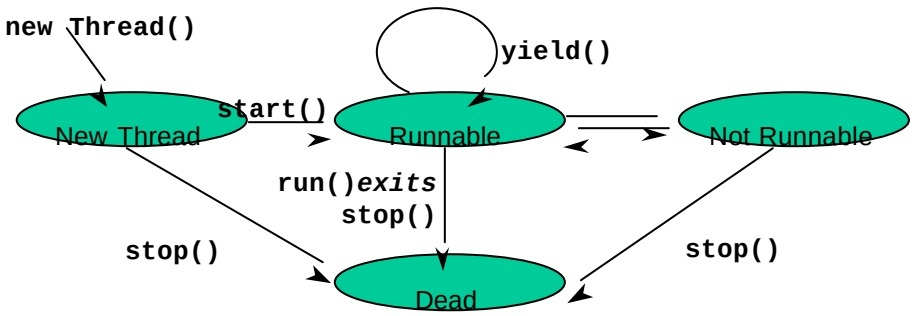
- class java.lang.Thread implements java.lang.Runnable
- **Regola generale**: Si implementa Runnable, anzichè derivare da Thread, quando la classe che contiene il metodo run() eredita già da un'altra classe.
  - Se ci fosse l'ereditarietà multipla, l'interfaccia Runnable non sarebbe necessaria.

Creazione di un thread tramite l'interfaccia Runnable

```
class PrimeRun extends ParentClass implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
            this.minPrime = minPrime;
     }
     public void run() {
            // calcola il numero primo più grande del numero assegnato…
     }
}

// Per creare un nuovo thread basterà eseguire:
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```


Stati di un Thread (1)



- **New Thread State**
  - *Thread nuovoThread = new Thread();*
  - non vengono allocate risorse per il thread in questo stato;
  - il sistema di scheduling non sa ancora della sua esistenza.

Stati di un Thread (2)

- **Runnable State**
  - *nuovoThread.start();*
  - La chiamata del metodo start() fa sì che il sistema:
    - *allochi le risorse necessarie al thread;*
    - *inserisca il nuovo thread nel meccanismo di scheduling;*
    - *chiami il metodo run() del thread (nel nuovo flusso di controllo).*
  - "Runnable" e non "Running": perchè?
    - *Sistema monoprocessore: solo un thread alla volta può essere running;*
    - *virtualmente però tutti i thread Runnable sono Running;*
    - *nello stato di Running le istruzioni del metodo run() vengono eseguite sequenzialmente.*
  - Yield(): provoca la temporanea sospensione del thread in corso, cosicché lo scheduler può cedere le risorse di calcolo della CPU ad altri eventuali Thread *Runnable*, che potrebbero così divenire *Running.*

Stati di un Thread (3)

- **Not Runnable State**
  - Passaggio da "Runnable" a "not Runnable" in quattro modi diversi:
    - ***1) suspend()***
    - ***2) sleep*(time in millisecondi)**
    - ***3) wait()*** -> il thread aspetta una condizione variabile
    - ***4)*** operazione di I/O bloccante
  - Esempio: per mettere un thread in sleep per 10 secondi (10000 ms)

```
try {
   myThread.sleep(10000);
} catch (InterruptedException e) {
   System.out.println("InterruptedException ...: " + e);
}
```

Durante i 10 secondi, anche se la CPU è libera, il thread rimane inattivo. Passati i dieci secondi lo scheduler lo reinserirà fra i thread Runnable.

Stati di un Thread (4)

- **Not Runnable State**
  - Passaggio da “not Runnable” a “Runnable” in quattro modi diversi che rispecchiano le modalità del passaggio inverso:
    - ***1) resume()*** [se era in suspend()]
    - ***2)*** trascorso tempo di **`sleep()`** o chiamato ***`interrupt()`***
    - ***3)*** **`notify()`** o ***`notifyAll()`*** **`->`** chiamati dall’oggetto che possiede il monitor
    - ***4)*** finisce l’operazione di I/O
  - L’invocazione di un metodo diverso non produce alcun effetto.
    - Esempio: se un thread è in uno stato di *sleep*, una chiamata di *resume()* non produrrà alcun effetto, perché il thread deve ritornare allo stato Runnable solo al verificarsi della condizione ***2).***

Stati di un Thread (5)

- **Dead State**
  - Vi sono due modi differenti per far terminare un thread:
    - fine del metodo *run()*
    - chiamata del metodo stop()

```
MyThreadClass myThread = new MyThreadClass();
myThread.start();
myThread.stop();
```

    - Il thread viene ucciso in maniera asincrona.
    - Se il thread stava eseguendo delle operazioni “delicate”, tale istruzione potrebbe lasciare il sistema in uno stato inconsistente.
    - È preferibile forzare l’arresto di un thread tramite un flag che faccia uscire il sistema dal metodo run().

Priorità e scheduling dei Thread

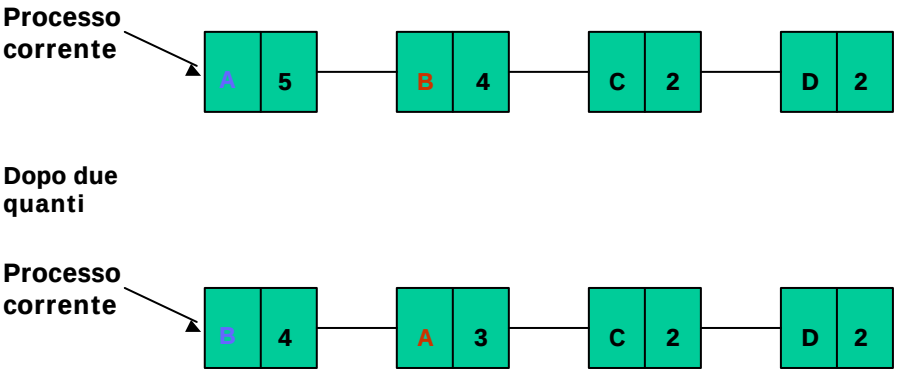
- Nei sistemi monoprocessore un solo thread alla volta può essere eseguito (necessario algoritmo di "scheduling" per ripartire equamente le risorse di calcolo).
- Java supporta l'algoritmo "***fixed priority scheduling***", basato sulle priorità di ogni thread.
- Priorità:
  - ogni thread eredita la priorità della classe che lo crea;
  - le priorità in Java sono interi compresi fra MIN_PRIORITY a MAX_PRIORITY;
  - è possibile cambiare una priorità mediante il metodo *setPriority(int)*.

Fixed Priority Scheduling

- E' basato sulle priorità di ogni thread.
- In ogni momento, se c'è più di un Runnable thread, il sistema sceglie più spesso quello a priorità più alta.
- Se ci sono più Runnable thread con la stessa priorità, il sistema ne sceglie uno operando in modalità "round-robin".
- Il thread scelto può continuare l'esecuzione fino a che:
  - un thread a più alta priorità diviene Runnable;
  - il thread invoca yield() o il metodo run() finisce;
  - in un sistema "time-slicing" il suo periodo di CPU è terminato;
  - passa da solo dallo stato "Runnable" a "not Runnable"
    - sleep() - wait() - condizione di I/O.

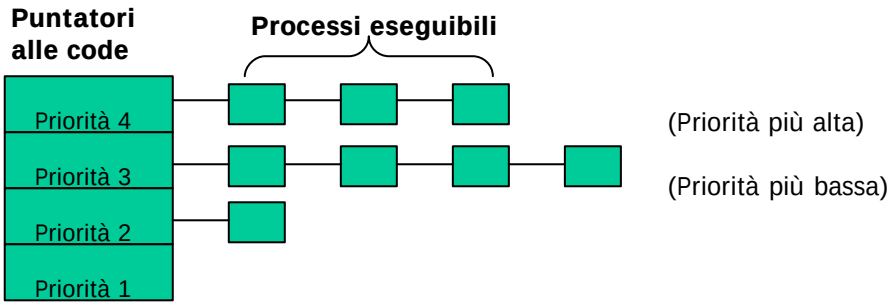
Schedulazione con Priorità (1)

Ad ogni processo viene assegnata una priorità e viene concessa l'esecuzione al processo eseguibile con priorità più alta



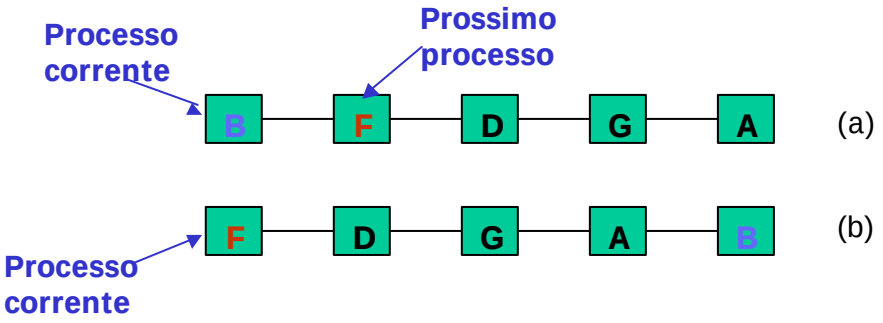
Schedulazione con Priorità (2)

In alcuni casi potrebbe essere utile raggruppare i processi in classi di priorità e usare lo scheduling a priorità fra le classi, ma round robin all'interno di ciascuna classe



Schedulazione Round Robin

Ad ogni processo si assegna un **quanto** di tempo durante il quale gli viene assegnata la CPU che rilascia automaticamente o al termine della sua azione o al termine del quanto.

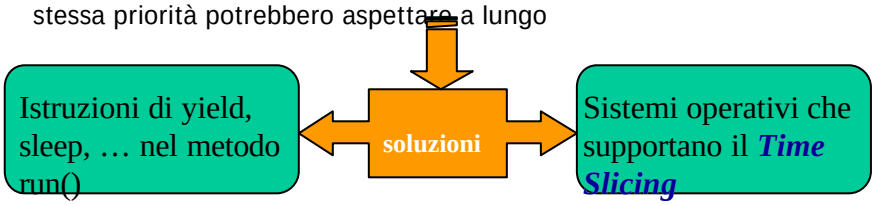


Selfish Threads

- Esempio: metodo run di un Thread

```
public int tick = 1;
public void run() {
        while (tick < 4000000) {
                tick++;
        }
}
```

Una volta in esecuzione, tale thread continua fino a che o finisce il ciclo o interviene qualche altro thread a priorità maggiore => i thread con la stessa priorità potrebbero aspettare a lungo



Selfish Thread: esempio

- Costruzione di una classe thread che inizia a contare da 1 fino a 40000000 ed ogni 500000 stampa il valore del contatore corrente.
- Il main lancia due SelfishThread con la stessa priorità.
- Classe SelfishThread

```
public class SelfishRunner extends Thread {
   private int tick = 1;    private int num;
   public SelfishRunner(int num) {   this.num = num;   }
   public void run() {
      while (tick < 40000000) {
         tick++;
         if ((tick % 500000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
         }
      }
   }
}
```


Esecuzione dell'esempio

Sistema NON Time-slicing

```
Thread #0, tick = 500000
Thread #0, tick = 1000000
...
Thread #0, tick = 40000000
Thread #1, tick = 500000
Thread #1, tick = 1000000
...
Thread #1, tick = 40000000
```

Il primo Thread che assume il controllo della CPU arriva fino in fondo al conteggio (cioè fino alla fine del suo metodo run()).

Sistema Time-slicing

```
Thread #0, tick = 500000
Thread #0, tick = 1000000
Thread #1, tick = 500000
Thread #0, tick = 1500000
Thread #1, tick = 1000000
...
```

La frequenza dei cambi dipende dalla larghezza dei singoli slot temporali in relazione alla potenza elaborativa del PC.

Modifica dell'esempio (1)

- Per garantire un funzionamento corretto del sistema anche su piattaforme che non supportano il time-slicing è possibile utilizzare il metodo yield().
- **Yield** fa rimanere il thread nello stato Runnable ma permette al sistema di cedere la CPU ad un altro thread Runnable con la stessa priorità.

```
public class SelfishRunner extends Thread {
   …
   public void run() {
      while (tick < 40000000) {
         tick++;
         if ((tick % 500000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
            yield();
         }
      ...
}
```


Modifica dell'esempio (2)

- Il sistema alternerà i thread nello stato Runnable con un algoritmo **round-robin**
- Nel caso di tre thread si avrà un output del tipo:

```
Thread #0, tick = 500000
Thread #1, tick = 500000
Thread #2, tick = 500000
Thread #0, tick = 1000000
Thread #1, tick = 1000000
Thread #2, tick = 1000000
Thread #0, tick = 1500000
Thread #1, tick = 1500000
Thread #2, tick = 1500000
```

La presenza dell'istruzione yield tende a rendere il risultato del programma **deterministico** indipendentemente dal sistema operativo (time-slicing o no).

Daemon Threads

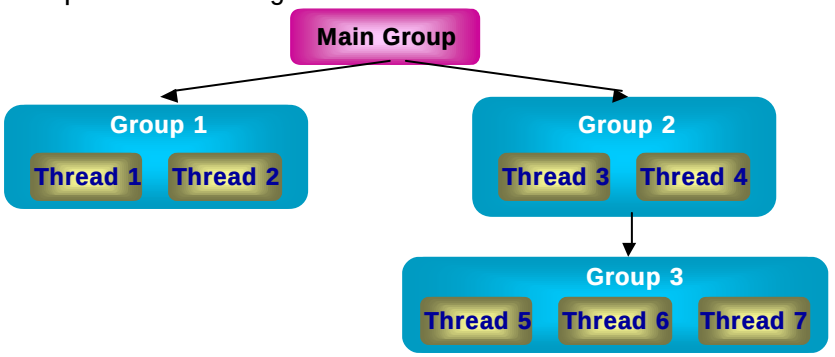
- **Daemon Thread**
  - Ogni thread può essere *daemon* tramite il metodo *setDaemon(true)*
  - Un daemon thread fornisce dei servizi agli altri thread presenti nello stesso processo:
    - Es: Il web browser HotJava usa dei daemon thread (ImageFetcher) per caricare le immagini e metterle a disposizione agli altri thread quando ne fanno richiesta.
  - Solitamente il metodo run() di un DaemonThread è un loop infinito che aspetta le richieste dagli altri thread.
  - Quando rimangono solo Daemon thread attivi, la JVM termina.
  - Il metodo *isDaemon()* serve per determinare se un thread è o meno un daemon thread.

Gruppi di Thread (1)

- **Thread Group**
  - Ogni thread è membro di un gruppo.
  - Ogni operazione effettuata su un gruppo si ripercuote su tutti i thread appartenenti al gruppo stesso (*start*, *resume*, *suspend*, ...).
  - E' necessario specificare il gruppo di appartenenza al momento della creazione di un Thread.
  - La classe *java.lang.ThreadGroup* fornisce tutti i metodi per la manipolazione dei gruppi.
  - Per un'applicazione Java, il gruppo di default è denominato "***main***" (per gli applet dipendente dalla JVM del browser).
  - Per creare un ThreadGroup:
    - *ThreadGroup myTG = new ThreadGroup("sono myTG");*
  - Costruttori della classe Thread con il parametro ThreadGroup
    - *Thread myT = new Thread(myTG, myRunnable, "sono myT");*

Gruppi di Thread (2)

- È possibile costruire dei gruppi derivanti da altri gruppi.
  - Costruttore: *ThreadGroup (ThreadGroup parent, String name).*
  - Si possono creare gerarchie di thread.



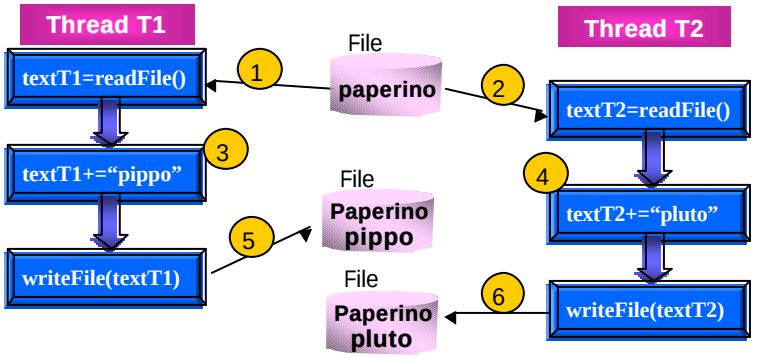
Ciascun thread può attraversare l'intera gerarchia senza avere l'handle dei vari thread a priori.

Sincronizzazione nei programmi multithreading

- **Sincronizzazione**
  - Quando più thread condividono dati e/o metodi, si necessita una sincronizzazione:
    - lettura/scrittura di un file
    - lettura/scrittura del valore di una variabile
- Un thread, nella sua esecuzione, deve tenere conto dello stato di altri thread con cui condivide risorse e metodi.
- La sincronizzazione dei thread prescinde dall'algoritmo di scheduling adottato.

Problematica di sincronizzazione

Esempio: accesso ad uno stesso file



- La stringa aggiunta dal Thread T1 viene persa.
  - Servono meccanismi di sincronizzazione fra thread e di lock di risorse condivise.

Sincronizzazione nei linguaggi multithread

- Nei linguaggi single-threaded è possibile gestire più task ricorrendo agli interrupt o al polling.
  - E' compito dell'utente gestire la sincronizzazione fra i diversi task (in C i processi, generati tramite un'operazione di ***fork***), e questo rende il meccanismo complesso anche per situazioni semplici.
- Linguaggi multithreaded: notevole semplificazione.
  - Dispongono di un insieme di primitive che consentono di gestire le caratteristiche base della sincronizzazione.

Paradigma Producer - Consumer

- Spiegazione del multithreading Java tramite l'esempio del producer/consumer.
- Il producer genera dei numeri in sequenza da 0 fino a 9 e li memorizza in una variabile.
- Il consumer accede alla variabile 10 volte per prelevarne il valore.

Il Consumer deve prelevare una sola volta i dati immessi dal Producer.

Shared Variable (oggetto condiviso)

- È costituita da una classe che mette a disposizione i metodi di *get* e di *put*.

```
public class SharedVariable {
   private int contents;
   public int get() {
      return contents;
   }
   public void put (int value) {
      contents = value;
   }
}
```

Producer

- Ogni volta che immette un numero, fa una pausa di una durata random fra 0 e 100 millisecondi.

```
public class Producer extends Thread {
 private SharedVariable sharedVariable;
 private int number;
 public Producer(SharedVariable sv, int number) {
  sharedVariable = sv;
  this.number = number;
 }
 public void run() {
  for (int i = 0; i < 10; i++) {
   sharedVariable.put(i);
   System.out.println("Producer #" + this.number + " put: " + i);
   try { sleep((int)(Math.random() * 100)); } catch (InterruptedException e) {} } } }
```

Producer

Consumer

- Ogni volta che preleva un numero fa una pausa di una durata random fra 0 e 100 millisecondi.

```
public class Consumer extends Thread {
    private SharedVariable sharedVariable;
    private int number;

    public Consumer(SharedVariable sv, int number) {
        sharedVariable = sv;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = sharedVariable.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
            try { sleep((int)(Math.random() * 100)); } catch (InterruptedException e) { }
        }
    }  }
```

Consumer

Risultato senza sincronizzazione

- Programma main: istanzia la variable condivisa (sharedVariable), il Producer ed il Consumer

```
public class ProducerConsumerTest {
   public static void main(String[] args) {
      SharedVariable sv = new SharedVariable();
      Producer p1 = new Producer(sv, 1);
      Consumer c1 = new Consumer(sv, 1);
      p1.start();
      c1.start();
   }
}
```

1) Caso: Producer più veloce del Consumer	2) Caso: Consumer più veloce del Producer
Consumer #1 got: 3	Producer #1 put: 4
Producer #1 put: 4	Consumer #1 got: 4
Producer #1 put: 5	Consumer #1 got: 4
Consumer #1 got: 5	Producer #1 put: 5

Problemi nel paradigma Producer/Consumer

- **1)** Quando il Producer invoca il metodo put della SharedVariable, al Consumer deve essere bloccato l'accesso al metodo get e viceversa.
  - Soluzione: Java Monitors, che prevengono l'accesso simultaneo di due o più thread a sezioni di codice in una classe Java.
- **2)** Coordinamento nell'accesso alla variabile condivisa: il Producer ed il Consumer si devono alternare nell'accesso.
  - Soluzione: Utilizzo dei metodi di *wait()* e *notify().*

Java Monitor

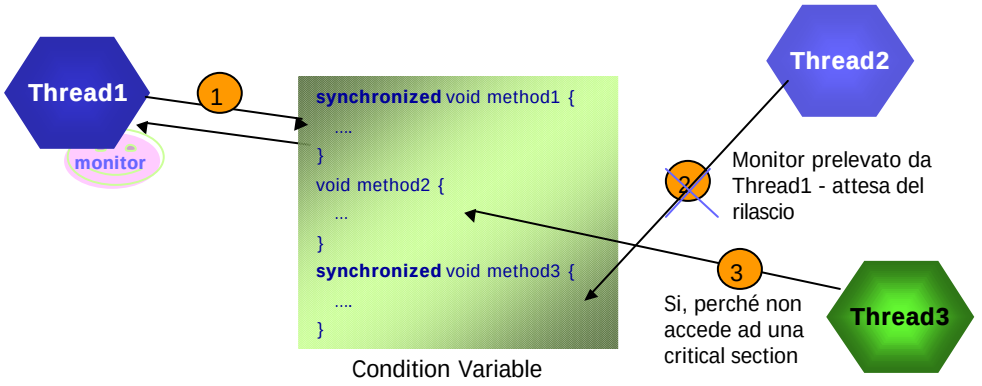
- Ogni qualvolta si hanno più thread che tentano di accedere ad una stessa risorsa si verificano "*race conditions*".
- Java utilizza il paradigma dei ***monitor***.
  - Una risorsa accedibile da più thread è detta "*condition variable*".
    - Es. La variabile *contents* nella classe *SharedVariable.*
  - I frammenti di codice che accedono a "*condition variable*" sono detti "*critical sections*".
    - Es. I metodi di *put* e *get* nella classe *SharedVariable.*
  - In Java si devono marcare le "*critical sections*" con la parola chiave "***synchronized***".
    - Es. I metodi put e get dovranno avere una signature del tipo:
      - public **synchronized** void put(int value);
      - public **synchronized** int get ().

Funzionamento dei Monitor (1)

- *In Java, un monitor è associato ad una singola istanza di una classe che possiede almeno una sezione critica.*
  - Ogni volta che un thread accede ad una sezione critica di un oggetto, preleva il monitor per quell'oggetto. Questo impedisce agli altri thread di accedere alle sezione critiche di quell'oggetto.
  - Il monitor è unico per ogni istanza della classe contenente le risorse condivise.
  - Per maggiore chiarezza del codice, è bene utilizzare la parola chiave synchronized solamente a livello dei metodi (e non per blocchi isolati di codice dentro i metodi).
- Quando un thread esce da una sezione critica, rilascia il monitor dell'oggetto, le cui sezioni critiche potranno essere accedute da altri thread che erano in attesa.

Funzionamento dei Monitor (2)

- Quando un thread ha acquisito il monitor per un'oggetto, gli altri thread potranno solamente accedere ai metodi **non** synchronized dell'oggetto.



Se una sottoclasse sovrascrive un metodo synchronized, il metodo nella sottoclasse può essere o meno synchronized.

Coordinamento delle azioni

- Nell'esempio del Producer/Consumer, l'inserimento della parola chiave synchronized evita l'accesso contemporaneo.
- Manca il coordinamento delle azioni: l'accesso deve essere alternato fra il producer ed il consumer.
- La soluzione dipende dal caso specifico e va risolta a livello di programma.
- È possibile utilizzare i metodi *wait()*, *notify()* e *notifyAll()* per aiutare il programmatore a risolvere questi problemi.
  - wait() pone il thread che aveva acquisito il monitor in attesa (passaggio da Runnable a not Runnable) e rilascia il monitor;
  - notify() effettua l'operazione contraria.
  - ***wait e notify() e notifyAll() sono definiti nella classe Object.***

Metodi notify e wait

- notify (notifyAll() e notify())
  - Notificano ai thread in attesa di prelevare il monitor che il thread corrente ha finito il suo accesso alla sezione critica.
  - Il sistema Java runtime sceglie un thread fra quelli in attesa e gli abilita l'accesso alla risorsa condivisa.
    - Con notifyAll() il sistema runtime di Java risveglia (passaggio da not Runnable a Runnable) tutti i thread in attesa e ne schedula uno.
    - Con notify() il sistema runtime di Java sceglie un thread a caso in attesa e lo fa passare nello stato Runnable, facendogli acquisire il monitor.
- Metodi di wait
  - Pongono un thread in attesa fino a che una condizione non cambia.
  - Un thread nella posizione di wait può essere riattivato se:
    - un altro thread che possedeva il monitor invoca notify;
    - è scaduto il tempo di attesa specificato nell'istruzione di wait.

Producer/Consumer: soluzione

```
class SharedVariable {
   private int contents;
   private boolean available = false;

   public synchronized int get() {
      while (available == false) {
        try {
           wait();
        } catch (InterruptedException e) { }
      }
      available = false;
      notifyAll();
      return contents;
   }
   ...
}
```

- La variabile *available* indica se un nuovo valore è disponibile o meno.
- Se *available* è false, il Consumer viene messo nello stato not Runnable tramite l'istruzione wait().
- Quando il Producer avrà immesso un nuovo valore, setterà available a true e invocherà notifyAll() => il Consumer riacquisisce il monitor e può prelevare il valore della variabile.

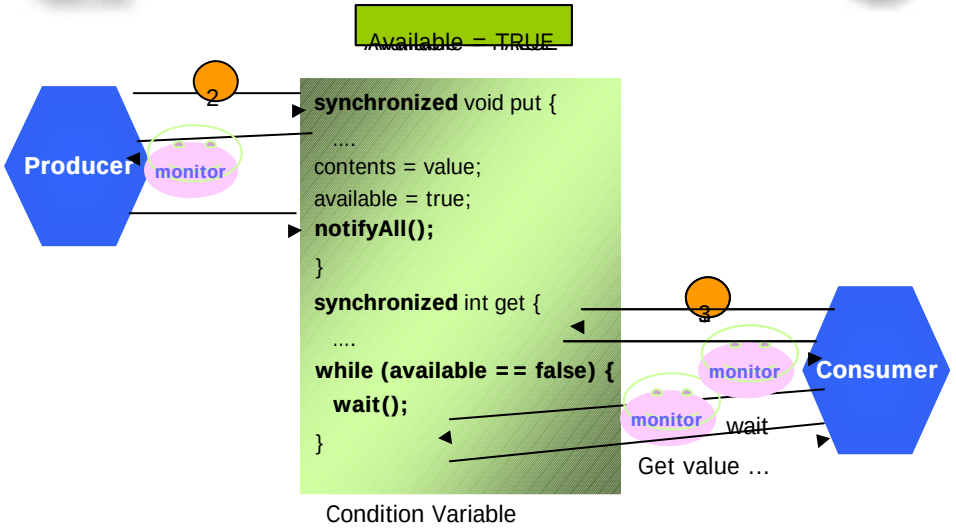
Producer/Consumer: soluzione

```
class SharedVariable {
   private int contents;
   private boolean available = false;
   ...

   public synchronized void put(int value) {
      while (available == true) {
         try {
            wait();
         } catch (InterruptedException e {}
      }
      contents = value;
      available = true;
      notifyAll();
   }
}
```

- All'inizio available è false, cosicché il Producer può inserire il primo valore (il Consumer viene bloccato).
- Se *available* è true, il Producer viene messo nello stato not Runnable tramite l'istruzione wait().
- Quando il Consumer avrà prelevato il valore corrente, setterà available a false e invocherà notifyAll() => il Producer riacquisisce il monitor e può immettere un nuovo valore.

Dinamica dell'accesso



Uso di wait e notify

- È indispensabile per la sincronizzazione.
- Supponendo di avere un sistema operativo che non supporta il time-slicing, l'assenza delle istruzioni di wait e notify provocherebbe un blocco del programma.
  - Es. Supponendo che il producer abbia messo un primo valore (available = true) e ripreso di nuovo il monitor, si avrebbe un blocco sull'istruzione

```
while (available == true) {
}
```

  - Lo stesso dicasi se inizia il Consumer.

1.2: deprecation

- A partire dalla versione 1.2 del JDK i metodi stop(), suspend(), resume() e destroy() sono stati deprecati.
- In tal modo si riducono i rischi di deadlock e di situazioni di scarsa robustezza.
- Il metodo **stop()** rilascia tutti i lock acquisiti dal thread col rischio che i relativi oggetti siano in uno stato inconsistente.
  - Per far terminare un thread è opportuno utilizzare una flag, sulla quale il metodo run() del thread fa polling per decidere se terminare la propria esecuzione (e quindi far terminare il thread).
  - Se il thread è sospeso su sleep o su wait si usa interrupt().
  - Se il thread è in attesa di input, gli si chiude lo stream.