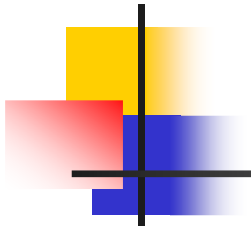
Linguaggi

Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2009-2010

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

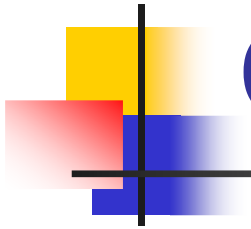
- lezione 16 -

Thread in Java



Cos'è un thread?

- Un programma sequenziale ha:
  - un inizio
  - una sequenza di esecuzione
  - Una fine.
- Ad ogni istante durante la sua esecuzione c'è un singolo punto in esecuzione
- **Un thread è simile a un programma sequenziale:**
- Un singolo thread ha:
  - una sequenza di esecuzione
  - Una fine.
  - Ad ogni istante durante l'esecuzione del thread c'è un singolo punto in esecuzione
- Comunque, non è esso stesso un programma; non può essere eseguito in modo indipendente; viene invece eseguito all'interno di un programma.



Cos'è un thread?

- Un thread è un singolo flusso sequenziale di istruzioni all'interno di un programma.
- Si possono utilizzare più threads all'interno di un singolo programma; essi eseguiranno allo stesso tempo ma eseguono tasks differenti
- Alcuni si riferiscono ai threads come "processo leggero", eseguito all'interno di un contesto di un programma completo; utilizza le risorse allocate per il programma e l'ambiente del programma, tuttavia…
- Come un flusso di controllo sequenziale, un thread deve ritagliarsi alcune delle sue risorse all'interno del programma in esecuzione. Esso deve quindi avere il suo stack di esecuzione e un program counter.

Cos'è un thread?

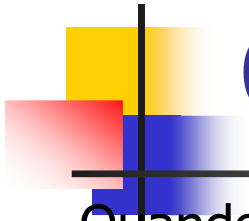
- L'uso dei thread comporta **svantaggi**:
  - Difficili da usare
  - Rendono i programmi difficili da debuggare
  - Rischio di deadlock
  - Bisogna avere cura che tutti I threads creati non invochino componenti di Swing (I componenti di Swing non sono "thread safe")
- L'uso dei thread comporta però anche **vantaggi**:
  - Miglioramento delle performance dei programmi, specie in applicazioni grafiche
  - Semplificazione del codice: per eventi temporizzati, piuttosto che eseguire cicli o polling è possibile utilizzare una classe Timer che notifica una certa quantità di tempo trascorsa
- **Alcuni tipici usi dei threads sono**:
  - Spostare un time-consuming initialization task all'esterno del thread, così che la GUI diventi più veloce.
  - Spostare un time-consuming task all'esterno del event-dispatching thread, cosi che la GUI rimane interattiva
  - Per eseguire operazioni ripetitive usualmente con un predeterminato periodo di tempo fra le operazioni.
  - Aspettare messaggi da altri programmi

Creazione di Thread

- Un programma diventa multithreaded se il singolo thread costruisce e avvia un secondo thread di esecuzione; Ogni thread può attivare (start) un qualsiasi numero di Threads.
- In Java i threads sono oggetti, e vi sono **2 modi per creare un thread**:
  - Implementare l'interfaccia Runnable (e passare questo nel costruttore di un Thread),
  - Estendere java.lang.Thread (che implementa Runnable)
- Solitamente, estendere la java.lang.Thread può essere scomodo perché:
  - non è possibile ereditare da altre classi (anche qualora ciò fosse reso necessario dal progetto in corso di realizzazione)
  - Si ereditano tutti i metodi di Thread, cosa non sempre necessaria
- Spesso allora si implementa Runnable, oggetto che può essere eseguito all'interno di un suo thread semplicemente fornendolo come argomento ad un costruttore della classe Thread:

  Class pippo implements Runnable {
  Public pippo() { New Thread(this).start(); }
  }

  - L'assenza di un riferimento al Thread non implica che tale riferimento non esista; un thread, quando creato, memorizza un riferimento a se stesso all'interno del ThreadGroup

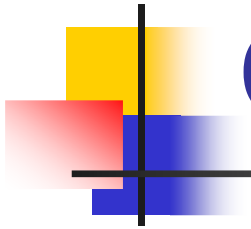


Creazione di Thread

- Quando viene implementata l'interfaccia Runnable, si deve fornire il metodo run() (l'unico) per indicare il lavoro da svolgere; tale metodo solitamente ha natura privata, ma essendo parte di un'interfaccia è di fatto pubblico, e questo potrebbe non essere accettabile (chiunque potrebbe invocarlo); vi sono due soluzioni:
  - Invocare Thread.currentThread per stabilire l'identità del Thread che invoca run() e confrontarla con quella del thread con cui si dovrebbe realmente lavorare
  - La seconda soluzione consiste nel definire una classe anonima dentro quella data che contenga un oggetto Runnable:

```
Class pippo {
    public pippo() {
        Runnable pluto=new Runnable () {
            Public void run() {    for (;;) <istruzioni>    }
        }
        new Thread(pluto).start();    } // END Costruttore
}
```

  - l'uso di oggetti Runnable permette maggiore flessibilità, perché ognuno di essi costituisce una unità lavorativa indipendente, anche rispetto al resto del programma



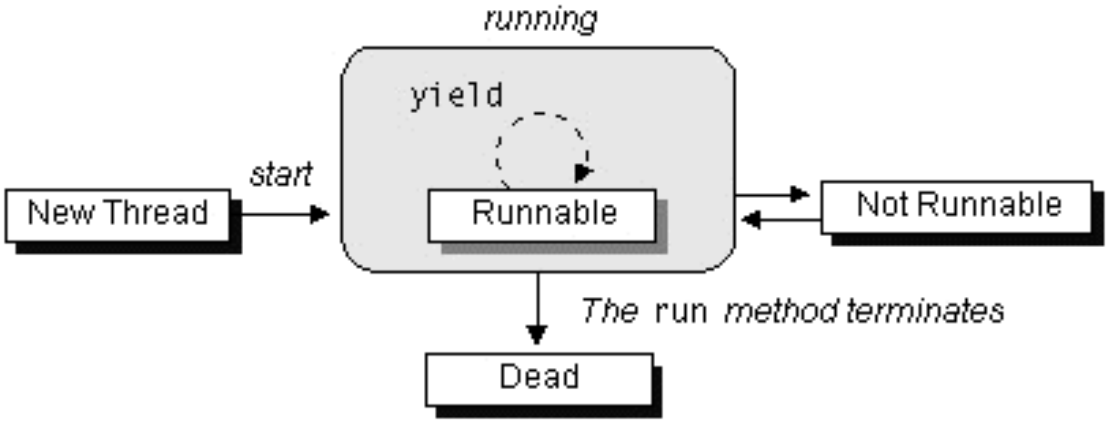
Creazione di Thread

- Il programa non invoca run() direttamente; se si vuole avviare un Thread t, si può chiamare t.start() e la JVM si occuperà di chiamare t.run().
- start() porta a conoscenza della JVM che il codice di questo metodo ha la necessità di essere eseguito su un proprio thread:

```
Thread t = new Thread(this);
t.run();    // ERRATO
t.start();  //OK
```

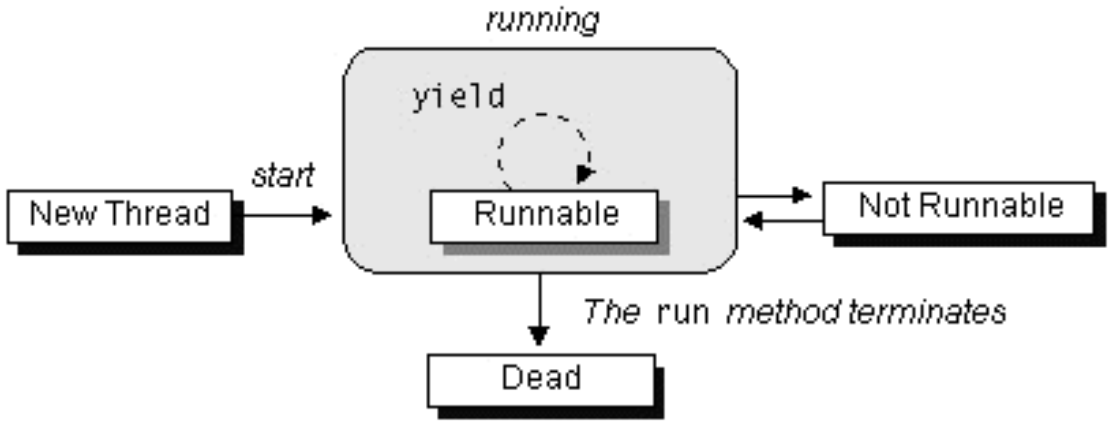
Stati di un Thread

- Quando un thread è creato (New Thread) nel suo stato iniziale è un empty Thread object; nessuna risorsa di sistema è allocata per esso. Quando un thread è in questo stato, su di esso può essere invocato il metodo start; se start è invocato su un oggetto thread che non si trova in tale stato iniziale si causa l'eccezione IllegalThreadStateException.
- Il metodo start crea le risorse di sistema necessarie per eseguire il thread, schedula il thread per eseguirlo, e chiama il metodo run della classe thread; dopo il return di start il thread è **Runnable**



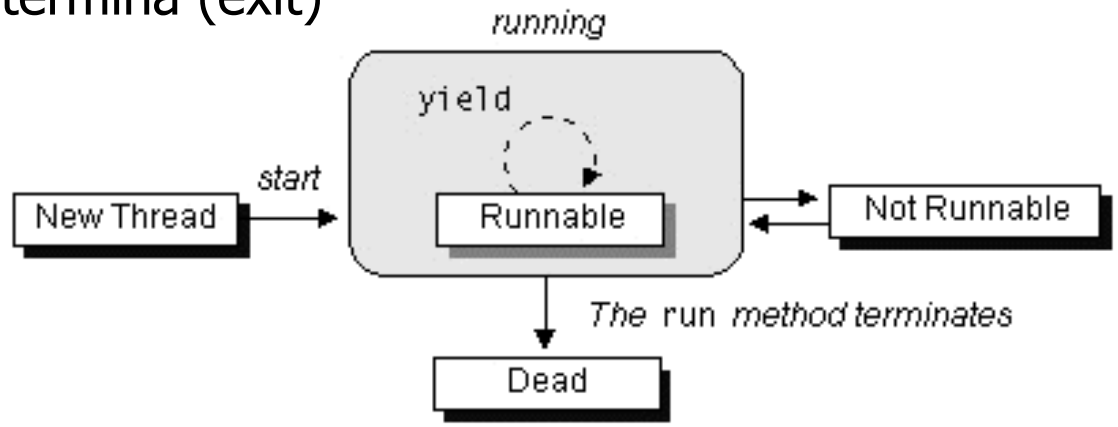
Stati di un Thread

- Un thread che inizializzato con start() è nello stato **Runnable**. Molti computer hanno un singolo processore, quindi è impossibile eseguire tutti I "running" threads allo stesso tempo. Il Java runtime system deve implementare uno schema di scheduling scheme che permette la condivisione del processore fra tutti i "running" threads, così, ad un dato istante un solo "running" thread è realmente in esecuzione, gli altri sono in waiting per il loro turno di CPU.
- Un thread diventa **Not Runnable** (o blocked) quando:
  - Il thread chiama il metodo wait per aspettare che una specifica condizione sia soddisfatta
  - Il thread è bloccato su un I/O.



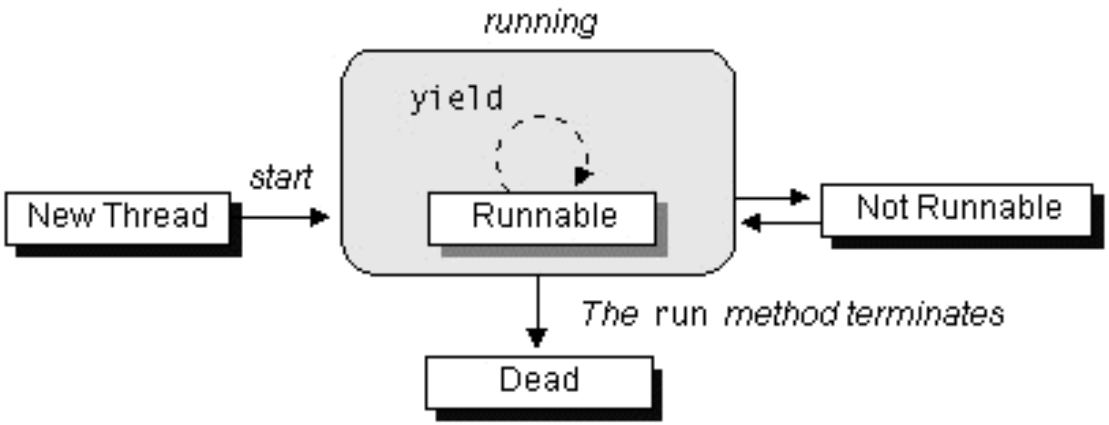
Stati di un Thread

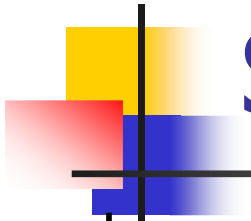
- Per ogni entrata in un Not Runnable state, c'è una specifica e distinta azione che permette il ritorno nello stato di Runnable:
  - Se un thread è stato posto in wait, il numero specifico di millisecondi (che è il parametro del metodo wait stesso) deve trascorrere.
  - Se un thread è in attesa per una condizione, allora un altro oggetto deve notificare il thread in attesa di un cambio nelle condizioni chiamando notify o notifyAll
  - Se un thread è blocked su I/O, allora l'I/O deve essere completato
- Un programma non termina un thread attraverso la chiamata di un metodo; piuttosto, un thread termina naturalmente (**dead**) quando il metodo run termina (exit)



Stati di un Thread

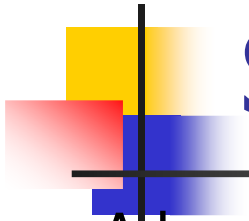
- La classe Thread ha un metodo chiamato isAlive().
- Il metodo isAlive ritorna true se il thread è stato started e non stopped.
- Se il metodo isAlive ritorna false, il thread è un New Thread o è morto.
- Se il metodo isAlive ritorna true, esso è Runnable o Not Runnable.
- Non è possibile distinguere un New Thread da un Dead thread, nè è possibile distinguere un Runnable thread da un Not Runnable thread.





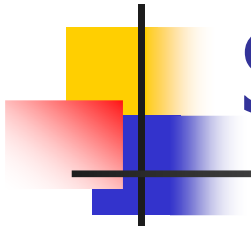
Schedulazione di Thread

- La concorrenza dell'esecuzione dei thread è vera, di norma, solo concettualmente; la maggior parte delle configurazioni di computer prevedono singole CPU, così i threads realmente sono eseguiti uno alla volta in modo da fornire una illusione di concorrenza.
- L'esecuzione di thread multipli su una singola CPU, in qualche ordine, è detto scheduling.
- Il Java runtime supporta un semplice algoritmo di scheduling deterministico conosciuto come fixed priority scheduling, che schedula i threads sulla base delle loro priorità relative agli altri runnable threads
- Quando un thread viene creato esso eredita la sua priorita dal thread che lo ha creato
- È possibile modificare la priorità di un thread in qualsiasi momento dopo la sua creazione utilizzando il metodo setPriority.
- Le priorità dei Thread sono interi compresi fra MIN_PRIORITY e MAX_PRIORITY (constanti definite nelle classe Thread); il valore intero più grande corrisponde alla più alta priorità



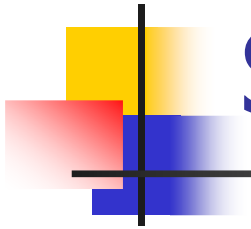
Schedulazione di Thread

- Ad un dato istante, quando più thread sono pronti per essere eseguiti, il runtime system sceglie il thread runnable con la più alta priorità per l'esecuzione.
- Solo quando il thread finisce o diventa not runnable per qualche ragione il thread di priorita minore viene eseguito
- Se due threads della stessa priorità sono in attesa per la CPU, lo scheduler ne sceglie uno di loro seguedo un modello round-robin.
- Il thread scelto sarà eseguito fino a quando:
  - un thread con più alta priorità non diventa runnable.
  - il suo metodo run esiste.
  - su un sistema con time-slicing, il suo slice è terminato
- Così i thread hanno una possibilità di essere eseguiti
- Lo scheduler del Java runtime system è preemptive: se in qualsiasi momento un thread con una più alta priorità di tutti gli altri runnable threads diventa runnable, allora il runtime system sceglie il nuovo thread di più alta priorità per l'esecuzione
- Ad ogni istante il thread di massima priorità dovrebbe essere in esecuzione. Comunque, ciò non è garantito.



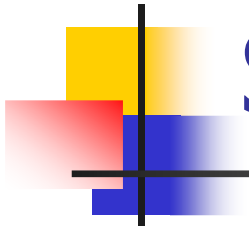
Schedulazione di Thread

- Il Java runtime non esegue la preemption di un thread in esecuzione della stessa priorità; in altre parole non utilizza un time-slice.
- Comunque, il sistema di implementazione dei thread sottostante la classe Java Thread può supportare il time-slicing.
- Inoltre, un dato thread può in un qualsiasi momento cedere i suoi diritti di esecuzione invocando il metodo **yield**; la cessione vale solo per thread con la stessa priorità
- Thread possiede anche metodi deprecated
  - **Stop**, ma è preferibile terminare il thread quando termina il metodo run()
  - **Suspend** e **resume**, che possono provocare deadlock



Sincronizzazione di Thread

- Se i Thread sono asincroni, ogni thread contiene tutti i dati e i metodi necessari per l'esecuzione; i thread pertanto evolvono indipendentemente e non vi è nessun problema per le velocita relative dei threads.
- Cosa accade invece quando esistono dati condivisi tra più threads? Esempio tipico, problema produttore/consumatore:
- Una applicazione Java ha un thread (produttore) che scrive dati su un file mentre un secondo thread (consumatore) legge i dati dallo stesso file.
- Digitando caratteri sulla tastiera il thread produttore pone l'evento key in una coda di eventi e il thread consumatore consumer legge gli eventi dalla stessa coda. Ambedue questi esempi coinvolgono thread concorrenti che condividono una risorsa comune (File e coda di eventi), pertanto i thread devono essere sincronizzati.



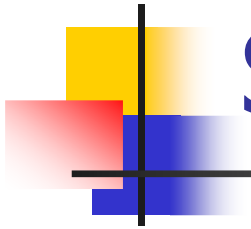
Sincronizzazione di Thread

Risorsa condivisa:

```
public class Box {
        private int contents;
        public void put(int x) {contents = x;}
        public int get() {return contentsi;}
}
```

Produttore:

```
public class Producer extends Thread{
private Box pozzo;
private int number;
public Producer(Box l, int number) {       pozzo = l;this.number = number;}
public void run (){
        for (int i=0; i<10; i++){
        pozzo.put(i);
        System.out.println("Produzione del numero "+this.number + "put" + i);
        try {sleep((int)(Math.random()*10));}
        catch (InterruptedException e){};
        }  // END FOR
        } // END RUN          }
```

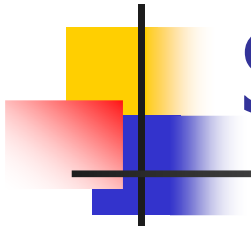


Sincronizzazione di Thread

- **Consumatore**:

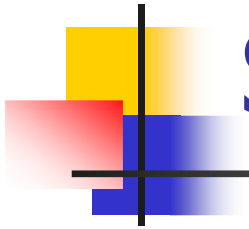
```
public class Consumer extends Thread
{
private Box pozzo;
private int number;
public Consumer(Box l, int number){pozzo = l;this.number = number;}
public void run ()
{ int value = 0;
for (int i=0; i<10; i++)
{ value=pozzo.get();
System.out.println("Consumo del num " +this.number + "get" + value);
try
{sleep((int)(Math.random()*1000));}
catch (InterruptedException e){};}}}
```

- Producer and Consumer in questo esempio condividono i dati di un oggetto comune. Il problema è che il Producer potrebbe essere piu lento o più rapido del Consumer, pertanto è necessaria la sincronizzazione fra i due threads.



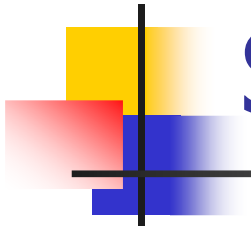
Sincronizzazione di Thread

- L'attività di Producer e Consumer può essere sincronizzata con due strumenti:
  - I due thread non devono accedere simultanemaente all'oggetto Box, quindi serve un lock
  - I due thread devono coordinarsi. Il Producer deve indicare al Consumer che il valore è pronto e il Consumer deve avere un modo per notificare che il valore è stato prelevato. La classe Thread ha una collezione di metodi (wait, notify, e notifyAll) che consentono ai threads di aspettare una condizione e notificare ad altri threads che la condizione è cambiata.
  - Il lock serve per regolare l'accesso alla risorsa condivisa; il lock tuttavia non garantisce nessun ordine nelle operazioni, ma solo il mutex, per questo serve anche il coordinamento
- Il segmento di codice entro un programma che accede ad un oggetto da diversi threads concorrenti è chiamato "regione critica".



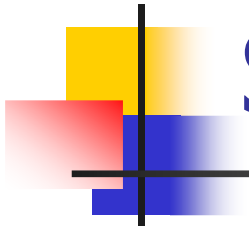
Sincronizzazione di Thread

- Le regioni critiche possono essere un blocco di istruzioni o un metodo e sono identificate dalla keyword synchronized:
  - posta prima delle istruzioni in accordo alla sintassi synchronized (expr) { <istruzioni> }, dove expr è un riferimento all'oggetto che sarà lockato
  - oppure nella dichiarazione del metodo (public int synchronized metodo…)
  - La prima scelta potrebbe essere la migliore, perché raffina il livello di granularità, riducendo l'acquisizione del lock solo alle istruzioni strettamente necessarie, ed inoltre la prima scelta permette di effettuare la sincronizzazione anche su oggetti diversi da this, specificandoli nella expr
  - Il metodo, d'altra parte, potrebbe essere una buona scelta progettuale per localizzarvi tutto il codice sincronizzato



Sincronizzazione di Thread

- Java associa un lock ad ogni oggetto che ha del codice synchronized.
- I lock sono acquisiti per thread, quindi se un metodo sincronizzato chiama un altro metodo sincronizzato, non si cerca di ottenere altri lock; in tal modo, è possibile effettuare chiamate anche ricorsive, e/o di metodi ereditati senza incorrere in blocchi o deadlock perché il lock acquisito resta sempre uno solo
- In ogni momento è possibile sapere se un oggetto è lockato da un thread invocando dal codice di tale thread il metodo statico holdslock della classe thread, che come parametro accetta l'oggetto e restituisce un booleano che indica l'acquisizione del lock
- Quando una classe estesa opera l'overriding di un metodo sincronizzato, può anche non imporre la sincronizzazione nel metodo sovrascrivente; esso risulterà però automaticamente sincronizzato se invoca super



Sincronizzazione di Thread

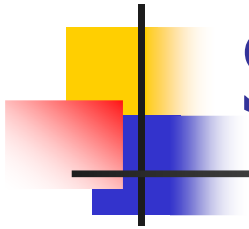
```
public class Box {
private int contents;
private boolean available = false;
public synchronized int get() { ... }
public synchronized void put(int value) { ... }
}
```

- Consumer non dovrebbe accedere a Box quando Producer sta cambiando il valore in Box, e Producer non deve eseguire modifiche quando Consumer sta recuperando il valore.
- Ogni volta che si entra in un metodo sincronizzato, il thread chiamante acquisisce il lock sull'oggetto; altri thread non possono acquisire il lock fintantochè non si ottiene il rilascio dell'oggetto da parte del thread

Sincronizzazione di Thread

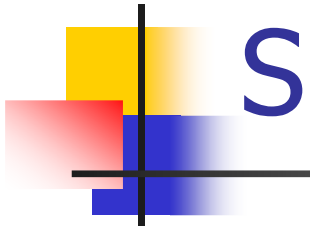
public synchronized void put(int value) { // Box locked dal Producer ..
// Box unlocked dal Producer
}
public synchronized int get() { // Box locked dal Consumer ...
// Box unlocked dal Consumer
}

- L'acquisizione e il rilascio di un lock è fatto automaticamente dal runtime system, non vi sono più race conditions e si mantiene la data integrity.
- Progettualmente, si può inserire il synchronized sulle classi produttore e consumatore che usano un oggetto condiviso (**approccio lato client**), o rendere sincronizzati i metodi di accesso all'oggetto condiviso direttamente nella sua classe (**approccio lato server**)
- L'approccio dal lato server è più sicuro e non fa nessuna ipotesi sul buon comportamento dei client, ma non è detto che chi offre la classe la esporti sincronizzata; i questi casi, si potrebbe estenderla ed operare l'overriding dei suoi metodi per renderli sincronizzati, invocando poi essi i corrispondenti super al loro interno
- L'approccio lato client diventa l'unica soluzione possibile quando si ha a che fare con più oggetti o più chiamate di metodi



Sincronizzazione di Thread

- Utilizzo di notify e wait:
  public synchronized void put(int value) {
  if (available == false) { available = true; contents = value;}}
  public synchronized int get() {
  if (available == false) {available = true; return value;} }
- Il codice riportato non funziona, infatti quando Producer non ha posto qualcosa in Box e available non è vero, get non fa nulla; allo stesso modo, se Producer chiama put prima che Consumer abbia prelevato il valore, put non fa nulla.
- In realtà Consumer deve attendere fino a che Producer mette qualcosa in Box e il Producer deve notificare al Consumer che lo ha fatto, ed allo stesso modo, Producer deve aspettare fino a quando Consumer non ha prelevato il valore (e notifica al Producer di averlo fatto) prima di riporre un altro valore in Box
- Occorrono metodi di attesa e notifica



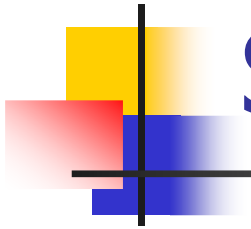
Sincronizzazione di Thread

```
public synchronized int get() {
    while (available == false) {
        try {
            // wait for Producer to put value
            wait();
        } catch (InterruptedException e) {
        }
    }
    available = false;
    // notify Producer that value has been retrieved
    notifyAll();
    return contents;
}
```

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    // notify Consumer that value has been set
    notifyAll();
}
```

...ed infine...

```
public class Test{
public static
void main(String s[]) {
Box b= new Box();
Producer p =
        new Producer(b,1);
Consumer c =
        new Consumer(b,1);
p.start();
c.start();}}
```

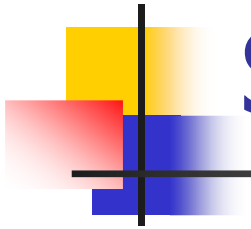


Sincronizzazione di Thread

- Un thread in attesa dovrebbe quindi implementare un pattern del tipo:

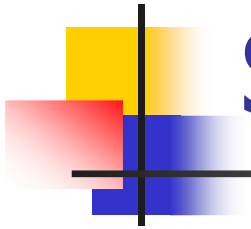
```
synchronized void metodo() {
        while (!condizione) wait();
        <istruzioni> da eseguire quando condizione è vera
}
```

- nel pattern descritto:
  - tutto ciò che viene eseguito è all'interno di codice sincronizzato; non è quindi possibile ad esempio che la condizione cambi subito dopo averla testata nel while, perché esiste il lock
  - nel momento in cui si invoca wait(), il thread viene sospeso e il lock viene rilasciato; le due azioni sono un'unica azione atomica
- in attesa sullo stesso oggetto possono trovarsi più thread
- la wait provoca l'attesa del thread che la invoca fino a che:
  - viene inviato un notify() o notifyall() da altri thread
  - scade il timeout (esiste il metodo wait(long timeout))
  - viene invocato il metodo interrupt del thread
- solitamente viene utilizzata solo la prima delle condizioni



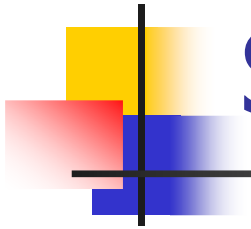
Sincronizzazione di Thread

- Un thread di notifica dovrebbe implementare un pattern del tipo:
  synchronized void metodo() {
  <istruzioni> che modificano il valore della condizione
  notify(); o notifyall();
  }
- notify effettua una notifica (risveglio) al più su uno dei thread in attesa della modifica della condizione; non è tuttavia possibile scegliere quale thread risvegliare, quindi notify andrebbe usato solo quando i thread sono in attesa della stessa identica condizione, ovvero quando è ininfluente la scelta del thread che verrà risvegliato; in caso contrario andrebbe usato notifyall
- In merito al **deadlock**, la responsabilità completa è del programmatore; solitamente, per prevenirlo si effettua l'ordinamento delle risorse, grazie a cui viene imposto che i lock vengano acquisiti rispettando l'ordine delle risorse; questo impedisce che due thread possano mantenere un lock ciascuno e contemporaneamente attendere anche quello dell'altro, e di fatto previene il deadlock



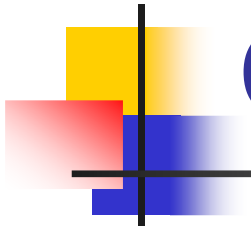
Sincronizzazione di Thread

- Ancora su notify vs notifyall:
- notifyAll() causes all the waiting threads (on the same lock) to wake up and start running, as opposed to notify(), which wakes up only one thread
- There is a difference between being blocked in the wait() state, and blocked while trying to reacquire the lock.
- It is true that only one thread can acquire the lock, hence, all the other threads (that wakeup with notifyAll()) will not run immediately -- but they will eventually wakeup as each in turn acquire and eventually free the lock.
- With notify(), only one thread will wakeup to acquire the lock, and eventually free the lock. The other threads will not unblock, even when the lock is freed, because they are still in the wait() state.



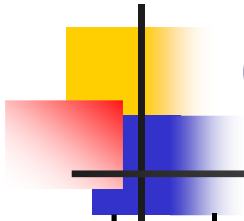
Sincronizzazione di Thread

- Ancora su notify vs notifyall:
- you would use notify() rather than notifyAll() only if you know that **exactly one** other thread is waiting for the lock.
- It's inefficient to wake 1000 threads and have 999 of them block while one executes. Better to just wake one at a time, usually. Though, if the monitor object *must* be made public, exposed to unknown code, unknown threads, then you probably need notifyAll(). Because if you don't control all the code that could possibly wait() on a given monitor, you can't guarantee that notify() will get called again.
- However, if you can't control all code that accesses a given monitor, you're usually vulnerable to other problems too, like deadlock, so maybe it is preferrable synchronizing on private monitors for this reason.
- Whether you call notify() or notifyAll() is dependant on the state that you just changed. If you changed the state, in a manner that can satisfy one waiting thread, then you call notify. If you change the state that can satisfy multiple waiting threads -- like releasing a group of semaphores, or shutting down the system (special case) -- then you call notifyAll().



Gruppi di Thread

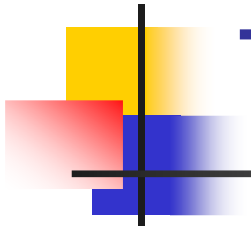
- Ogni thread Java è un membro di un thread group.
- Il Thread group fornisce un meccanismo per collezionare più thread in un singolo oggetto e manipolare questi threads tutti insieme piuttosto che individualmente. Per esempio, è possibile invocare lo start su tutti i threads all'interno di un group con la chiamata di un singolo metodo
- I Java thread groups sono implementati dalla classe java.lang.ThreadGroup.
- Il runtime system pone un thread in un thread group durante la sua costruzione.
- Quando viene creato un thread, si può permettere al runtime system di porlo in un default group o si può esplicitamente selezionare il gruppo desiderato
- Il thread è un membro permanente del gruppo in cui è inserito all'atto della creazione; non è quindi possibile spostare un thread in un nuovo gruppo dopo la sua creazione.



Gruppi di Thread

- La classe Thread ha tre costruttori che permettono di settare un nuovo gruppo di thread:
  public Thread(ThreadGroup group, Runnable runnable)
  public Thread(ThreadGroup group, String name)
  public Thread(ThreadGroup group, Runnable| runnable, String name)
- In presenza di un security manager, il relativo metodo checkAccess viene invocato e riceve come parametro il ThreadGroup; se non si dispone dei permessi necessari, viene sollevata una SecurityException; l'utilizzo di criteri di sicurezza è una delle ragioni per l'uso dei gruppi di thread
- Se si crea un nuovo Thread senza specificare il suo gruppo, il runtime system automaticamente pone il nuovo thread nello stesso gruppo (current thread group) del thread che lo ha creato (current thread)
- Quando una applicazione Java inizia, il Java runtime system crea un ThreadGroup di nome main. Se non specificato altrimenti, tutti I nuovi thread creati diventano membri del main thread group.
- E' possibile creare un proprio gruppo di thread :

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group");
Thread myThread = new Thread(myThreadGroup,"a thread");
```

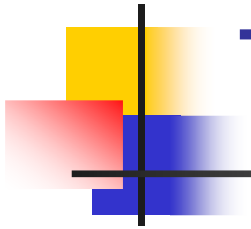


Thread user e daemon

- Se un'applicazione non crea alcun thread, esiste solo il thread che esegue il main, e l'applicazione avrà termine quando termina l'esecuzione del main
- Quando invece un'applicazione crea altri thread, che saranno quindi in aggiunta a quello che esegue il main, l'applicazione finirà ancora semplicemente con la fine del main o attende la terminazione di tutti i thread che essa ha generato?
- La risposta dipende dal tipo di thread che ha generato: esistono i thread user e i thread daemon, opzione impostabile con il metodo setDaemon(true) invocato su un thread t dopo averlo creato e prima di invocare start:

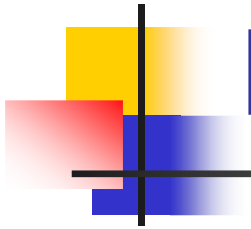
```
Thread t1 = new Thread(this);
t1.setDaemon(true); //t1 è il daemon
t1.start();
```

- Lo stato di demone, una volta impostato, non può essere mutato



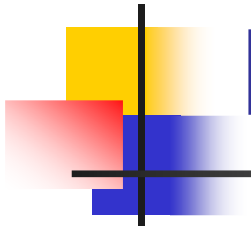
Thread user e daemon

- Se vengono generati solo thread daemon, opzione NON di default, l'applicazione termina quando termina il primo thread di partenza (quello che esegue il main); in quel momento, anche tutti i thread demoni sono stoppati. Se si desidera questo comportamento, devono quindi essere marcati esplicitamente come demoni tutti i thread generati
- Se esiste invece almeno un thread utente, l'applicazione non termina con il thread del main, ma solo quando tutti i thread utente saranno terminati; al completamento dell'ultimo thread utente, anche tutti gli eventuali thread demoni saranno arrestati e l'applicazione potrà avere fine
- i thread utente sono quindi dotati di maggiore indipendenza rispetto ai demoni, e permettono ad un'applicazione di avere una vita più lunga rispetto al thread del solo main. Questa situazione potrebbe in alcuni casi non essere desiderabile; esiste sempre la possibilità di invocare il metodo exit di System.



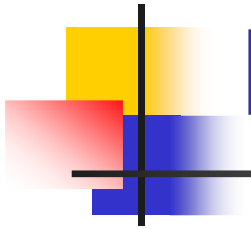
Modello della memoria

- L'ordine delle azioni di un thread è determinato dall'ordine delle istruzioni del programma così come sono scritte dal programmatore (**ordine del programma**)
- invece i valori di una qualsiasi variabile letta dal thread sono determinati dal **modello della memoria**, la cui effettiva implementazione definisce l'insieme di valori ammissibili che possono essere restituiti al momento della lettura (richiesta con una qualche istruzione presente all'interno del thread)
- Solitamente, ci si aspetta che questo insieme di valori ammissibili sia ad ogni istante costituito da un solo valore, coincidente con il valore più recente scritto nella variabile da leggere da un qualche thread che ha operato su di essa; in realtà il modello della memoria può anche operare differentemente, rendendo invisibile il valore più recente



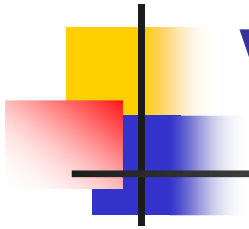
Modello della memoria

- Per forzare quello che da un programmatore verrebbe considerato l'unico comportamento corretto possibile, è possibile utilizzare la sincronizzazione: quando l'accesso ad una variabile viene effettuato all'interno di un blocco sincronizzato, allora il modello della memoria garantisce che ogni lettura restituisce il valore più recente
- Esiste un secondo meccanismo, più debole ma che permette di non utilizzare thread, attivabile mediante lo specificatore **volatile** applicato alla variabile su cui si desidera avere il comportamento sopra indicato (ad ogni lettura cioè, si vuole sempre essere sicuri di ottenere il valore più recente)
- L'utilizzo di variabili volatili sostituisce raramente operazioni sincronizzate perché non fornisce atomicità per azioni diverse dalla semplice lettura; l'uso quindi è limitato



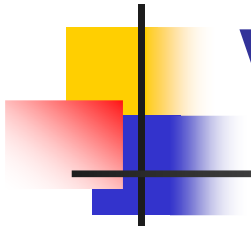
Modello della memoria

- Per inciso, il modello della memoria può influenzare anche l'ordine di esecuzione delle istruzioni; in genere esso tenta di garantire l'approccio happens-before, ossia un'istruzione che nell'ordine del programma è scritta prima di un'altra, si verificherà prima di questa, come il programmatore desidererebbe; potrebbe tuttavia anche aversi un ordine effettivo di esecuzione differente dall'ordine del programma, a patto che al thread di esecuzione dello stesso l'effetto appaia come se venisse rispettato l'ordine del programma; questo eventuale cambiamento dell'ordine effettivo permette compilatori con ottimizzazioni sofisticate, e il più delle volte può essere ignorato dal programmatore



Varie sui thread

- Un thread può essere **interrotto** invocando su di esso il metodo interrupt(); tale metodo rappresenta però solo una segnalazione per il thread che lo riceve:
  - se il thread stava lavorando, continuerà a farlo
  - se il thread stava in wait(), viene sollevata un'eccezione
  - se il thread era bloccato su un'operazione di I/O, su alcuni sistemi si potrebbe ottenere l'eccezione, ma generalmente il thread resta bloccato e interrupt non ha effetto
- un thread può anche aspettare che un altro finisca, senza pretendere un notify da questo; a tale scopo, si invoca il metodo **join**() sul thread del quale si desidera attendere il completamento; join consente anche di specificare un timeout come parametro, per evitare eventuali attese infinite; t.join() è equivalente a while (t.isalive()) wait();



Varie sui thread

- Se un thread solleva **un'eccezione**, ed essa non viene intercettata durante la vita del thread, alla sua morte scompare anche l'eccezione; un'eccezione non intercettata tuttavia potrebbe nascondere un errore grave; Java permette di specificare un eventuale gestore delle eccezioni non intercettate dal thread stesso tramite l'interfacca UncaughtException
- Per il **debugging** dei thread, esistono diversi metodi:
  - toString(), che stampa nome, priorità e gruppo di appartenenza di un thread
  - getState(), che ritorna lo stato come tipo enumerativo
  - dumpstack(), che visualizza lo stack delle chiamate per un thread