

Programmazione Concorrente in Java

Lorenzo Bettini

Dip. Sistemi e Informatica, Univ. Firenze
<http://www.dsi.unifi.it/~bettini>

Ottobre 2005

Gestione dei Processi

- Un processo è un programma in esecuzione.
- Più processi possono eseguire lo stesso programma.
- Il SO ha il compito di:
 - curare l'esecuzione del programma;
 - nel caso più programmi possano essere eseguiti contemporaneamente il SO si occupa di assegnare il processore, ciclicamente, ai vari processi.

Svantaggi dei Processi

- Due processi che eseguono lo stesso programma ne condividono solo il codice, non le risorse
- Non è immediato far comunicare due processi
 - Memoria condivisa
 - Pipe
 - File temporanei
 - ecc.

Multithreading

- Un *thread* è una porzione di processo che esegue contemporaneamente insieme ad altri thread dello stesso processo
- Ogni thread:
 - Ha i propri program counter, registri, Stack
 - Condivide con gli altri thread dello stesso processo: codice, dati, risorse, memoria.
- Multithreading è simile ad un “multitasking all’interno dello stesso processo”

Interazione con l'utente

- Un programma esegue certe operazioni oltre a rispondere agli input dell'utente
 - Invece di controllare periodicamente se l'utente ha inserito dei dati in input...
 - un thread esegue le operazioni principali
 - un altro thread gestisce l'input dell'utente
- Mentre un thread è in attesa di input l'altro thread può effettuare altre operazioni

Operazioni di I/O

- Dal punto di vista del processore, il tempo che intercorre fra due pressioni di tasti è enorme, e può essere sfruttato per altre operazioni.
- Anche le operazioni di lettura/scrittura su un hard disk sono lentissime se confrontate con la velocità di esecuzione del processore
- Ancora di più l'accesso alla rete...

Processori multipli

- Se la macchina dispone di più di un processore, i programmi multithreaded sfrutteranno i processori in parallelo
 - Se il sistema operativo gestisce processori multipli
 - Se la JVM gestisce processori multipli
- Tali programmi multithreaded non dovranno essere riscritti:
 - useranno automaticamente queste caratteristiche

Svantaggi

- Ogni thread usa ulteriori risorse di memoria
- Overhead dovuto allo scheduling dei thread
- *Context switch*:
 - Quando un thread viene sospeso e viene eseguito un altro thread
 - Sono necessari diversi cicli di CPU per il context switch e se ci sono molti thread questo tempo diventa significativo
- È necessario del tempo per creare un thread, farlo partire, e deallocare le sue risorse quando ha terminato
 - Ad es., invece di creare un thread ogni 5 minuti per controllare la posta, è meglio crearlo una volta e metterlo in pausa per 5 minuti fra un controllo e l'altro.

Programmi senza input/output

In questi casi, se non si dispone di più processori è meglio non utilizzare i thread in quanto il programma non solo non ne trarrà beneficio, ma peggiorerà le prestazioni.

Calcolo matematico

In programmi di solo calcolo matematico si dovrebbero usare i thread solo se si dispone di un multiprocessore.

Part I

Programmazione Multithreading in Java

Threads in Java

La classe principale è `java.lang.Thread`

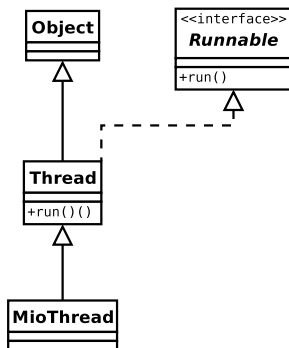


Figure: Thread hierarchy

Usare i thread in Java

Passi principali:

- 1 Estendere la classe `java.lang.Thread`
- 2 Riscrivere (ridefinire, override) il metodo `run()` nella sottoclasse di Thread
- 3 Creare un'istanza di questa classe derivata
- 4 Richiamare il metodo `start()` su questa istanza

Il metodo run()

- L'implementazione di run() in Thread non fa niente
- Il metodo run() costituisce l'*entry point* del thread:
 - Ogni istruzione eseguita dal thread è inclusa in questo metodo o nei metodi invocati direttamente o indirettamente da run()
 - Un thread è considerato *alive* finché il metodo run() non ritorna
 - Quando run() ritorna il thread è considerato *dead*
- Una volta che un thread è "morto" non può essere rieseguito (pena un'eccezione `IllegalThreadStateException`): se ne deve creare una nuova istanza.
- Non si può far partire lo stesso thread (la stessa istanza) più volte

Primo Esempio

ThreadExample

```
public class ThreadExample extends Thread {  
    public void run() {  
        for (int i = 0; i < 20; ++i)  
            System.out.println("Nuovo thread");  
    }  
}
```

Far partire il thread: `start()`

- Una chiamata di `start()` ritorna immediatamente al chiamante senza aspettare che l'altro thread abbia effettivamente iniziato l'esecuzione
 - semplicemente la JVM viene avvertita che l'altro thread è pronto per l'esecuzione (quando lo scheduler lo riterrà opportuno)
- Prima o poi verrà invocato il metodo `run()` del nuovo thread
- I due thread saranno eseguiti in modo concorrente ed indipendente

Importante

L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine in cui le istruzioni dei vari thread saranno eseguite effettivamente è indeterminato (*nondeterminismo*).

Primo Esempio (completo)

ThreadExample

```
public class ThreadExample extends Thread {  
    public void run() {  
        for (int i = 0; i < 20; ++i)  
            System.out.println("Nuovo thread");  
    }  
  
    public static void main(String[] args) {  
        ThreadExample t = new ThreadExample();  
        t.start();  
  
        for (int i = 0; i < 20; ++i)  
            System.out.println("Main thread");  
    }  
}
```


Applicazione e thread

- C'è sempre un thread in esecuzione: quello che esegue il metodo `main()`, chiamiamolo *main thread*
- Quando il main thread esce dal main il programma NON necessariamente termina
- Finché ci sono thread in esecuzione il programma NON termina

Fare una pausa: busy loop?

```
// wait for 60 seconds  
long startTime = System.currentTimeMillis();  
long stopTime = startTime + 60000;  
  
while (System.currentTimeMillis() < stopTime) {  
    // do nothing, just loop  
}
```

Evitare

Utilizza inutilmente cicli del processore!

Fare una pausa: `sleep()`

Il metodo `Thread.sleep()`

```
public static native void sleep(long msToSleep)  
    throws InterruptedException
```

- Non utilizza cicli del processore
- è un metodo *statico* e mette in pausa il thread corrente
- non è possibile mettere in pausa un altro thread
- mentre un thread è in sleep può essere *interrotto* da un altro thread:
 - in tal caso viene sollevata un'eccezione `InterruptedException`
 - quindi `sleep()` va eseguito in un blocco `try` `catch` (oppure il metodo che lo esegue deve dichiarare di sollevare tale eccezione)

Esempio di sleep()

```
public class ThreadSleepExample extends Thread {  
    public void run() {  
        for (int i = 0; i < 20; ++i) {  
            System.out.println("Nuovo thread");  
            try { Thread.sleep(200); }  
            catch (InterruptedException e) { return; }  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        new ThreadSleepExample().start();  
  
        for (int i = 0; i < 20; ++i) {  
            System.out.println("Main thread");  
            Thread.sleep(200);  
        }  
    }  
}
```

Thread corrente

- Siccome una stessa sequenza di istruzioni può essere eseguita da thread differenti, potrebbe essere utile sapere quale thread la sta effettivamente eseguendo
- Si può utilizzare il metodo statico `currentThread()` che restituisce il thread (istanza di classe `Thread`) corrente

Thread & OOP

L'esempio seguente mostra anche che i metodi di un'istanza `Thread` possono essere eseguiti anche da un'altro thread, non necessariamente dal thread dell'istanza.

Thread & istanze

È sbagliato assumere che all'interno di un metodo di una classe thread il `this` corrisponda al thread corrente!

Esempio: currentThread()

```

public class CurrentThreadExample extends Thread {
    private Thread creatorThread;

    public CurrentThreadExample() {
        creatorThread = Thread.currentThread();
    }

    public void run() {
        for (int i = 0; i < 1000; ++i)
            printMsg();
    }

    public void printMsg() {
        Thread t = Thread.currentThread();
        if (t == creatorThread) {
            System.out.println("Creator thread");
        } else if (t == this) {
            System.out.println("New thread");
        } else {
            System.out.println("Unknown thread");
        }
    }
}

public static void main(String[] args) {
    CurrentThreadExample t =
        new CurrentThreadExample();
    t.start();

    for (int i = 0; i < 1000; ++i)
        t.printMsg();
}

```

Thread e nomi

- Ad ogni thread è associato un nome
- utile per identificare i vari thread
- se non viene specificato ne viene generato uno di default durante la creazione
- il nome può essere passato al costruttore
- metodi:
 - `String getName()`
 - `setName(String newName)`

Altro esempio

```
class EsempioThread1 extends Thread {  
    private char c;  
  
    public EsempioThread1( String name, char c )  
    {  
        super( name );  
        this.c = c;  
    }  
  
    public void run()  
    {  
        for (int i = 0; i < 100; ++i)  
            System.out.print(c);  
  
        System.err.println( "\n" + getName() + " finito" );  
    }  
}
```


Altro esempio

```
public class ThreadTest1 {  
    public static void main( String args[] ) {  
        EsempioThread1 thread1, thread2, thread3, thread4;  
        thread1 =  
            new EsempioThread1( "thread1", '@' );  
        thread2 =  
            new EsempioThread1( "thread2", '*' );  
        thread3 =  
            new EsempioThread1( "thread3", '#' );  
        thread4 =  
            new EsempioThread1( "thread4", '+' );  
  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
    }  
}
```

- Il main thread crea più istanze della stessa thread class.
- Dopo aver fatto partire i thread non fa nient'altro e termina

Usare i thread in Java (alternativa)

Nel caso si sia già utilizzata l'ereditarietà (Java non supporta l'ereditarietà multipla).

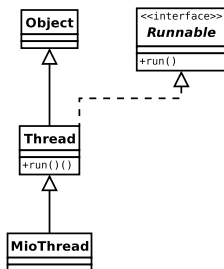


Figure: Thread hierarchy

- 1 Creare una classe che implementa `java.lang.Runnable` (la stessa implementata anche da Thread)
- 2 Implementare il metodo `run()` in questa classe
- 3 Creare un'istanza di Thread passandogli un'istanza di questa classe
- 4 Richiamare il metodo `start()` sull'istanza di Thread

Esempio di Runnable

```
class EsempioThread2 implements Runnable {
    String name;
    private int sleepTime;

    public EsempioThread2( String name ) {
        this.name = name;
        // pausa fra 0 e 5 secondi
        sleepTime = (int) ( Math.random() * 5000 );
        System.out.println( "Name: " + name + "; sleep: " + sleepTime );
    }

    public void run() {
        for (int i = 0; i < 5; ++i) {
            System.out.println (name + " : in esecuzione.");
            try { Thread.sleep(sleepTime); }
            catch (InterruptedException e) {}
        }
        System.err.println( name + " finito" );
    }
}
```

Esempio di Runnable

```
public class ThreadTest2 {  
    public static void main( String args[] )  
    {  
        Thread thread1, thread2, thread3, thread4;  
  
        thread1 = new Thread(new EsempioThread2( "thread1" ));  
        thread2 = new Thread(new EsempioThread2( "thread2" ));  
        thread3 = new Thread(new EsempioThread2( "thread3" ));  
        thread4 = new Thread(new EsempioThread2( "thread4" ));  
  
        System.err.println( "\nI thread stanno per partire" );  
  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
  
        System.err.println( "I thread sono partiti\n" );  
    }  
}
```

Terminazione con System.exit()

Usando questa funzione, l'applicazione termina, anche se ci sono ancora thread in esecuzione

Uscita dall'applicazione in modo "brusco"

```
public class ThreadTestExitErr {  
    public static void main(String args[]) throws Exception {  
        System.err.println("\nI thread stanno per partire");  
        new EsempioThreadSleep("thread1").start();  
        new EsempioThreadSleep("thread2").start();  
        new EsempioThreadSleep("thread3").start();  
        new EsempioThreadSleep("thread4").start();  
  
        System.err.println("I thread sono partiti\n");  
        Thread.sleep(2000);  
        System.err.println("Il main chiude l'applicazione");  
        System.exit(0);  
    }  
}
```

Metodo `isAlive()`

- Può essere utilizzato per testare se un thread è “vivo”:
 - quando viene chiamato `start()` il thread è considerato “alive”
 - il thread è considerato “alive” finché il metodo `run()` non ritorna

Usare `isAlive()` per “attendere” un thread?

Si potrebbe utilizzare `isAlive()` per testare periodicamente se un thread è ancora “vivo”, ma non è efficiente.

Attendere la terminazione: `join()`

- Il metodo `join()` attende la terminazione del thread sul quale è richiamato
- Il thread che esegue `join()` rimane così bloccato in attesa della terminazione dell'altro thread
- Il metodo `join()` può lanciare una `InterruptedException`
- Ne esiste una versione per specificare il timeout di attesa

Esempio: join()

```
public class ThreadTestExitJoin {  
    public static void main(String args[]) throws Exception {  
        Thread thread1, thread2, thread3, thread4;  
        thread1 = new EsempioThreadSleep("thread1");  
        thread2 = new EsempioThreadSleep("thread2");  
        thread3 = new EsempioThreadSleep("thread3");  
        thread4 = new EsempioThreadSleep("thread4");  
  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
  
        thread1.join();  
        thread2.join();  
        thread3.join();  
        thread4.join();  
  
        System.exit(0);  
    }  
}
```


Terminare un thread: `stop()`

- Il metodo `stop()` termina immediatamente un thread
- Tale metodo è stato deprecato all'uscita del JDK 1.2:
 - Il thread terminato non ha il tempo di rilasciare eventuali risorse
 - Si possono così avere dati corrotti
 - I *lock* acquisiti non vengono rilasciati quindi si possono avere situazioni di *deadlock*

ATTENZIONE

Assolutamente da EVITARE!

Terminare un thread: `interrupt()`

- Il metodo `interrupt()` setta un flag di interruzione nel thread di destinazione e ritorna
- Il thread interrotto non viene effettivamente interrotto (quindi al ritorno di `interrupt()` non si può assumere che il thread sia stato effettivamente interrotto):
 - Il thread può controllare se tale flag è settato e nel caso uscire (dal `run()`)
 - I metodi che mettono in pausa un thread controllano il flag di interruzione prima e durante lo stato di pausa
 - Se tale flag risulta settato, allora lanciano un'eccezione `InterruptedException` (e resettano il flag)
 - Il thread che era stato interrotto intercetta l'eccezione e "dovrebbe" terminare l'esecuzione, ma questo non viene controllato
 - Ci si affida alla correttezza del programmatore!

Implementazione corretta

```
class EsempioThreadSleep extends Thread {  
  
    public void run() {  
        for (int i = 0; i < 5; ++i) {  
            System.out.println(name + " : in esecuzione.");  
            try {  
                Thread.sleep(sleepTime);  
            } catch (InterruptedException e) {  
                System.err.println(name + " interrotto");  
                break;  
            }  
        }  
  
        System.err.println(name + " finito");  
    }  
}
```

Interrompere i thread

```
public class ThreadTestInterrupt {  
    public static void main(String args[]) throws Exception {  
        // crea i thread e li lancia  
        Thread.sleep(2000);  
  
        thread1.interrupt();  
        thread2.interrupt();  
        thread3.interrupt();  
        thread4.interrupt();  
  
        thread1.join();  
        thread2.join();  
        thread3.join();  
        thread4.join();  
  
        System.exit(0);  
    }  
}
```

Implementazione scorretta

```
class EsempioThreadMalicious extends Thread {  
  
    public void run() {  
        for (int i = 0; i < 20; ++i) {  
            System.out.println(name + " : in esecuzione.");  
            try {  
                Thread.sleep(sleepTime);  
            } catch (InterruptedException e) {  
                System.err.println(name + " interrotto ma continuo :->");  
            }  
        }  
  
        System.err.println(name + " finito");  
    }  
}
```

Utilizzo di timeout

```
public class ThreadTestInterruptMaliciousTimeout {  
    public static void main(String args[]) throws Exception {  
        Thread thread1, thread2, thread3, thread4;  
        thread1 = new EsempioThreadSleep("thread1");  
        thread2 = new EsempioThreadSleep("thread2");  
        thread3 = new EsempioThreadSleep("thread3");  
        thread4 = new EsempioThreadMalicious("thread4");  
        // ... fa partire i thread...  
        thread1.interrupt();  
        thread2.interrupt();  
        thread3.interrupt();  
        thread4.interrupt();  
  
        thread1.join(1000);  
        thread2.join(1000);  
        thread3.join(1000);  
        thread4.join(1000);  
        System.exit(0);  
    }  
}
```

Ulteriori problemi

- Il metodo `interrupt()` non funziona se il thread “interrotto” non esegue mai metodi di attesa
- Ad es. se un thread si occupa di effettuare calcoli in memoria, non potrà essere interrotto in questo modo
- I thread devono cooperare:
 - Un thread può controllare periodicamente il suo stato di “interruzione”:
 - `isInterrupted()` controlla il flag di interruzione senza resettarlo
 - `Thread.interrupted()` controlla il flag di interruzione del thread corrente e se settato lo resetta

Esempio: isInterrupted

```
public void run() {  
    while (condizione) {  
        // esegue un'operazione complessa  
        if (Thread.interrupted()) {  
            break;  
        }  
    }  
}
```


Collaborazione: Thread.yield()

- Permette ad un thread di lasciare volontariamente il processore ad un altro thread
- Utile nel caso un thread che esegue spesso operazioni che non lo mettono in attesa

Non abusarne

L'operazione richiede del tempo e può dar luogo ad un context switch.

Esercizio

Realizzare una classe contenitore (usando una classe contenitore del pacchetto `java.util`) di thread con le seguenti operazioni:

- `insert`: inserisce un thread nel contenitore
- `start`: avvia tutti i thread contenuti, che non sono ancora partiti
- `interrupt`: interrompe tutti i thread contenuti
- `join`: attende che tutti i thread contenuti abbiano terminato l'esecuzione

Esercizio 2

Variazione: la possibilità di identificare all'interno del contenitore i thread con i loro nomi:

- `insert`: come sopra ma non inserisce il thread se ne esiste già uno con lo stesso nome (e ritorna `false`)
- `get(name)`: ritorna il thread col nome specificato (o `null` altrimenti)
- `interrupt(name)`: interrompe il thread col nome specificato
- `join(name)`: attende la terminazione del thread specificato
- `remove(name)`: rimuove dal contenitore il thread selezionato, lo interrompe e ne attende la terminazione

Part II

Accesso Concorrente a Risorse Condivise

Condivisione dati

- I thread non sono del tutto indipendenti
- Le operazioni su risorse condivise non sono eseguite in modo atomico:
 - Quando i thread eseguono operazioni su dati condivisi possono essere interrotti da un context switch prima che abbiano terminato la “transazione”

Esempio (errato)

```
class AssegnatoreErrato {  
    private int tot_posti = 20;  
  
    public boolean assegna_posti(String cliente, int num_posti) {  
        System.out.println("--Richiesta di " + num_posti + " da " + cliente);  
        if (tot_posti >= num_posti) {  
            System.out.println("---Assegna " + num_posti + " a " + cliente);  
            tot_posti -= num_posti;  
            return true;  
        }  
        return false;  
    }  
  
    int totale_posti() { return tot_posti; }  
}
```

Problemi

- Se più thread eseguono quella parte di codice in parallelo e si ha un context switch nel mezzo della transazione, la risorsa sarà in uno stato inconsistente:
 - Il numero di posti assegnato alla fine sarà maggiore di quello realmente disponibile!
 - *Race condition*
 - *Codice non rientrante*
- Non è detto che il problema si verifichi ad ogni esecuzione (non determinismo)

Esempio di client

```
public class Richiedente extends Thread {  
    private int num_posti;  
  
    private Assegnatore assegnatore;  
  
    public Richiedente(String nome, int num_posti, Assegnatore assegnatore) {  
        super(nome);  
        this.num_posti = num_posti;  
        this.assegnatore = assegnatore;  
    }  
  
    public void run() {  
        System.out.println("-" + getName() + ": richiede " + num_posti + "...");  
        if (assegnatore.assegna_posti(getName(), num_posti))  
            System.out.println("-" + getName() + ": ottenuti " + num_posti  
                               + "...");  
        else  
            System.out.println("-" + getName() + ": posti non disponibili");  
    }  
}
```


Esempio (errato)

```
public class AssegnaPostiErrato {  
    public static void main(String args[]) throws InterruptedException {  
        AssegnatoreErrato assegnatore = new AssegnatoreErrato ();  
  
        Richiedente client1 =  
            new Richiedente("cliente1", 3, assegnatore);  
        Richiedente client2 =  
            new Richiedente("cliente2", 10, assegnatore);  
        Richiedente client3 =  
            new Richiedente("cliente3", 5, assegnatore);  
        Richiedente client4 =  
            new Richiedente("cliente4", 3, assegnatore);  
  
        client1.start (); client2.start (); client3.start (); client4.start ();  
        client1.join(); client2.join(); client3.join(); client4.join();  
  
        System.out.println("Numero di posti ancora disponibili: " +  
            assegnatore.totale_posti ());  
    }  
}
```

Accesso sincronizzato

- Un solo thread alla volta deve eseguire il metodo `assegna_posti`
- Se un thread lo sta già eseguendo gli altri thread che cercano di eseguirlo dovranno aspettare
- Più esecuzioni concorrenti di quel metodo devono in realtà avvenire in modo sequenziale

Accesso sincronizzato

- Ogni oggetto (istanza di `Object`) ha associato un *mutual exclusion lock*
- Non si può accedere direttamente a questo lock, però:
 - gestito automaticamente quando si dichiara un metodo o un blocco di codice come `synchronized`

Accesso sincronizzato

- Quando un metodo è `synchronized` lo si può invocare su un oggetto **solo se si è acquisito il lock su tale oggetto**
- Quindi i metodi `synchronized` hanno accesso esclusivo ai dati incapsulati nell'oggetto (se a tali dati si accede solo con metodi `synchronized`)
- I metodi non `synchronized` non richiedono l'accesso al lock e quindi si possono richiamare in qualsiasi momento

Variabili locali

Poiché ogni thread ha il proprio stack, se più thread stanno eseguendo lo stesso metodo, ognuno avrà la propria copia delle variabili locali, senza pericolo di “interferenza”.

Accesso mutuamente esclusivo completo

```
public class SharedInteger {  
    private int theData;  
  
    public SharedInteger(int data) { theData = data; }  
  
    public synchronized int read() { return theData; }  
  
    public synchronized void write(int newValue) { theData = newValue; }  
  
    public synchronized void incrementBy(int by) { theData += by; }  
}
```

Monitor

- Collezione di dati e procedure
- I dati sono accessibili in mutua esclusione solo tramite la chiamata di una procedura del monitor:
 - Quando un thread sta eseguendo una procedura del monitor, gli altri thread devono attendere
 - Il thread che sta eseguendo una procedura del monitor può eseguire altre procedure del monitor

Monitor in Java

- Oggetto con metodi synchronized (Ogni oggetto può essere un monitor)
- Solo un thread alla volta può eseguire un metodo synchronized su uno stesso oggetto:
 - Prima di eseguirlo deve ottenere il lock (mutua esclusione)
 - Appena uscito dal metodo il lock viene rilasciato
- Se un thread sta eseguendo un metodo synchronized altri thread non possono eseguire quel metodo o altri metodi synchronized sullo stesso oggetto
 - Ma possono eseguire metodi non synchronized sullo stesso oggetto
- Il thread che sta eseguendo un metodo synchronized può eseguire altri metodi synchronized sullo stesso oggetto

Esempio (corretto)

```
class Assegnatore {  
    private int tot_posti = 20;  
  
    public synchronized boolean assegna_posti(String cliente, int num_posti) {  
        System.out.println("--Richiesta di " + num_posti + " da " + cliente);  
        if (tot_posti >= num_posti) {  
            System.out.println("---Assegna " + num_posti + " a " + cliente);  
            tot_posti -= num_posti;  
            return true;  
        }  
        return false;  
    }  
  
    int totale_posti() { return tot_posti; }  
}
```


In dettaglio: synchronized

- Quando un thread deve eseguire un metodo `synchronized` su un oggetto si blocca finché non riesce ad ottenere il lock sull'oggetto
- Quando lo ottiene può eseguire il metodo (e tutti gli altri metodi `synchronized`)
- Gli altri thread rimarranno bloccati finché il lock non viene rilasciato
- Quando il thread esce dal metodo `synchronized` rilascia automaticamente il lock
- A quel punto gli altri thread proveranno ad acquisire il lock
- Solo uno ci riuscirà e gli altri torneranno in attesa

Blocchi synchronized

- Singoli blocchi di codice possono essere dichiarati synchronized su un certo oggetto
- Un solo thread alla volta può eseguire un tale blocco su uno stesso oggetto
- Permette di minimizzare le parti di codice da serializzare ed aumentare il parallelismo

Equivalenza

```
public synchronized int read() {  
    return theData;  
}  
  
≡  
  
public int read() {  
    synchronized (this) {  
        return theData;  
    }  
}
```

Blocchi synchronized

Implementano una sorta di *Sezioni Critiche*

Metodi `synchronized` vs. blocchi `synchronized`

Si perde la possibilità di rendere evidente (anche nella documentazione) i vincoli di sincronizzazione nell'interfaccia della classe.

Esempio (alternativa)

```
public class Assegnatore2 extends Assegnatore {  
    private int tot_posti = 20;  
  
    public boolean assegna_posti(String cliente, int num_posti) {  
        System.out.println("--Richiesta di " + num_posti + " da " + cliente);  
        synchronized (this) {  
            if (tot_posti >= num_posti) {  
                System.out.println("---Assegna " + num_posti + " a " + cliente);  
                tot_posti -= num_posti;  
                return true;  
            }  
        }  
        return false;  
    }  
  
    int totale_posti() { return tot_posti; }  
}
```

Blocchi synchronized

- Ci si può “sincronizzare” anche su oggetti differenti dal `this`
- In questo caso la correttezza è affidata a chi usa l'oggetto condiviso (client) e non all'oggetto stesso
- Spesso questa è l'unica alternativa quando non si può/vuole modificare la classe dell'oggetto condiviso

Esempio di client sincronizzato

```
public class RichiedenteSincronizzato extends Thread {  
    private int num_posti;  
    private AssegnatoreErrato assegnatore;  
  
    /* costruttore omissso */  
  
    public void run() {  
        System.out.println("-" + getName() + ": richiede " + num_posti + "...");  
        synchronized (assegnatore) {  
            if (assegnatore.assegna_posti(getName(), num_posti))  
                System.out.println("-" + getName() + ": ottenuti " + num_posti  
                    + "...");  
            else  
                System.out.println("-" + getName() + ": posti non disponibili");  
        }  
    }  
}
```

Esempio di client sincronizzato (2)

- Si può utilizzare un client sincronizzato di un oggetto a sua volta sincronizzato. Visto che l'oggetto è lo stesso tutto continua a funzionare:
- Quando il client andrà a richiamare il metodo sincronizzato avrà già ottenuto il lock sull'oggetto

Esempio di client sincronizzato (2)

```
public class RichiedenteSincronizzato2 extends Thread {  
    private int num_posti;  
    private Assegnatore assegnatore;  
  
    /* costruttore omissso */  
  
    public void run() {  
        System.out.println("-" + getName() + ": richiede " + num_posti + "...");  
        synchronized (assegnatore) {  
            if (assegnatore.assegna_posti(getName(), num_posti))  
                System.out.println("-" + getName() + ": ottenuti " + num_posti  
                    + "...");  
            else  
                System.out.println("-" + getName() + ": posti non disponibili");  
        }  
    }  
}
```


Ereditarietà & synchronized

- La specifica `synchronized` non fa parte vera e propria della segnatura di un metodo
- Quindi una classe derivata può ridefinire un metodo `synchronized` come non `synchronized` e viceversa

```
class Base {  
    public void synchronized m1() { /* ... */ }  
    public void synchronized m2() { /* ... */ }  
}  
  
class Derivata extends Base {  
    public void m1() { /* not synchronized */ }  
  
    public void m2() {  
        // do stuff not synchronized  
        super.m2(); // synchronized here  
        // do stuff not synchronized  
    }  
}
```

Ereditarietà & synchronized

Si può quindi ereditare da una classe “non sincronizzata” e ridefinire un metodo come `synchronized`, che richiama semplicemente l'implementazione della superclasse:

Ridefinire un metodo come `synchronized`

```
public class AssegnatoreDerivato extends AssegnatoreErrato {  
    public synchronized boolean assegna_posti(String cliente, int num_posti) {  
        System.out.println("Assegnatore derivato");  
        return super.assegna_posti(cliente, num_posti);  
    }  
}
```

Questo assicura che l'implementazione della superclasse (non sincronizzata) avvenga adesso in modo sincronizzato.

Variabili statiche

- metodi e blocchi `synchronized` non assicurano l'accesso mutuamente esclusivo ai dati "statici"
 - I dati statici sono condivisi da tutti gli oggetti della stessa classe
- In Java ad ogni classe è associato un oggetto di classe **Class**
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto **Class**:
 - si può dichiarare un metodo statico come `synchronized`
 - si può dichiarare un blocco come `synchronized` sull'oggetto **Class**

Attenzione

Il lock a livello classe NON si ottiene quando ci si sincronizza su un oggetto di tale classe e viceversa.

Sincronizzazione a livello classe

```
class StaticSharedVariable {  
    // due modi per ottenere un lock a livello classe  
  
    private static int shared;  
  
    public int read() {  
        synchronized(StaticSharedVariable.class) {  
            return shared;  
        }  
    }  
  
    public synchronized static void write(int i) {  
        shared = i;  
    }  
}
```

Campi “volatili”

- Un campo può essere dichiarato come **volatile**
- Java richiede che un campo volatile non deve essere mantenuto in una memoria locale
- tutte le letture e scritture devono essere fatte in memoria principale (condivisa)
- Le operazioni su campi volatili devono essere eseguite esattamente nell'ordine richiesto dal thread
- le variabili `long` e `double` devono essere lette e scritte in modo atomico

È comodo quando un solo thread modifica il valore di un campo e tutti gli altri thread lo possono leggere in ogni momento.

Cooperazione fra thread

- Tramite la sincronizzazione un thread può modificare in modo sicuro dei valori che potranno essere letti da un altro thread
- Ma come fa l'altro thread a sapere che i valori sono stati modificati/aggiornati?

No Busy Loop!

```
while (getValue() != desiredValue) {  
    Thread.sleep(500);  
}
```

Esempio: Produttore–Consumatore

- Non leggere quando la memoria è vuota
- Non scrivere quando la memoria è piena
- Usare solo parti di codice `synchronized` non è sufficiente:
 - Produttore acquisisce il lock
 - La memoria è piena, aspetta che si svuoti
 - Perché si liberi il consumatore deve accedere alla memoria, che è bloccata dal produttore
 - probabile Deadlock!

Esempio: Produttore–Consumatore

- Non leggere cose già lette
- Non sovrascrivere cose non ancora lette
- Usare solo parti di codice `synchronized` non è sufficiente
 - Produttore acquisisce il lock
 - Scrive una nuova informazione
 - Produttore riacquisisce il lock nuovamente, prima del consumatore
 - Produttore sovrascrive un'informazione non ancora recuperata dal consumatore

Esempio: Produttore–Consumatore (errato)

```
class ProduttoreErrato extends Thread {  
    CellaCondivisaErrata cella;  
  
    public ProduttoreErrato(CellaCondivisaErrata cella) {  
        this.cella = cella;  
    }  
  
    public void run() {  
        for (int i = 1; i <= 10; ++i) {  
            synchronized (cella) {  
                ++(cella.valore);  
                System.out.println ("Prodotto: " + i);  
            }  
        }  
    }  
}
```

Esempio: Produttore–Consumatore (errato)

```

class ConsumatoreErrato extends Thread {
    CellaCondivisaErrata cella;

    public ConsumatoreErrato(CellaCondivisaErrata cella) {
        this.cella = cella;
    }

    public void run() {
        int valore_letto;
        for (int i = 0; i < 10; ++i) {
            synchronized (cella) {
                valore_letto = cella.valore;
                System.out.println ("Consumato: " + valore_letto);
            }
        }
    }
}

```

Attesa & Notifica

- Un thread può chiamare `wait()` su un oggetto sul quale ha il lock:
 - Il lock viene rilasciato
 - Il thread va in stato di waiting
- Altri thread possono ottenere tale lock
- Effettuano le opportune operazioni e invocano su un oggetto:
 - `notify()` per risvegliare un singolo thread in attesa su quell'oggetto
 - `notifyAll()` per risvegliare tutti i thread in attesa su quell'oggetto
 - I thread risvegliati devono comunque riacquisire il lock
 - I notify non sono cumulativi
- Questi metodi sono definiti nella classe `Object`

Attesa & Notifica

- Ad ogni oggetto Java è associato un *wait-set*: l'insieme dei thread che sono in attesa per l'oggetto
- Un thread viene inserito nel wait-set quando esegue la `wait`
- I thread sono rimossi dal wait-set attraverso le notifiche:
 - `notify` ne rimuove solo uno
 - `notifyAll` li rimuove tutti

Eccezioni

- Wait:
 - `IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor.
 - `InterruptedException` - if another thread has interrupted the current thread.
- Notify:
 - `IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor.

Segnatura

```
public final native void notify()  
    throws IllegalMonitorStateException // RuntimeException  
public final native void notifyAll()  
    throws IllegalMonitorStateException // RuntimeException  
  
public final native void wait()  
    throws InterruptedException,  
        IllegalMonitorStateException // RuntimeException  
public final native void wait(long msTimeout)  
    throws InterruptedException,  
        IllegalMonitorStateException, // RuntimeException  
        IllegalArgumentException // RuntimeException  
public final native void wait(long msTimeout, int nanoSec)  
    throws InterruptedException,  
        IllegalMonitorStateException, // RuntimeException  
        IllegalArgumentException // RuntimeException
```

Variabili Condizione

- Il meccanismo wait/notify non richiede che una variabile sia controllata da un thread e modificata da un altro
- Comunque, è bene utilizzare sempre questo meccanismo insieme ad una variabile
- In questo modo si eviteranno le *missed notification* e le *early notification*
- La JVM (in alcune implementazioni) può risvegliare thread in attesa indipendentemente dall'applicazione (*spurious wake-up*).

Java

non mette a disposizione delle vere e proprie variabili condizione: devono essere implementate dal programmatore.

Esempio: Produttore–Consumatore (corretto)

```

class Produttore extends Thread {
    public void run() {
        for (int i = 1; i <= 10; ++i) {
            synchronized (cella) {
                while (!cella.scrivibile) {
                    try {
                        cella.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                ++(cella.valore);
                cella.scrivibile = false; // cede il turno
                cella.notify(); // risveglia il consumatore
                System.out.println("Prodotto: " + i);
            }
        }
    }
}

```


Esempio: Produttore–Consumatore (corretto)

```

class Consumatore extends Thread {
    public void run() {
        int valore_letto;
        for (int i = 0; i < 10; ++i) {
            synchronized (cella) {
                while (cella.scrivibile) {
                    try {
                        cella.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                valore_letto = cella.valore;
                cella.scrivibile = true; // cede il turno
                cella.notify(); // notifica il produttore
                System.out.println("Consumato: " + valore_letto);
            }
        }
    }
}

```

Esempio: Produttore–Consumatore (alternativa)

```
class CellaCondivisa2 {  
    private int valore = 0;  
    private boolean scrivibile = true;  
  
    public synchronized void produci(int i) throws InterruptedException {  
        while (! scrivibile)  
            wait();  
        valore = i;  
        scrivibile = false;  
        notify();  
    }  
  
    public synchronized int consuma() throws InterruptedException {  
        while (scrivibile)  
            wait();  
        scrivibile = true;  
        notify();  
        return valore;  
    }  
}
```

Esempio: Produttore–Consumatore (alternativa)

```
class Produttore2 extends Thread {  
    public void run() {  
        try {  
            for (int i = 1; i <= 10; ++i) {  
                cella.produci(i);  
                System.out.println ("Prodotto: " + i);  
            }  
        } catch (InterruptedException e) {}  
    }  
}  
  
class Consumatore2 extends Thread {  
    public void run() {  
        try {  
            for (int i = 0; i < 10; ++i) {  
                int valore_letto = cella.consuma();  
                System.out.println ("Consumato: " + valore_letto);  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

notify & notifyAll

- Un `notify` effettuato su un oggetto su cui nessun thread è in wait viene perso (non è un semaforo)
- Se ci possono essere più thread in attesa usare `notifyAll`:
 - Risveglia tutti i thread in attesa
 - Tutti si rimettono in coda per riacquisire il lock
 - Ma solo uno alla volta riprenderà il lock
- Prima che un thread in attesa riprenda l'esecuzione il thread che ha notificato deve rilasciare il monitor (uscire dal blocco `synchronized`)

`notifyAll`

È sempre più sicuro, ma è più inefficiente.

Scenario di problema

- Ci sono più thread in attesa sullo stesso oggetto ma su condizioni differenti
- si usa `notify` ed in questo modo se ne risveglia solo uno
- il thread testa la sua condizione che però risulta ancora falsa e ritorna in attesa
- gli altri thread non hanno la possibilità di testare la loro condizione
- **DEADLOCK!**

Gestire InterruptedException

```
public synchronized void startWrite()
    throws InterruptedException
{
    while (readers > 0 || writing) {
        waitingWriters++;
        wait();
        waitingWriters--;
    }
    writing = true;
}
```

Se viene ricevuta un'eccezione `InterruptedException` viene correttamente propagata, ma il valore `waitingWriters` non viene decrementato.

Gestire InterruptedException

Versione corretta:

```
public synchronized void startWrite()
    throws InterruptedException
{
    try {
        while (readers > 0 || writing) {
            waitingWriters++;
            wait();
            waitingWriters--;
        }
        writing = true;
    } catch (InterruptedException e) {
        waitingWriters--;
        throw e;
    }
}
```

Gestire InterruptedException

Versione corretta (alternativa):

```
public synchronized void startWrite()
    throws InterruptedException
{
    while (readers > 0 || writing) {
        waitingWriters++;
        try {
            wait();
        } finally {
            waitingWriters--;
        }
    }
    writing = true;
}
```