21b.
Il Java Memory Model

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

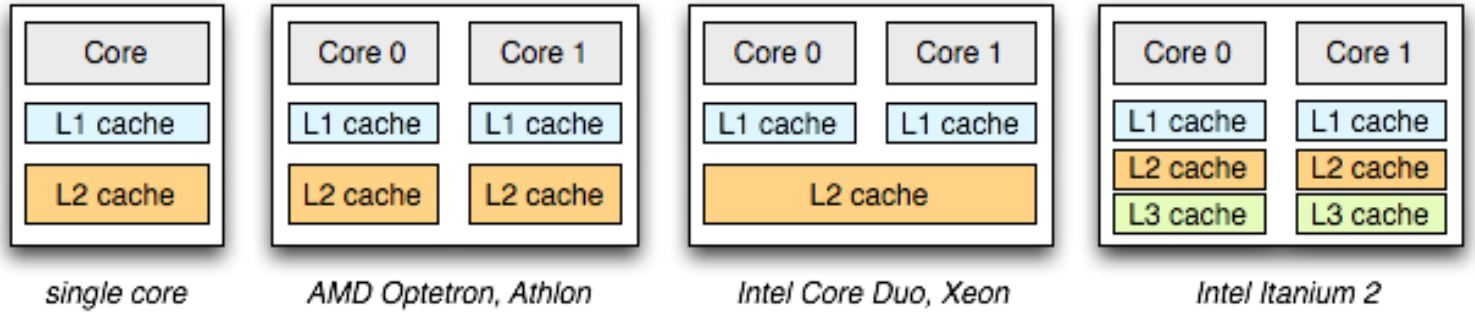
Corso di Linguaggi di Programmazione II

I modelli di memoria

- Un modello di memoria (memory model) è una descrizione di come una architettura hardware (reale o virtuale) gestisce gli accessi alla memoria

- Nelle architetture moderne la memoria è *stratificata* in diversi livelli, che vanno dalla memoria di massa (hard disk), fino ai registri della CPU, passando per diversi livelli di cache
  - questa stratificazione prende il nome di "gerarchia della memoria" (memory hierarchy)
- Alcuni livelli, come i registri e i primi livelli di cache, sono **separati** tra i diversi core
- Altri livelli, come la RAM, sono **condivisi** tra i core

Ad esempio, questa è una parte della gerarchia della memoria in alcuni processori:



- Questa stratificazione crea notevoli problemi di **sincronizzazione** tra processori (o *core*) diversi
  - una famiglia di problemi di questo tipo prende il nome di “coerenza delle cache” (*cache coherence*)

- Il memory model presenta un **modello astratto del comportamento della memoria**, in modo da consentire agli utenti (in primo luogo, sviluppatori di compilatori, di sistemi operativi, etc.) di ragionare senza conoscere i dettagli dell'hardware

- Ad esempio, il memory model specifica le regole in base alle quali una scrittura in memoria da parte di un core diventa visibile ad un altro core

Introduzione al Java Memory Model

- Il Java Memory Model (JMM) è una descrizione di come la JVM gestisce l'accesso alla memoria
- E' di particolare importanza in un contesto multi-threaded

- Contiene regole di tre tipi:
  - **Atomicità** Quali operazioni sono naturalmente atomiche?
  - **Visibilità** Quand'è che una scrittura in memoria diventa visibile agli altri thread?
  - **Ordinamento** In quale ordine vengono effettuate le operazioni?

Introduzione al Java Memory Model

- Il JMM è un ingrediente fondamentale della **portabilità** promessa da Java
- Difatti, il JMM offre alle applicazioni (e al programmatore) delle regole certe sull'accesso concorrente alla memoria
- Il compilatore e la JVM hanno il compito di conciliare il JMM con il memory model dell'architettura hardware sottostante

- Il JMM è stato ridefinito nel 2004, perché la versione precedente aveva problemi di vario tipo

Regole di **Atomicità**

Il modificatore ***volatile***

- ***volatile*** è un modificatore che si può applicare esclusivamente ai **campi** di una classe
- Intuitivamente, *volatile* indica che quel campo **può essere modificato da più thread**
- Tecnicamente, *volatile* ha conseguenze di **atomicità**, **visibilità**, e **ordinamento**, come illustrato nelle slide seguenti
- Un campo *volatile* non può essere *final*, in quanto l'accoppiata è priva di senso

Definizione:

Un'operazione è **atomica** se, dal punto di vista di *qualsiasi altro thread*, i suoi effetti vengono visti per intero, o per niente (ma mai "a metà")

Quali operazioni sono naturalmente atomiche?

(ovvero, sono atomiche anche in assenza di meccanismi di mutua esclusione)

Regole:

1) La lettura e la scrittura di variabili di tipo primitivo, esclusi i tipi *long* e *double*, e di tipo riferimento sono operazioni atomiche
2) La lettura e la scrittura di variabili ***volatili*** sono operazioni atomiche

Commenti:

- La modifica di una variabile *long* può avvenire in due distinte operazioni, che modificano separatamente i 32 bit più significativi e i 32 bit meno significativi
- Queste due operazioni potrebbero essere interrotte dallo scheduler

Esempi di atomicità

Date le seguenti variabili:

```
int x, y;
long n;
volatile long m;
```

Esaminiamo le seguenti assegnazioni:

1) x = 8; Operazione **atomica**.

2) x = y; Operazione **non atomica**, perché comprende una lettura e una scrittura.
Se l'operazione viene interrotta, un altro thread può modificare y
ed x potrebbe comunque assumere il *vecchio* valore di y.

3) n = 0x1122334455667788; (costante long espressa in esadecimale)
Operazione **non atomica**.
Se l'operazione viene interrotta, un thread potrebbe osservare
n == 0x1122334400000000 e un altro thread
n == 0x0000000055667788.

4) m = 0x1122334455667788; Operazione **atomica**.

5) m++; Operazione **non atomica**, perché comprende una lettura e una scrittura.

Esempi di atomicità

Aggiungiamo le seguenti variabili:

```
Object a, b;
volatile Object c, d;
```

6) a = null; Operazione **atomica**

7) a = b; Operazione **non atomica**, perché comprende una lettura e una scrittura. Stesso rischio del caso 2.

8) c = d; Operazione **non atomica**. Stesso rischio dei casi 2 e 7.

Mutua atomicità

Definizione:

Due operazioni (o blocchi di istruzioni) A e B sono **mutuamente atomiche** se, dal punto di vista di un thread che sta eseguendo A, gli effetti dell'esecuzione di B da parte di un altro thread vengono visti per intero, o per niente (ma mai "a metà").

Questa è una forma più debole di atomicità, **relativa** anziché assoluta

La mutua atomicità è una forma più astratta di mutua esclusione:

- Il termine *mutua esclusione* suggerisce che le due operazioni verranno eseguite in tempi diversi (non contemporaneamente)
- Il termine *mutua atomicità* parla solo di visibilità degli effetti, non di implementazione

La mutua esclusione è *un modo di implementare la mutua atomicità*

La mutua atomicità viene garantita dai meccanismi di locking, come il costrutto synchronized

Tutti i blocchi synchronized sullo stesso monitor sono mutuamente atomici

Regole di **Visibilità**

Esempio 1: un problema di visibilità

Quali sono gli output possibili di questo programma?

```
static boolean done;
static int n;

public static void main(String args[])
{
     Thread t = new Thread() {
        public void run() {
           n = 42;
           try { sleep(1000); }
           catch (InterruptedException e) { return; }
           done = true;
           System.out.println("Fatto");
        }
     };
     t.start();

     while (!done) { }
     System.out.println(n);
}
```

Esempio 1: un problema di visibilità

Il programma della slide precedente può essere schematizzato come segue:

```
done = false
n = 0
```

Thread 1 (main):	**Thread 2:**
`while (!done) { };`	`n = 42;`
`System.out.println(n);`	`sleep(1000);`
	`done = true;`
	`System.out.println(“Fatto”);`

Si noti che i due thread condividono le variabili *n* e *done*, ma non usano alcun meccanismo di sincronizzazione.

Sorprendentemente, il ciclo *while* del thread 1 può comportarsi come un ciclo **infinito** (provare per credere), anche se il thread 2 dopo un'attesa di un secondo esegue *done* = *true*.

Questo perché, in mancanza di sincronizzazione, il JMM non offre alcuna garanzia su quando la scrittura nella variabile *done* effettuata dal thread 2 sarà **visibile** al thread 1.

Questo punto verrà illustrato nelle slide successive.

Regole di visibilità

I principi fondamentali della visibilità inter-thread:

- In mancanza di sincronizzazione, le operazioni (scritture in memoria) svolte da un thread possono rimanere nascoste agli altri thread **a tempo indefinito**
- In particolare, alcune operazioni possono rimanere nascoste ed altre essere visibili

La visibilità è garantita solamente dalle seguenti operazioni:

1) Acquisire un **monitor** (cioè, entrare in un metodo o blocco sincronizzato) **rende visibili** le operazioni effettuate dall'ultimo thread che possedeva quel monitor, fino al momento in cui l'ha rilasciato
2) Leggere il valore di una **variabile *volatile*** **rende visibili** le operazioni effettuate dall'ultimo thread che ha modificato quella variabile, fino al momento in cui l'ha modificata
3) Invocare ***t.start()*** **rende visibili** al nuovo thread t tutte le operazioni effettuate dal thread chiamante, fino all'invocazione a start
4) Ritornare da una invocazione ***t.join()*** **rende visibili** tutte le operazioni effettuate dal thread *t* fino alla sua terminazione

Esempio 1 rivisitato

Nell'esempio visto in precedenza, supponiamo di dichiarare la variabile done *volatile*:

```
static volatile boolean done;
static int n;

public static void main(String args[])
{
     Thread t = new Thread() {
        public void run() {
           n = 42;
           try { sleep(1000); }
           catch (InterruptedException e) { return; }
           done = true;
           System.out.println("Fatto");
        }
     };
     t.start();

     while (!done) { }
     System.out.println(n);
}
```

Ora il programma avrà il comportamento atteso, perché ogni lettura della variabile done effettuata dal thread principale rende visibili le modifiche a done fatte dall'altro thread (regola 2)

Un confronto tra *synchronized* e *volatile*

- Come si è visto, sia *synchronized* sia *volatile* offrono garanzie di **atomicità** e di **visibilità**

- Tuttavia, il modificatore *volatile* rende atomica soltanto una singola scrittura nella variabile in questione
- Come si è visto, il modificatore *volatile* **non** rende atomica nemmeno un'assegnazione del tipo "a = b", anche se a e b fossero entrambe *volatile*
- Come ulteriore esempio, il modificatore *volatile* **non** rende atomica l'espressione "n++"

- Quindi, un blocco o metodo *synchronized* rappresenta l'unica opzione per rendere **mutuamente atomica** una sequenza di istruzioni (più precisamente, per renderla mutuamente atomica rispetto ad altre sequenze critiche, sincronizzate sullo stesso monitor)

- Il modificatore *volatile* è indicato nei casi in cui il contesto richieda la **visibilità** dei cambiamenti, ma **non la mutua atomicità**

Regole di **Ordinamento**

Esempio 2: un problema di ordinamento

Si considerino i seguenti thread, che condividono due variabili A e B, inizialmente poste a **0**

Thread 1:	**Thread 2:**
`int r1;`	`int r2;`
`r1 = B;`	`r2 = A;`
`A = 1;`	`B = 1;`

Che valori possono assumere alla fine le variabili r1 ed r2?

Esempio 2: un problema di ordinamento

Si considerino i seguenti thread, che condividono due variabili A e B, inizialmente poste a **0**

Thread 1:	**Thread 2:**
`int r1;`	`int r2;`
`r1 = B;`	`r2 = A;`
`A = 1;`	`B = 1;`

Che valori possono assumere alla fine le variabili r1 ed r2?

r1 = 0, r2 = 1	se il Thread 1 viene eseguito per primo
r1 = 1, r2 = 0	se il Thread 2 viene eseguito per primo
r1 = 0, r2 = 0	se lo scheduler interrompe il primo thread tra le due assegnazioni

Sorprendentemente, il JMM consente anche questo risultato, apparentemente assurdo:
r1 = 1, r2 = 1

Difatti, in mancanza di sincronizzazione, al compilatore/JVM/CPU è consentito di **riordinare le istruzion**i, a patto che tale riordino sia ininfluente *dal punto di vista del singolo thread*.

In questo caso, è consentito invertire l'ordine delle due istruzioni del Thread 1 (oppure del Thread 2). Se l'ordine viene invertito, il risultato r1 = 1 ed r2 = 1 diventa possibile.

Regole di ordinamento

In generale, il compilatore e la JVM possono riordinare **qualsiasi sequenza di istruzioni** a patto che il risultato non cambi per un singolo thread che esegua quelle istruzioni.

I costrutti *synchronized* e *volatile* riducono le possibilità di riordino.

Consideriamo due istruzioni successive:

x

y

e supponiamo che **non abbiano dipendenze** dal punto di vista di un thread singolo.
In quali casi x e y possono essere eseguite in ordine inverso?

Per rispondere a questa domanda, dobbiamo distinguere tre categorie di istruzioni:

1) Letture di una variabile volatile, oppure inizio di un blocco o metodo sincronizzato
2) Scritture di una variabile volatile, oppure fine di un blocco o metodo sincronizzato
3) Tutte le altre (istruzioni "normali")

Regole di ordinamento

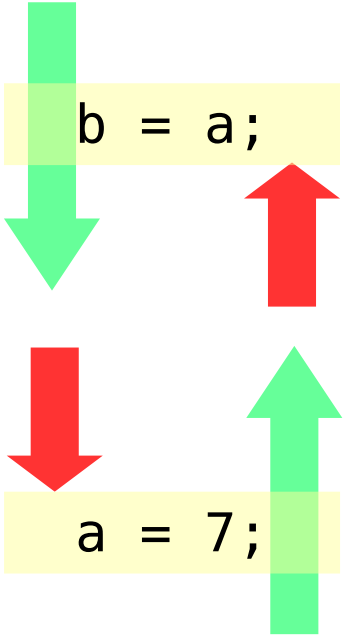
La seguente tabella specifica se è possibile **scambiare** di posto le istruzioni x e y:

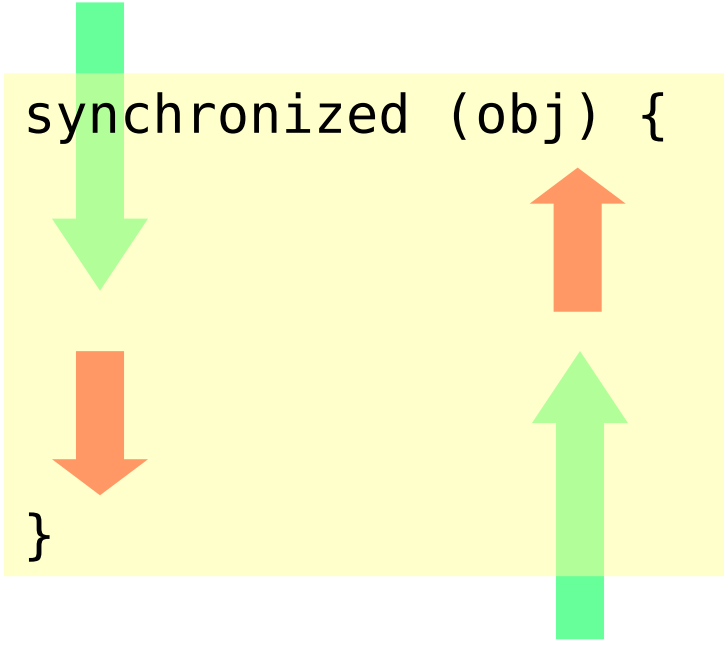
tipo di x \ tipo di y	normale	lettura volatile inizio synchronized	scrittura volatile fine synchronized
normale	**Si**	**Si**	No
lettura volatile inizio synchronized	No	No	No
scrittura volatile fine synchronized	**Si**	No	No

In pratica:

1) Le istruzioni normali si possono sempre scambiare
2) Le istruzioni normali si possono portare *dentro* un blocco sincronizzato
3) Le istruzioni normali che precedono la lettura di una volatile si possono spostare dopo la lettura
4) Le istruzioni normali che seguono la scrittura di una volatile si possono spostare prima della scrittura

```
volatile int a;
```





Esempio 2 con blocchi sincronizzati

Modifichiamo l'esempio 2, aggiungendo un oggetto condiviso “obj” e due blocchi sincronizzati:

Thread 1:	**Thread 2:**

`int r1;`
`synchronized (obj) {`
`  r1 = B;`
`  A = 1;`
`}`
	`int r2;`
	`synchronized (obj) {`
	`  r2 = A;`
	`  B = 1;`
	`}`

Quali valori sono possibili adesso per r1 e r2?

Esempio 2 con blocchi sincronizzati

Modifichiamo l'esempio 2, aggiungendo un oggetto condiviso "obj" e due blocchi sincronizzati:

Thread 1:	**Thread 2:**

`int r1;`
`synchronized (obj) {`
`  r1 = B;`
`  A = 1;`
`}`
	`int r2;`
	`synchronized (obj) {`
	`  r2 = A;`
	`  B = 1;`
	`}`

I blocchi sincronizzati **non** impediscono alle istruzioni al loro interno di essere riordinate

In compenso, rendono i due blocchi **mutuamente atomici**

Quindi, uno dei due blocchi verrà interamente eseguito prima dell'altro

Gli unici output possibili sono:
r1 = 0, r2 = 1
r1 = 1, r2 = 0

Esempio 2 con volatile

In alternativa, supponiamo che A e B siano ***volatile***

Thread 1:	**Thread 2:**
`int r1;`	`int r2;`
`r1 = B;`	`r2 = A;`
`A = 1;`	`B = 1;`

Consultando le regole di ordinamento, scopriamo che il compilatore adesso **non può riordinare** le istruzioni, perché sono tutte letture o scritture di variabili *volatile*

Gli output possibili sono:

r1 = 0, r2 = 0

r1 = 0, r2 = 1

r1 = 1, r2 = 0

Il primo output può capitare perché volatile **non** rende i due thread mutuamente esclusivi

Esercizio 1

(SimpleThread, 2015-6-24)

Indicare tutti gli output possibili di un programma che faccia partire contemporaneamente due istanze della seguente classe SimpleThread.

```
public class SimpleThread extends Thread {
    private static volatile int n = 0;

    public void run() {
1       n++;
2       int m = n;
3       System.out.println(m);
    }
}
```

Esercizio 2

Dati i seguenti campi:

```
volatile Integer n = 0;
Integer m = 0;
```

Quali **scambi tra righe consecutive** sono consentiti dalle regole del JMM nel seguente frammento di codice Java?

```
1  System.out.println(n);
2  synchronized (m) {
3     m = 10;
4     n = m;
5  }
6  m = n;
```

Applicazione:
Lazy Inizialization

Esempio: la *lazy initialization*

Consideriamo il problema di una classe che voglia rendere disponibile un oggetto, che sarà istanziato soltanto alla prima richiesta

Questo problema prende il nome di *lazy initialization* (inizializzazione pigra)

La soluzione naif è la seguente:

```
class A {
     private static HeavyClass special;

     public static HeavyClass getSpecial() {
          if (special == null)
               special = new HeavyClass();
          return special;
     }
}
```

Il riferimento special sarà inizializzato con un nuovo oggetto alla prima invocazione di getSpecial.

Purtroppo, questa implementazione non è *thread-safe*, come illustrato nella prossima slide.

Il primo rischio della *lazy initialization*

L'implementazione della slide precedente soffre di **due** problemi diversi.

Scenario 1:

Due thread invocano contemporaneamente *getSpecial:*

- Il primo thread trova special a *null* e viene interrotto dallo scheduler. Anche il secondo thread trova special a *null*, quindi istanzia un oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a *special* ed esce da *getSpecial*.
- Il primo thread riprende la sua esecuzione, istanzia un **secondo oggetto** di tipo *HeavyClass*, ne assegna l'indirizzo a *special* ed esce anch'esso da *getSpecial*.

Problema: sono stati istanziati **due** diversi oggetti HeavyClass, contrariamente alle intenzioni.

Commenti: questo è il classico problema dovuto alla mancata atomicità della sequenza “lettura di special – scrittura di special”. Il problema non è legato alle sottigliezze del JMM.

Il secondo rischio della *lazy initialization*

Scenario 2:

Due thread invocano contemporaneamente *getSpecial:*

- Il primo thread trova special a *null*, istanzia un nuovo oggetto di tipo *HeavyClass* e ne assegna l'indirizzo a *special*.
- Il secondo thread riceve da *getSpecial* un riferimento allo stesso oggetto, accede a questo oggetto e lo trova in uno **stato incoerente**, cioè non completamente inizializzato dal costruttore.

Problema: Il secondo thread potrebbe **vedere** l'oggetto a metà della sua costruzione, anche se dal punto di vista del primo thread l'oggetto è stato completamente costruito.

Commenti: In mancanza di sincronizzazione, non vi è alcuna garanzia che il secondo thread veda tutte le operazioni svolte dal primo. Potrebbe darsi che il secondo thread veda il valore corrente di special (cioè, l'indirizzo del nuovo oggetto HeavyClass), ma non veda il valore corrente di alcuni campi di quell'oggetto.

Questo problema è separato dal primo ed è legato alle regole di **visibilità** del JMM.

Soluzione semplice al problema della *lazy initialization*

Dichiarare **sincronizzato** (synchronized) il metodo **getSpecial** risolve **entrambi** i problemi descritti.

Infatti, il **primo** problema viene risolto rendendo mutuamente esclusive le invocazioni a getSpecial.

Il **secondo** problema viene risolto grazie alle garanzie di visibilità offerte da synchronized (regola di visibilità n.1).

Questa soluzione impone però un **overhead di performance**, dovuto alla necessità di acquisire e rilasciare un mutex, su **tutte** le invocazioni di getSpecial, anche molto tempo dopo l'inizializzazione dell'oggetto HeavyClass, quando ormai la sincronizzazione non sarebbe più necessaria.

La prossima slide presenta una soluzione più avanzata.

Un’operazione atomica

Java garantisce che **l'inizializzazione di una classe sia un'operazione atomica**.

L’inizializzazione consiste nell’esecuzione di tutti i *blocchi di inizializzazione statici* e di tutti gli *inizializzatori di campi statici*

```
class A {
     private static B b = <inizializzatore>;

     static {
          <blocco di inizializzazione statico>
     }
}
```

L’inizializzazione avviene subito dopo il caricamento (loading) di quella classe

Si ricordi che il caricamento avviene *dinamicamente*, la prima volta che il programma usa quella classe (e non necessariamente all’avvio del programma)

Soluzione avanzata al problema della *lazy initialization*

Consideriamo la seguente struttura:

```
class A {
     private static class HeavyClassHolder {
          static HeavyClass special = new HeavyClass();
     }
     public static HeavyClass getSpecial() {
          return HeavyClassHolder.special;
     }
}
```

Il riferimento *special* viene spostato all'interno di una classe interna statica e privata.

La classe *HeavyClass* sarà instanziata quando il campo statico *special* verrà inizializzato.
La VM inizializza la classe *HeavyClassHolder*, e quindi il suo campo *special*, **soltanto al primo utilizzo**, cioè alla prima invocazione di *getSpecial*. (si veda la sezione 12.4.1 del JLS)

Non ci sono problemi di sincronizzazione perché la VM garantisce che **l'inizializzazione di una classe sia un'operazione atomica**.

Quindi, questa soluzione (proposta in [3]) risolve il problema della lazy initialization, senza utilizzare sincronizzazione (esplicita).

Bibliografia

1) Java Concurrency in Practice
di Goetz, Peierls, Bloch, Bowbeer, Holmes e Lea
Addison-Wesley

2) Il **Java Memory Model**
Disponibile in rete come *Java Specification Request (JSR) 133*, oppure come capitolo 17 del *Java Language Specification* (definizione del linguaggio Java)

3) Effective Java: a Programming Language Guide
di Joshua Bloch
Addison-Wesley

4) The JSR-133 Cookbook for Compiler Writers
pagina web curata da Doug Lea