Reti e Laboratorio III
Modulo Laboratorio III
AA. 2022-2023

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 6
Stream Sockets for servers
Volatile, Atomic
20/10/2022

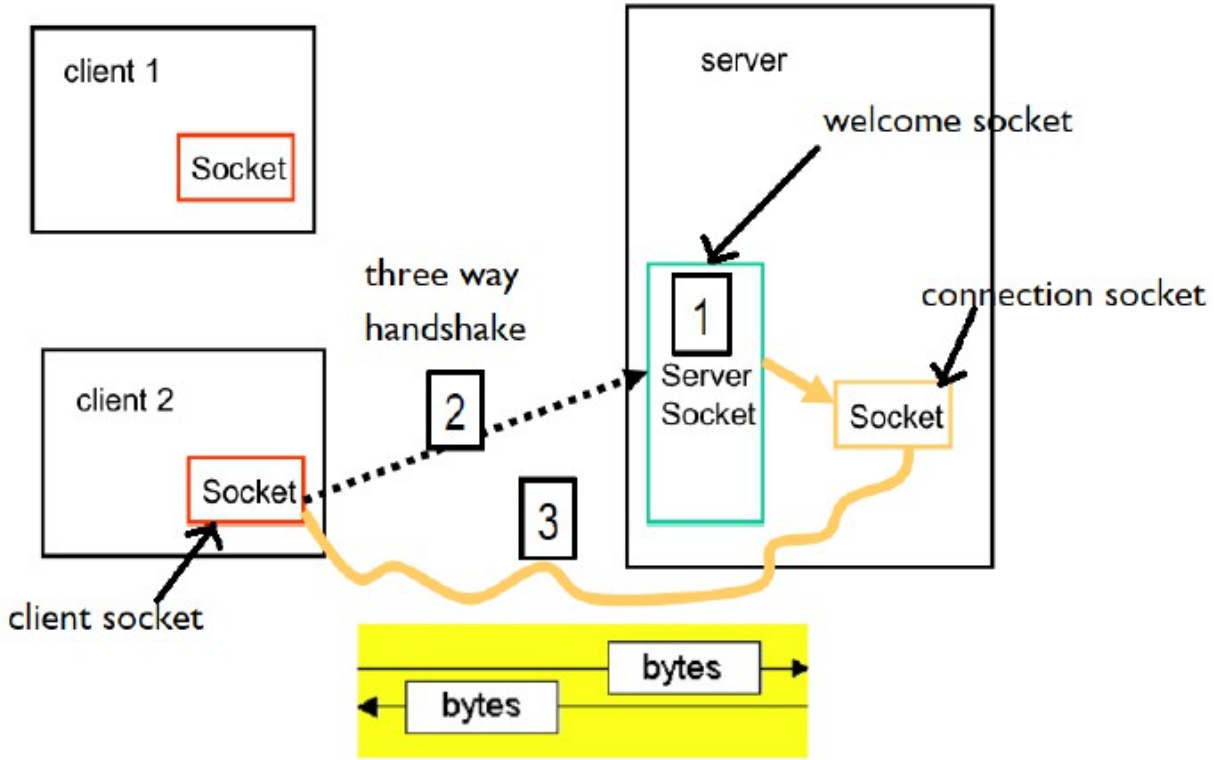
SOCKET LATO SERVER

- esistono due tipi di socket TCP, lato server:
  - welcome (passive, listening) sockets: utilizzati dal server per accettare le richieste di connessione
  - connection (active) sockets: connettono il server ad un particolare client e supportano lo streaming di byte tra di essi
- il client crea un active socket per richiedere la connessione
- il server accetta una richiesta di connessione sul welcome socket
  - crea un proprio connection socket che rappresenta il punto terminale della sua connessione con il client
  - la comunicazione vera e propria avviene mediante la coppia di active socket presenti nel client e nel server

SOCKET LATO SERVER

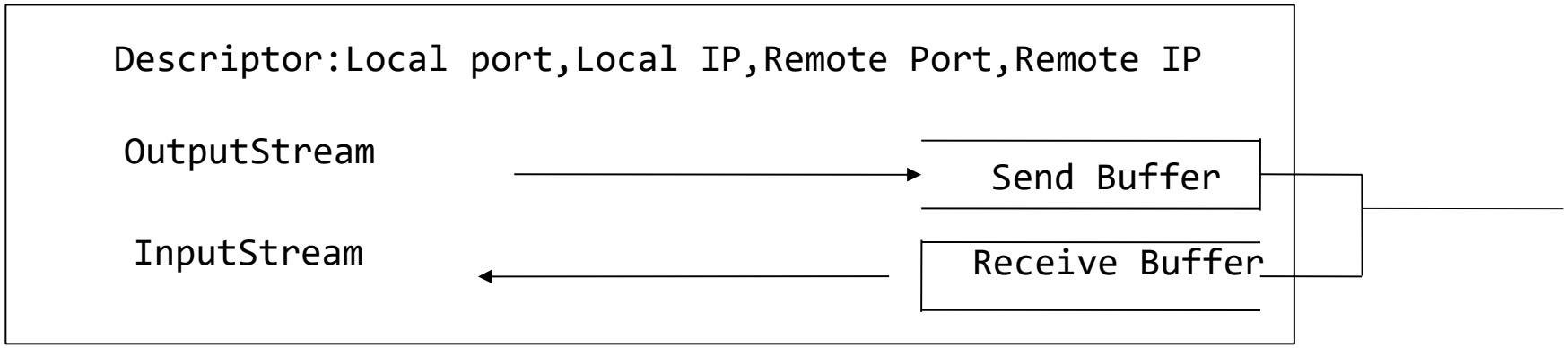
- il server pubblica un proprio servizio
  - gli associa un welcome socket, sulla porta remota PS, all'indirizzo IPS
  - usa un oggetto di tipo `ServerSocket`
- il client crea un `Socket` e lo connette all'endpoint IPS + PS
- la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server
  - three way handshake completamente gestito dal supporto
  - se la richiesta viene accettata,
    - il server crea un socket dedicato per l'interazione con quel client
    - tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

SOCKET LATO SERVER



STREAM BASED COMMUNICATION

- dopo che la richiesta di connessione viene accettata, client e server associano streams di bytes di input/output ai socket dedicati a quella connnessione, poichè gli stream sono unidirezionali
  - a seconda del servizio può essere necessario un solo stream di output dal server verso il client, oppure una coppia di stream da/verso il client
- la comunicazione avviene mediante lettura/scrittura di dati sullo stream
- eventuale utilizzo di filtri associati agli stream



Struttura del Socket TCP

JAVA STREAM SOCKET API: LATO SERVER

`java.net.ServerSocket:` costruttori

```
public ServerSocket(int port)throws BindException, IOException
public ServerSocket(int port,int length) throws BindException,
                                                IOException
```

- costruisce un listening socket, associandolo alla porta `port.`
- `length`: lunghezza della coda in cui vengono memorizzate le richieste di connessione.

  se la coda è piena, ulteriori richieste di connessione sono rifiutate

```
public ServerSocket(int port,int length,Inetaddress bindAddress)....
```

- permette di collegare il socket ad uno specifico indirizzo `IP` locale.
- utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi `IP`, uno visibile da Internet, l'altro visibile solo a livello di rete locale
- se voglio servire solo le richieste in arrivo dalla rete locale, associo il connection socket all'indirizzo `IP` locale

JAVA STREAM SOCKET API: LATO SERVER

- accettare una nuova connessione dal connection socket

```
public Socket accept( ) throws IOException
```

metodo della classe **ServerSocket**.

- quando il processo server invoca il metodo `accept()`, pone il server in attesa di nuove connessioni.
- bloccante: se non ci sono richieste, il server si blocca (possibile utilizzo di time-outs)
- quando c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket tramite cui avviene la comunicazione effettiva tra cliente server

PORT SCANNER LATO SERVER

- ricerca dei servizi attivi sull'host locale

```
import java.net.*;
public class LocalPortScanner {
    public static void main(String args[])
    {for (int port= 1; port<= 1024; port++)
        try   {ServerSocket server = new ServerSocket(port);}
        catch (BindException ex)
               {System.out.println(port + "occupata");}

        catch (Exception ex) {System.out.println(ex);}
         } }
```

CICLO DI VITA TIPICO DI UN SERVER

```
// instantiate the ServerSocket
ServerSocket servSock = new ServerSocket(port);
while (! done) // oppure while(true) {
     // accept the incoming connection
     Socket sock = servSock.accept();
    // ServerSocket is connected ... talk via sock
     InputStream in = sock.getInputStream();
     OutputStream out = sock.getOutputStream();
     //client and server communicate via in and out and do their wotk
      sock.close();
              }
servSock.close();
```

DAYTIME SERVER

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
public final static int PORT = 1313;
  public static void main(String[] args) {
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try (Socket connection = server.accept()) {
         Writer out = new OutputStreamWriter(connection.getOutputStream());
         Date now = new Date();
         out.write(now.toString() +"\r\n");
         out.flush();
        // connection.close();
       } catch (IOException ex) {}
     }
   } catch (IOException ex) {
     System.err.println(ex); } } }
```

porte 0-1023 privilegiate

si ferma qui ed aspetta, quando un client si connette restituisce un nuovo Socket

servizio della richiesta

inutile perchè si è usato il try with resources

try-with-resource: autoclose

DAYTIME SERVER: CONNETTERSI CON TELNET

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
 public final static int PORT = 13;
 public static void main(String[] args) {
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try (Socket connection = server.accept()) {
         Writer out = new OutputStreamWriter(connection.getOutputStream());
         Date now = new Date();
         out.write(now.toString() +"\r\n");
         out.flush();
         connection.close();
       } catch (IOException ex) {}
     }  } catch (IOException ex) {System.err.println(ex);}}}
```

```
$ telnet localhost 1333
trying 127.0.0.1....
connected to localhost
San Oct 17 23:16:12 CEST 2021
```

MULTITHREADED SERVER

- nello schema del lucido precedente, la fase "communicate and work" può essere eseguita in modo concorrente da più threads
- un thread per ogni client, gestisce le interazioni con quel particolare client
- il server può gestire le richieste in modo più efficiente
- tuttavia.....threads: anche se processi lightweigth ma tuttavia utilizzano risorse !
  - esempio: un thread che utilizza 1MB di RAM. 1000 thread simultanei possono causare problemi !
- Soluzioni alternative:
  - Thread Pooling
  - `ServerSocketChannels` di NIO

A CAPITALIZER SERVICE: SERVER

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.concurrent.*;
public static void main(String[] args) throws Exception {
        try (ServerSocket listener = new ServerSocket(10000)) {
            System.out.println("The capitalization server is running...");
            ExecutorService pool = Executors.newFixedThreadPool(20);
            while (true) {
                pool.execute(new Capitalizer(listener.accept()));
            }
        }
    }
```

A CAPITALIZER SERVICE: SERVER

```
private static class Capitalizer implements Runnable {
     private Socket socket;
     Capitalizer(Socket socket) {
        this.socket = socket; }
     public void run() {
        System.out.println("Connected: " + socket);
        try (Scanner in = new Scanner(socket.getInputStream());
             PrintWriter out = new PrintWriter(socket.getOutputStream(),
                                                                  true))
            { while (in.hasNextLine()) {
                    out.println(in.nextLine().toUpperCase()); }
            } catch (Exception e) { System.out.println("Error:" + socket); }
           }
      }
     }
```

A CAPITALIZER SERVICE: CLIENT

```
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
public class CapitalizeClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command line
                                argument");
            return;
        }
        Scanner scanner=null;
        Scanner in=null;
```

A CAPITALIZER SERVICE: CLIENT

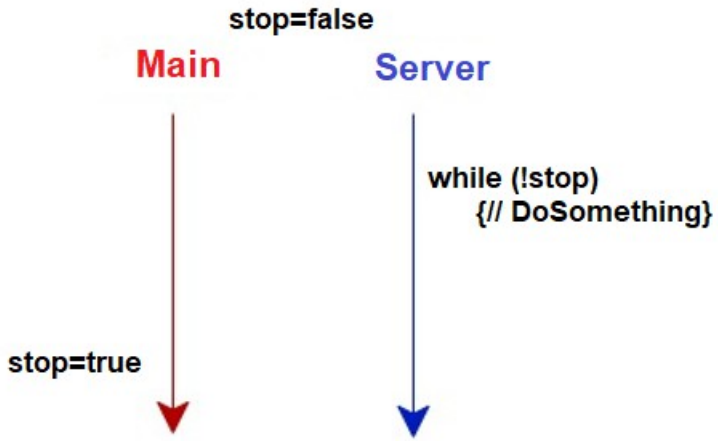
```
try (Socket socket = new Socket(args[0], 10000)) {
            System.out.println("Enter lines of text then EXIT to quit");
            scanner = new Scanner(System.in);
            in = new Scanner(socket.getInputStream());
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                                                                  true);
            boolean end=false;
            while (!end) {
                   { String line= scanner.nextLine();
                    if (line.contentEquals("exit")) end=true;
                    out.println(line);
                    System.out.println(in.nextLine());}
            }}
        finally {scanner.close(); in.close();}
        }
    }
```

ANCORA SUL MULTITHREADING

- Variabili `volatile`
- Variabili `Atomic`

PERCHE' VOLATILE?

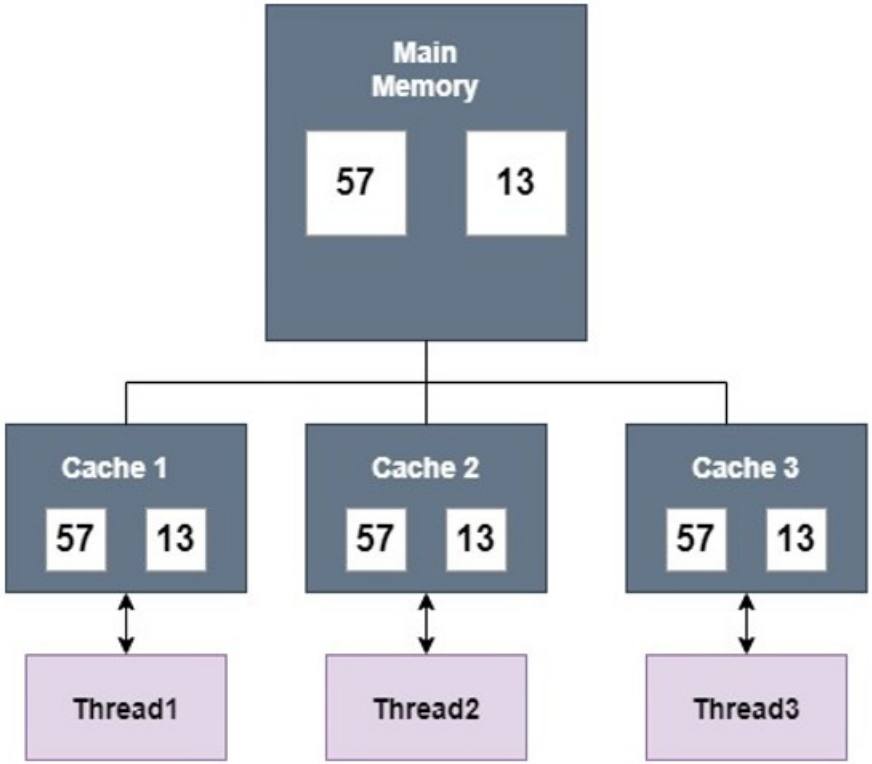
```
public class Server extends Thread
{ boolean stop = false; int i;
  public void run()
   { while(! stop) {};
     System.out.println("Server is stopped....");}
  public void stopThread()
    { stop = true;}}
```




```
public class StoppingAThread
{ public static void main(String args[]) throws InterruptedException
{ Server myServer = new Server();
  myServer.start();
  System.out.println(Thread.currentThread().getName() + " is stopping Server thread");
  Thread.sleep(1000);
  myServer.stopThread();
  System.out.println(Thread.currentThread().getName() + " is finished now"); }}
```

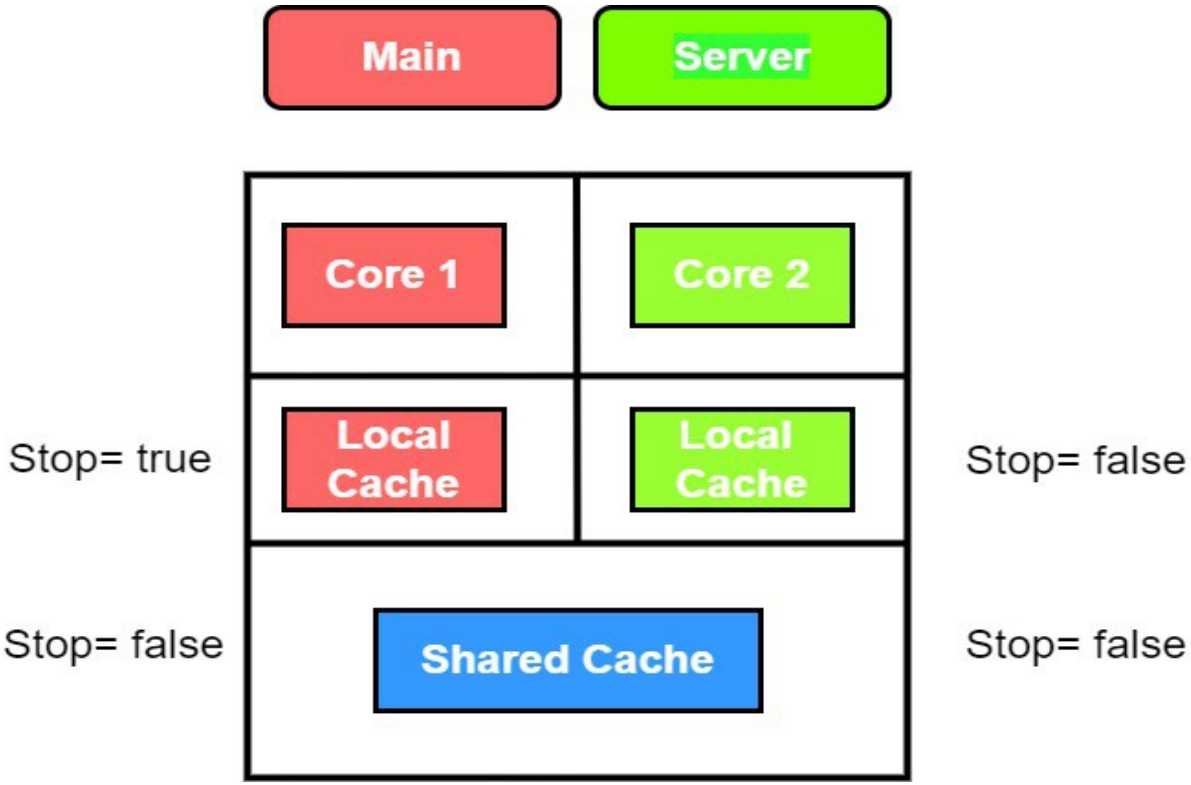
il programma non termina!

IL PROBLEMA DELLA VISIBILITA'



architettura di riferimento, utile per capire il problema della visibilità

IL PROBLEMA DELLA VISIBILITA'



- quando il `Main` aggiorna `Stop`, è possibile che la modifica non sia riportata nella memoria condivisa
- il `Main` aggiorna la variabile `Stop` nella propria cache, ma la modifica non viene riportata nella memoria condivisa

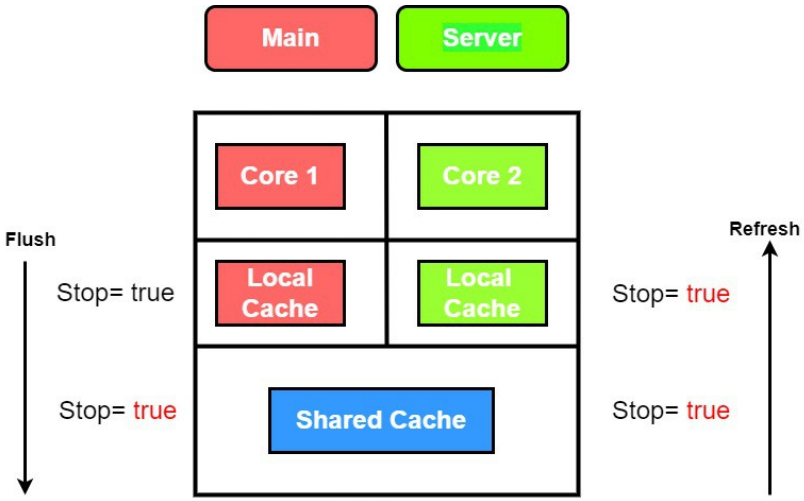
IL MODIFICATORE VOLATILE

- il problema riguarda la "visibilità" della modifica, non la sincronizzazione: read e write di un booleano sono atomiche
- modifichiamo la dichiarazione con la keyword `volatile`

```
volatile boolean stop = false
```

  - l'aggiornamento ad una variabile `volatile` è sempre effettuato nella main memory
    - flush della cache
  - il valore della variabile `volatile` è sempre letto dalla memoria

Text



VOLATILE: VISIBILITA' DI SCRITTURE

- tutte le scritture su una variabile volatile sono riportate direttamente nella memoria condivisa
- inoltre, tutte la variabili visibili dal thread che sta eseguendo la modifica vengono anche sincronizzate sulla memoria condivisa
- esempio:

```
this.nonVolatileVarA = 34;
this.nonVolatileVarB = new String("Text");
this.volatileVarC    = 300;
```

- quando viene eseguita la terza istruzione, sulla variabile `volatileC`, i valori delle due variabili non-volatile vengono sincronizzati in memoria condivisa

VOLATILE: VISIBILITA' DI LETTURE

- quando viene letto il valore di una variabile volatile, viene garantito che tale valore venga letto direttamente dalla memoria condivisa
- inoltre, viene fatto il refresh di tutte la variabili visibili dal thread che sta eseguendo la lettura
- esempio:

```
c = other.volatileVarC;
b = other.nonVolatileB;
a = other.nonVolatileA;
```

- la prima istruziine è la lettura di una variabile volatile. Quando questa variabile viene letta dalla memoria, viene effettuato il refresh anche delle due altre variabili

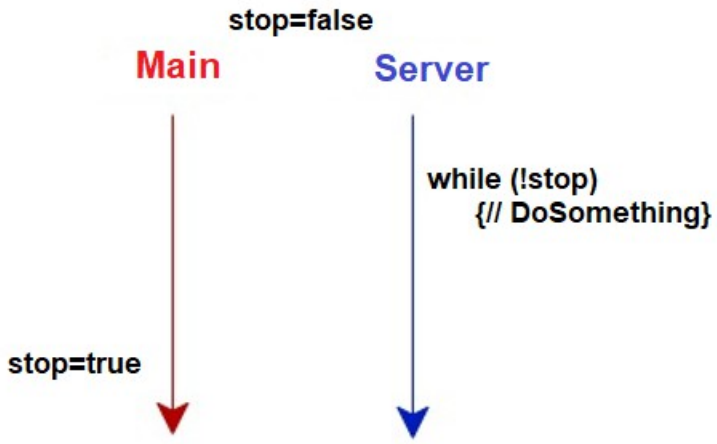
UNA SOLUZIONE ALTERNATIVA

```
public class Server extends Thread
 { Boolean stop = false; int i;
  public void run()
   { synchronized(stop) {};
     while(! stop)
      {synchronized(stop) {}};
  System.out.println("Server is stopped....");}
  public synchronized void stopThread(){ stop = true; }}

public class StoppingAThread
  {….}
```

sincronizzarsi sulla variabile stop ha lo stesso effetto di usare il modificatore volatile

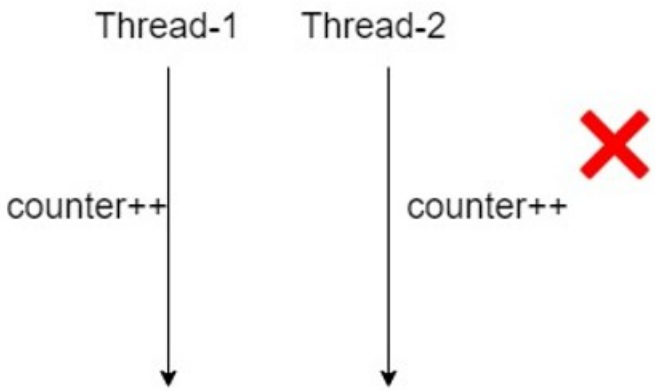
la variabile deve essere definita `Boolean`, per poter acquisire la lock



SINCRONIZZAZIONE: VISIBILITA'

- blocchi e metodi sincronizzati forniscono garanzia di visibilità simile a quella offerta dal modificatore `volatile`
- quando un thread entra in un metodo o blocco sincronizzato, viene effettuato un refresh di tutte le variabili visibili dal thread
- quando un thread esce da un blocco sincronizzato, tutte le variabili visibili dal thread vengono scritte in memoria
- monitor garantisce sia sincronizzazione che visibilità
- quando usare volatile?
  - quando la variabile condivisa è di tipo semplice
  - per acquisire la lock occorrerebbe fare il cast al corrispondente oggetto
  - tipico del pattern “termina l'esecuzioen di un thread”

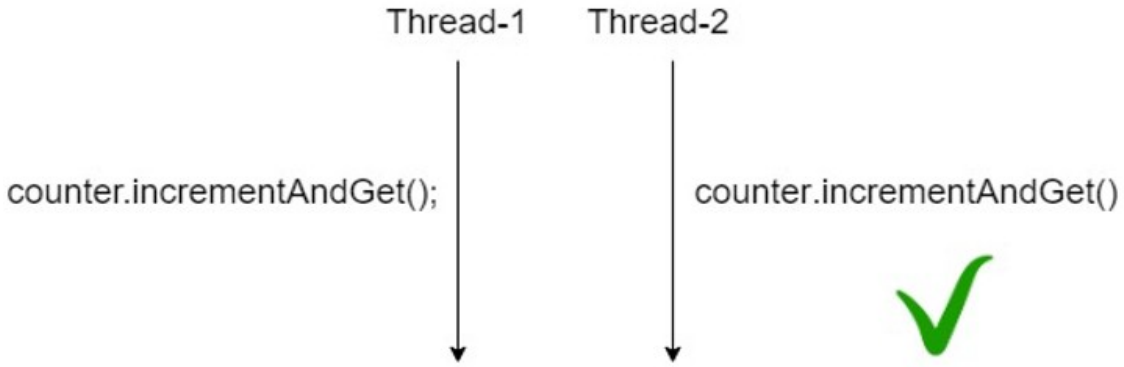
SINCRONIZZAZIONE SU VARIABILI



Thread-1	Thread-2
Read value (=1)	
	Read value (=1)
Add 1 and write (=2)	
	Add 1 and write (=2)

- l'incremento di una variabile (volatile o meno) non è atomico
- se più thread provano ad incrementare una variabile in modo concorrente, un aggiornamento può andare perduto (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- soluzione alternativa: usare le variabili `Atomic`

ATOMIC VARIABLES



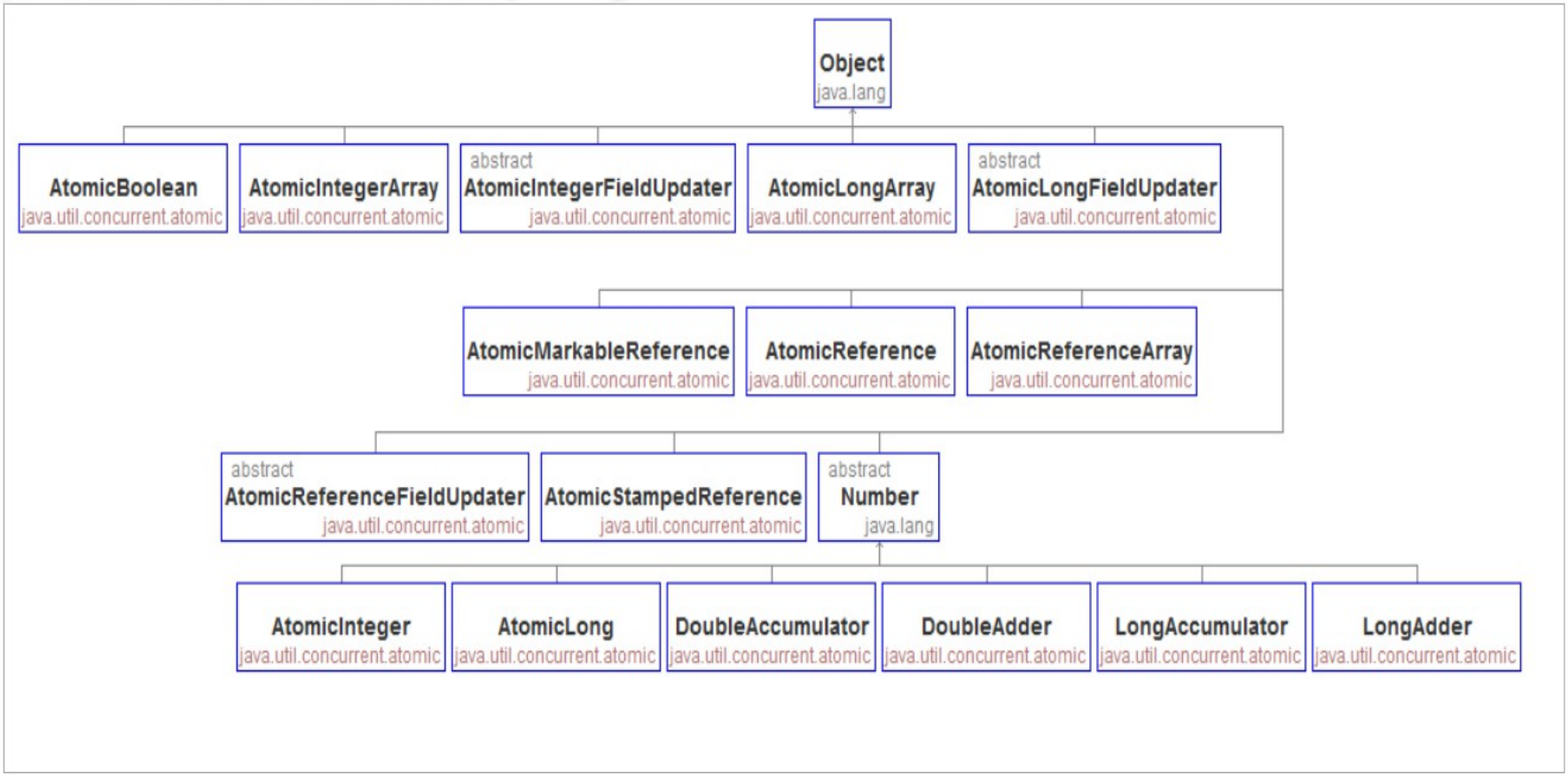

```
AtomicInteger value = new AtomicInteger(1);
```

- operazioni atomiche che non richiedono sincronizzazioni esplicite o lock: è la JVM che garantisce la atomicità
  - `incrementAndGet()`: atomically increments by one
  - `decrementAndGet()`: atomically decrements by one
  - `compareAndSet(int expectedValue, int newValue)`
- molte altre classi
  - `AtomicLong`
  - `AtomicBoolean`
  - …

ATOMIC VARIABLES: UN ESEMPIO

```
import java.util.concurrent.*; import java.util.concurrent.atomic.*;
public class AtomicIntExample {
public static void main(String[] args) {
   ExecutorService executor = Executors.newFixedThreadPool(2);
   AtomicInteger atomicInt = new AtomicInteger();
   for(int i = 0; i < 10; i++){
        CounterRunnable runnableTask = new  CounterRunnable(atomicInt);
        executor.submit(runnableTask);
    }
    executor.shutdown(); }
 class CounterRunnable implements Runnable {
    AtomicInteger atomicInt;
    CounterRunnable(AtomicInteger atomicInt){this.atomicInt = atomicInt;}
    @Override
    public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());}}
```

JAVA.UTIL.CONCURRENT.ATOMIC



ASSIGNMENT 6: DUNGEON ADVENTURES

- sviluppare un'applicazione client server in cui il server gestisce le partite giocate in un semplice gioco, "`Dungeon adventures`" basato su una semplice interfaccia testuale
- ad ogni giocatore viene assegnato, ad inizio del gioco, un livello X di salute e una quantità Y di una pozione, X e Y generati casualmente
- ogni giocatore combatte con un mostro diverso. Anche al mostro assegnato a un giocatore viene associato, all'inizio del gioco un livello Z di salute generato casualmente

ASSIGNMENT 6: DUNGEON ADVENTURES

- il gioco si svolge in round, ad ogni round un giocatore può
  - *combattere con il mostro:* il combattimento si conclude decrementando il livello di salute del mostro e del giocatore. Se LG è il livello di salute attuale del giocatore e MG quello del mostro, tale livello viene decrementato di un valore casuale X, con $0 \leq X \leq LG$. Analogamente, per il mostro si genera un valore casuale K, con $0 \leq X \leq MG$.
  - *bere una parte della pozione*, la salute del giocatore viene incrementata di un valore proporzionale alla quantità di pozione bevuta, che è un valore generato casualmente
  - *uscire dal gioco*. In questo caso la partita viene considerata persa per il giocatore
- il combattimento si conclude quando il giocatore o il mostro o entrambi hanno un valore di salute pari a 0.
- se il giocatore ha vinto o pareggiato, può chiedere di giocare nuovamente, se invece ha perso deve uscire dal gioco.

ASSIGNMENT 6: DUNGEON ADVENTURES

- sviluppare una applicazione client server che implementi `Dungeon adventures`
  - il server riceve richieste di gioco da parte dei cliente e gestisce ogni connessione in un diverso thread
  - ogni thread riceve comandi dal client li esegue. Nel caso del comando "combattere", simula il comportamento del mostro assegnato al client
  - dopo aver eseguito ogni comando ne comunica al client l'esito
  - comunica al client l'eventuale terminazione del del gioco, insieme con l'esito
- il client si connette con il server
  - chiede iterativamente all'utente il comando da eseguire e lo invia al server. I comandi sono i seguenti `1:combatti`, `2: bevi pozione`, `3: esci del gioco`
  - attende un messaggio che segnala l'esito del comando
  - nel caso di gioco concluso vittoriosamente, chiede all'utente se intende continuare a giocare e lo comunica al server