Java: Using Threads and Synchronizers

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

alessio.bechini@unipi.it

Outline

Understanding how to use threads and, in the Java scenario, what high-level mechanisms help support their synchronization and coordination

- Java Threads and JVM Runtime Data Areas (RDA)
- Tasks vs Threads
- The Executor framework and thread pooling
- Java synchronizers and thread-safe Data Structures
- Java Memory Consistency Model

Language-level Threading

Defining Threads in a Program

Defining a thread means:

- constructing all the supporting data structures,
  possibly specifying through them the thread characteristics.
- providing the relative management actions.

All this, in Java, is done making use of the class `Thread`.

Thread body: sequence of instructions.

How to define it? Natural choice: via a function/method body.

The body of a Java thread corresponds to a *dedicated method*.

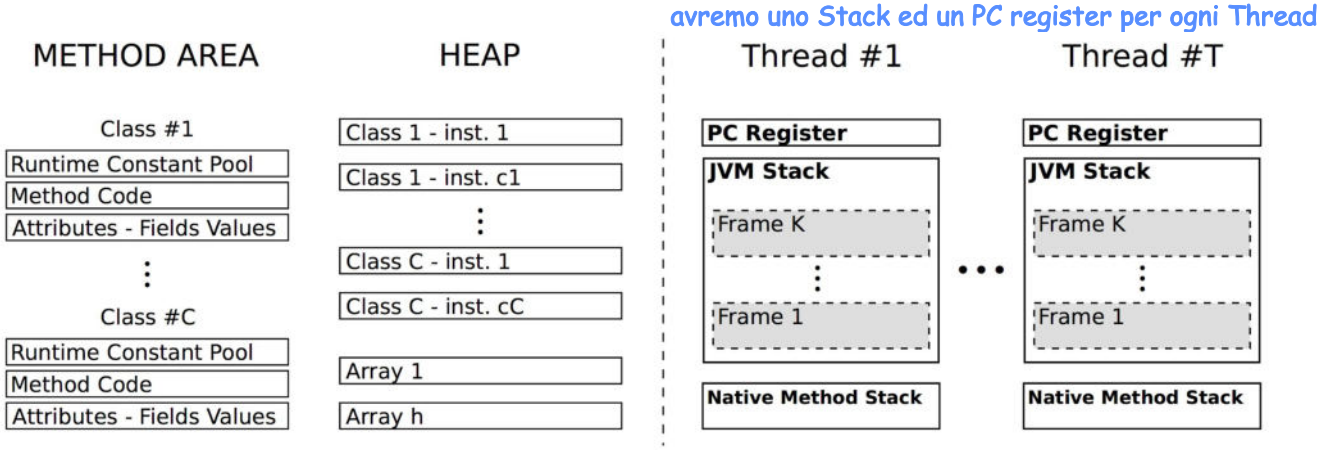
JVM Runtime Data Area

(Java Virtual Machine)

JVM RDA (Runtime Data Areas)

3 distinct parts to host: Bytecode, Objects, and Stacks for Threads

avremo uno Stack ed un PC register per ogni Thread

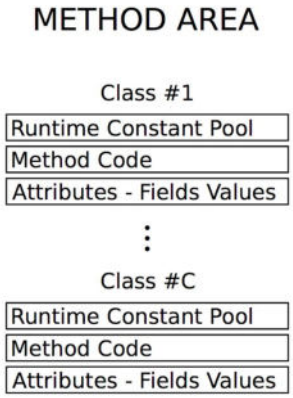


le classi in java vengono caricate dinamicamente, cioè il codice di una classe viene caricato solo quando una classe viene utilizzata per la prima volta nel codice.

· Nello heap troviamo le istanze delle classi referenziate dalle variabili (reference) memorizzate nelle altre aree di memoria.

Method Area

Shared *among all threads*. Holds class-level data of each *.class* file.



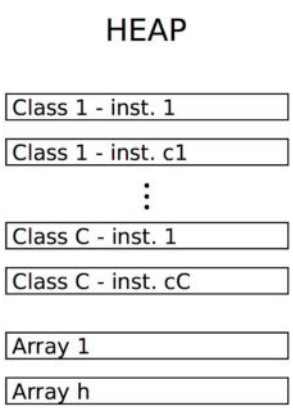
For each class, it contains:

- the Runtime Constant Pool, acting similarly to an ordinary symbol table;
- code for methods and constructors;
- field and method data.

È una struttura di dati che funge da tabella dei simboli per una classe specifica. Contiene costanti letterali e riferimenti simbolici necessari per la risoluzione dinamica.

Heap

Shared *among all threads*. Aimed to host *objects* of the program.



It contains the representations of all the **instances** of all the classes handled in the program, as well as representations of arrays.

It is subject to garbage collection.

Qui troveremo le variabili interne alle classi (campi) istanziate.
Le variabili locali delle funzioni invece le troveremo nella Stack Area.

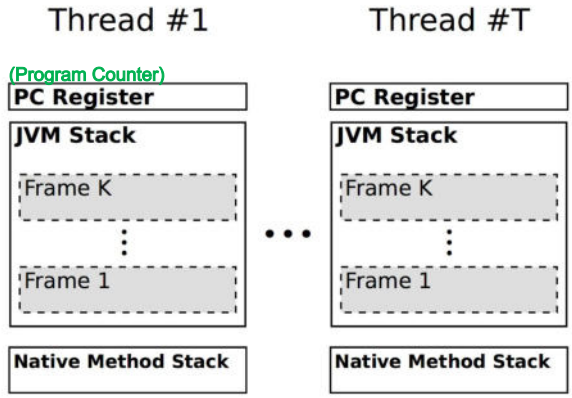
Stack Area

Il PC register (uno per ogni thread) tiene traccia dell'indirizzo della prossima istruzione di bytecode che deve essere eseguita dal thread corrente. In altre parole, è un puntatore che indica la posizione dell'istruzione successiva nel flusso di esecuzione.

Each Thread has its own stack. Any function call is managed using a Frame in the pertinent stack.

Each frame contains:
- an array of local variables
- its operand stack
  (the JVM is a stack machine!)
- a ref to its runtime const pool

Native methods are supported by “C stacks.”



A differenza di quello che abbiamo visto a Reti Logiche in cui le operazioni fra operandi venivano fatte principalmente sfruttando i registri del processore (dopo aver spostato in essi i valori su cui operare), la JVM invece sfrutta principalmente lo stack. Quindi le operazioni vengono eseguite tra operandi memorizzati nello stack.

Basic Thread Handling in Java

Thread Creation and Running

This topic has been already presented in other courses.
Two basic ways:

- **Extension of class `Thread`** - a.k.a. "Threading by subclassing."
  Problem: no possibility to extend any other class. Siccome in Java l'eredità è singola, il dover estendere la classe Thread può essere una forte limitazione.
- **Use of a `Runnable` instance in the constructor of `Thread`** -
  a more flexible mechanism.

This is a "functional" interface, i.e. with only one abstract method

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

Explicit thread spawning:
"unblocking" instance method `mythread.start()`

Noi useremo sempre il secondo modo

Thread Definition by `Runnable`

The Runnable object can be defined either via a class that implements Runnable, or via an anonymous class, or via a 0-parameter lambda.

For example:

```
Runnable mytask = () -> {
    String myName = Thread.currentThread().getName();
    System.out.println("This is " + myName);
};

mytask.run();  //the run method can be either invoked...
// ... or used as the body of a thread
Thread myThread = new Thread(mytask);
myThread.start();
```

Ordinary method call

Thread spawning

il metodo 1 è quello che abbiamo usato a Prog Avanzata ed è sicuramente il più semplice (almeno finchè non imparo le lambda)

Thread Termination

Some early methods *deprecated* due to related problems:

`stop()` - A stopped thread releases all the locked monitors, so it may lead to *state inconsistencies*

`suspend()/resume()` - Deadlock-prone: a thread may suspend in a critical section, and another thread able to resume it cannot access the code to do it.

stop() e suspend() sono metodi che un thread può chiamare su un altro thread, non su se stesso

incline

Solution: *cooperative mechanism* based on *inter-thread interruption* where one thread asks another to stop.

tra

"invece di ucciderlo io, gli chiedo gentilmente di suicidarsi"

Inter-Thread Interruption

Each Thread holds an "interrupted" flag to be possibly set by another thread, by invoking the `interrupt()` method on it.

A Thread can check its own interrupted flag via the static method `Thread.interrupted()`, which *clears the flag as well*.

The flag of any thread can be checked via the instance method `isInterrupted()`, which does not modify the flag value.

Typically, a thread that becomes aware of being interrupted is programmed to perform cleanup actions and terminate.

sono io programmatore che devo controllare periodicamente il flag e definire come deve avvenire la terminazione del thread (non c'è un comportamento di default).

Un thread B deve verificare periodicamente se è stato interrotto. Questo può essere fatto manualmente attraverso il metodo isInterrupted() o automaticamente da alcune operazioni che possono sollevare InterruptedException.

Interruption and Locking

What if an inter-thread interruption is performed to a *blocked* thread? It depends...

Cosa accade se voglio interrompere un thread bloccato? Dipende dal motivo per cui è bloccato:

- If it is blocked on an explicit waiting operation, an `InterruptedException` awakes the target thread (and clears the flag).
- If it is blocked on accessing a lock/monitor, it is not awaken; with explicit locks, `lockInterruptibly()` can also be used.

Dealing with InterruptedException

It is particularly important to provide the proper handling of interrupt exceptions.

Guidelines: apart possible cleanup actions, we should either:

opzione 1:
- re-throw the exception, after a possible re-wrapping;*

opzione 2:
- or restore the interrupted flag by invoking `Thread.currentThread().interrupt()`, so to keep memory of the occurrence of the interruption event.

  Il flag di interruzione deve essere ripristinato in modo che altri pezzi di codice che utilizzano il thread siano consapevoli che il thread è stato interrotto.

* opz1: Questo può essere fatto se si ritiene che l'eccezione debba essere gestita più in alto nella pila delle chiamate o se il thread stesso non può continuare l'esecuzione in modo significativo dopo l'interruzione.

Thread Pooling and the Executor Framework

Task vs Thread, i.e. *What* vs *How*

"**Thread**": a sequence of instructions that can be executed independently of others in the same program, typically under the control of a scheduler.

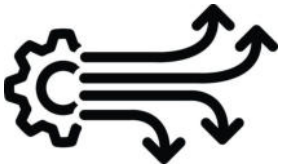
Preoccupazione principale nei programmi concorrenti

Prime concern in concurrent programs: identification of actions/jobs to be carried out, *regardless of the way they will be executed*.

indipendentemente

In this cases, the term "**task**" is used for the relative instructions.

Example: in a server, requests have to trigger *tasks*; such tasks *could* be executed using multithreading.



- Thread (How): Riguarda il "come" le operazioni vengono eseguite. I thread sono meccanismi specifici che eseguono il codice in parallelo.
- Task (What): Riguarda il "cosa" deve essere fatto. I task sono le azioni o i lavori che devono essere completati, senza specificare il metodo di esecuzione.

Separating Definition and Execution

A ***task*** can be defined via a `Runnable` class, in the `run()` method.

The task execution could be done in many different ways;
A general solution asks for *developing classes aimed at this*.
Such classes must implement at least the following interface:

```
public interface Executor {
    void execute(Runnable command);
}
```

This, as most of classes/interfaces for concurrent programming, is in java.util.concurrent packages

L'esecuzione di un task definito in una classe Runnable può essere realizzata in vari modi. Ad esempio, potrebbe essere eseguito da un thread direttamente, o potrebbe essere passato a un esecutore che gestisce l'esecuzione.

- Uno dei metodi più semplici per eseguire un task definito come Runnable è creare un nuovo oggetto Thread, passando l'istanza del Runnable al costruttore della classe Thread (a Programmazione Avanzata facevamo così).
- Java però fornisce una libreria di classi nell'API java.util.concurrent che gestiscono l'esecuzione di task in modo più flessibile ed efficiente. Si possono quindi sviluppare classi specifiche che si occupano di eseguire i task, tali classi devono implementare almeno l'interfaccia "Executor".

Use of an Executor Class

In a program, specific executor classes can be first defined and then instantiated.

A task can be executed by means of them.

Un task può essere eseguito utilizzando queste classi esecutori, che si occupano di gestire i dettagli dell'esecuzione.

Interface

Concrete class

```
Executor myExecutor = new MyCustomExecutor();
myExecutor.execute(new RunnableTask1());
myExecutor.execute(new RunnableTask2());
```


In questo esempio, MyCustomExecutor è una classe personalizzata che implementa l'interfaccia Executor. Questa classe è stata scritta dal programmatore con l'intento di definire un determinato comportamento per l'esecuzione dei task.

Nella prossima slide vediamo due possibili modi (sbagliati) di implementare la classe MyExecutor.

Two Trivial Examples

banali

A task can be executed as part of the same thread that asks its execution...

```
class DirectExecutor implements Executor {
     public void execute(Runnable mytask) {
          mytask.run();
     }
}
```

Or we can spawn a brand new thread to execute one task!

nuovo di zecca

```
class ThreadPerTaskExecutor implements Executor {
     public void execute(Runnable mytask) {
          new Thread(mytask).start();
     }
}
```

Of course, none of these looks to be a brilliant solution.

Perché nel primo caso non sfrutto il possibile parallelismo nel nostro sistema, mentre nel secondo, l'utilizzo di un nuovo thread per il task potrebbe portare a casi in cui ho dieci task, per i quali verranno avviati 10 thread concorrenti e magari nella mia macchina ho solo 2 core...

Performance Penalties: Common Causes

avere tanti (troppi) thread mi rallenta il sistema. Ad esempio dovrò fare dei context switch...

Penalties come from wasting time in *management* actions, which can be seen as *overhead* in our program. Possible causes:

- **Context switches** - not only they take time, but they also determine cache flushes that slow down the subsequent execution.
- **Lock contention** - determines task serialization (no parallelism), hampering scalability: thread waiting time should be reduced.

contesa

ostacolando

- **Memory Synch** - many threads may need to see last updates. Some unnecessary synch can be removed by the JVM (*lock elision*) or upon static *escape analysis* to spot out thread-local objects.

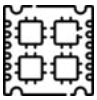
```
synchronized (new Object()){
     foo();
}
```

in questo esempio lokkare quell'oggetto non ha senso perché sto lokkando un oggetto che quel thread stesso crea. Infatti ogni thread che eseguirà questo codice, creerà un suo nuovo oggetto e quindi non ha senso lokkarlo. ! ha senso lokkare solo gli oggetti condivisi tra più thread.

Towards Thread Pooling

Thread creation/disposal (smaltimento) determines computational overhead.

It is not reasonable to allow an unbounded amount of threads to execute at the same time a large number of tasks, because:



- the number of CPUs/cores is limited
- memory is a limited resource!

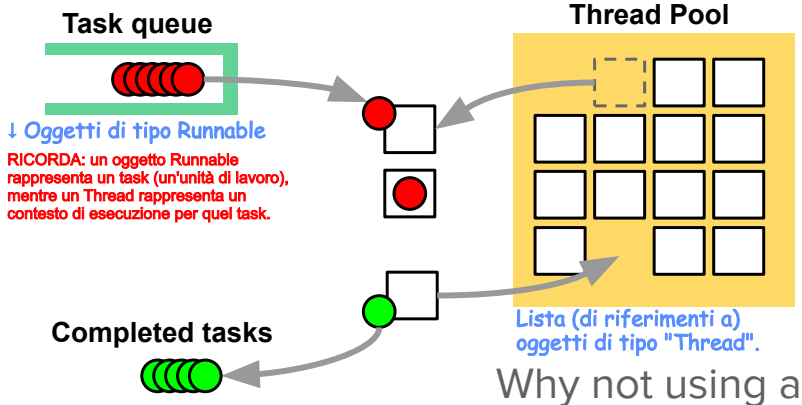
This kind of problems usually affect *server* software.

Idea: Keep some threads ready to execute upcoming tasks and, upon completion of each execution, make the thread available again for another task.

Creo all'inizio un giusto numero di thread e poi non ne creo di nuovi. Quando avrò dei task da portare a termine li assegnerò a questi thread. Questo approccio è chiamato "thread pooling".

Thread Pooling

One of the most popular design pattern for multithreaded software.

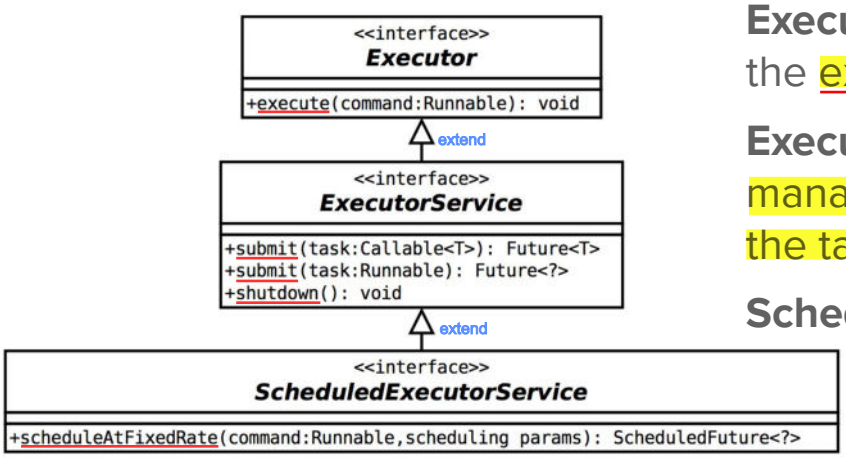


The pool size can be either static or dynamic, and different policies can be devised for the size update.

Why not using an Executor for this?

è una domanda retorica, in realtà è una buonissima idea

Executor Interfaces



Executor: only support for the execution of a task

ExecutorService: support for managing the lifecycle of both the task and the executor (stesso)

ScheduledExecutorService: support to timed/periodic execution of tasks.

La submit() è diversa dalla execute() base, infatti usandola indico all'Executor che un certo task dovrà essere eseguito, sarà poi l'Executor a scegliere quando e come eseguirlo. Inoltre con la sumbit posso avere un parametro di ritorno: un Future che può essere usato per ottenere lo stato del task o annullarlo.

In ExecutorService abbiamo due tipi di submit(), in base al tipo di parametro passato:
- submit(Runnable task): Questa versione accetta un Runnable, consentendo di continuare a usare Runnable ("per retrocompatibilità"). Poiché Runnable non restituisce un risultato, il Future<?> restituito non conterrà un risultato utile (sarà null), ma può essere utilizzato per controllare se il task è completato, annullato, o se ha lanciato un'eccezione.
- submit(Callable<T> task): Questa versione accetta un Callable<T>, che restituisce un risultato. Il Future<T> restituito può essere utilizzato per ottenere il risultato una volta completato il task.

Gli esecutori possono essere progettati per implementare specifiche politiche di esecuzione, come il numero di thread da utilizzare, la gestione delle code di task, e la priorità dei task.
Separando la sottomissione dall'esecuzione, è possibile applicare queste politiche senza influire sul modo in cui i task vengono sottomessi.

Separating Submission and Execution

Il task viene definito e inviato per l'esecuzione. Questa operazione non implica che il task verrà eseguito immediatamente.

Executors can be designed to *implement an execution policy*, so it is important to tell apart task submission and execution. This is the rationale behind methods in `ExecutorService`.

Executors are a means to separate task *submission* and *execution*.

The management of executions can be more complex than expected, as some additional aspects must be considered:

- A task result may need to be explicitly managed;
- Synchronization for getting the result may be required.

Infatti, vedremo che:
- ExecutorService utilizza l'interfaccia *Future* per rappresentare i risultati dei task che possono non essere immediatamente disponibili.
- La sincronizzazione è necessaria per ottenere i risultati dei task in modo sicuro. Questo può implicare l'attesa che un task completi la sua esecuzione prima di poter accedere al risultato.

Il metodo get() dell'interfaccia Future è bloccante, il che significa che attende finché il risultato del task non è disponibile.

Quando un task deve restituire un risultato, l'interfaccia Runnable non è sufficiente poiché non fornisce un modo per gestire il risultato. In questi casi, il task deve essere implementato utilizzando l'interfaccia Callable.
L'interfaccia Callable fa parte del pacchetto java.util.concurrent e permette ai task di restituire un valore e di lanciare eccezioni.

Tasks: Beyond Runnable

Whenever a result for a task has to be managed, the task must be implemented according to the `Callable` interface:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

- Generics (<V>): l'interfaccia Callable in Java è definita come generica, il che significa che può essere parametrizzata con un tipo specifico al momento della sua dichiarazione. Questa parametrizzazione permette di specificare il tipo del valore che il metodo call() restituirà.
- Metodo call(): Questo metodo è dove si implementa il codice del task. Può restituire un valore di tipo V e può lanciare eccezioni.

How to retrieve the returned value? This is not a plain invocation!

Poiché il metodo call() non viene invocato direttamente (come nel caso di Runnable), è necessario un meccanismo per recuperare il valore di ritorno. Questo meccanismo è fornito dall'interfaccia Future.

The solution is in the way the task is submitted: `submit()` returns back a **`Future<V>`**, which in turn will include the result as it will be computed by the task (and accessible via `get()`).

Nella frase "Poiché il metodo call() non viene invocato direttamente (come nel caso di Runnable), è necessario un meccanismo per recuperare il valore di ritorno", il soggetto implicito è il codice che utilizza un oggetto che implementa l'interfaccia Callable.

Quando si utilizza un oggetto che implementa Callable, il metodo call() non viene chiamato direttamente dal codice chiamante come accade con Runnable, dove si chiama direttamente il metodo run(). Invece, il metodo call() di un oggetto Callable viene eseguito all'interno di un contesto gestito dall'executor (ad esempio ExecutorService), che è responsabile di eseguire il task in un thread separato.

Il "meccanismo per recuperare il valore di ritorno" si riferisce al fatto che, poiché il metodo call() viene eseguito in modo asincrono (non direttamente dal chiamante), è necessario un modo per ottenere il risultato prodotto dal metodo call() una volta completato l'esecuzione del task. Questo viene gestito tramite l'uso di un oggetto Future<V>, che rappresenta il risultato futuro dell'esecuzione del task. Utilizzando Future.get(), il chiamante può ottenere il valore restituito dal metodo call() una volta che è disponibile.

Futures and Promises

A **future** is a general thread-safe construct adopted in concurrent languages, referring *a proxy for a result computed asynchronously*.
che funge
The value for a future is allowed to be written only once.

terminologia: Sometimes the function in charge of setting the value for a future is named **promise**.

Different possible promises may set the value of a given future, though this can be done only once for a given future.

Java Futures

The behavior of a “future” is described by the relative interface:

```
public interface Future<V> {
    boolean isDone();
    V get(); //also a timed version exists
    boolean cancel(boolean m);
    boolean isCancelled();
}
```

It is possible to check if a result is ready (isDone()), retrieve it (get()), try to cancel a task execution, check if a task has been cancelled.


```
public class FutureExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(1);

        // Creazione di un future per un task che restituisce una stringa
        Future<String> future = executor.submit(() -> {
            Thread.sleep(2000); // Simula un calcolo lungo
            return "Hello, Future!";
        });

        // Esempio di utilizzo del future per ottenere il risultato
        if (future.isDone()) {
            String result = future.get(); // Blocca fino a quando il risultato non è disponi
            System.out.println("Risultato del future: " + result);
        }

        executor.shutdown();
    }
}
```

! ERRORE NEL CODICE:
Il metodo future.isDone() restituirà false la prima volta che viene eseguito, perché il task asincrono (eseguito nel thread del pool) non è ancora stato completato.
Quindi, la condizione falsa fa sì che il programma non eseguirà mai future.get()

Off-the-shelf ExecutorServices

`java.util.concurrent.Executors` has factory methods to create instances of executors with pooled threads, e.g.:

`newFixedThreadPool(int n)` creates a pool of n threads, with fixed size

cioè il numero di thread nel pool rimane costante una volta che è stato creato

`newCachedThreadPool()` provides a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

`newWorkStealingPool(int parallelism)` creates a pool with dynamically managed threads, able to handle the specified level of parallelism;
the task execution order does not necessarily reflect the order of submission.

A differenza dai tipi precedenti in cui si aveva una coda comune per i task da eseguire, nel *WorkStealingPool* ogni thread ha la propria coda di task da eseguire. Se un thread finisce i suoi task e la sua coda è vuota, può "rubare" un task dalla coda di un altro thread. Questo è il motivo per cui l'ordine di esecuzione dei task può non riflette l'ordine di presentazione, a causa di questo "furto" di lavoro.

ScheduledExecutorService è una specializzazione di ExecutorService che aggiunge funzionalità per il scheduling dei task, consentendo di specificare tempi di esecuzione ritardati o periodici.

ScheduledExecutorService

Interface for an ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

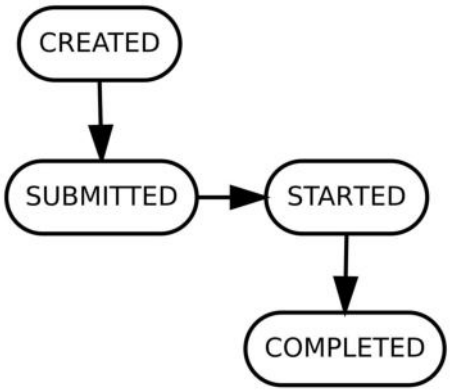
Scheduling options: see API documentation

"se vi serve andate a guardare la documentazione"

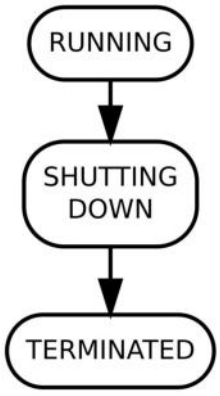
Standard implementation returned by factory method in “Executors”:

`newScheduledThreadPool(int n)` creates a pool of n threads, to be scheduled (often used with n=1)

Lifecycle of Tasks and ExecutorService



Lifecycle of a Task



Lifecycle of an ExecutorService

NB: gli ExecutorService eseguono i task su thread diversi da quelli che gestiscono l'ExecutorService stesso.

Stopping Executors

If we want to shut down an executor, we need to take care: the relative threads *might possibly be still running at that time*.

why? because if I want to stop an executor, there may be some tasks in the internal queques that have been submitted but they have not been executed (or completed) yet.

Distinct operations, as methods of `ExecutorService`:

- `shutdown()` starts the shutdown procedure; previously submitted tasks are completed, and no new task will be accepted.
- `shutdownNow()` - only *currently executing* tasks are completed.
- Check state by using `isShutdown()`; `isTerminated()` checks if all tasks have been terminated.
- Methods above are not-blocking; to wait for completion of all tasks, call `awaitTermination()`

Bilancio finale: se dobbiamo usare i thread, evitare di usare le funzione base (start e run), usare invece gli executors. Se dobbiamo fare qualcosa di particolare come una specifica policy per la gestione dei nostri thread, ci creiamo un nostro custom executor e dopodiché userò quello, senza maneggiare direttamente i thread.

Java Synchronizers

Ora sappiamo come usare i thread, ma questi devono essere sincronizzati così da poter lavorare in modo coordinato, vediamo quindi dei meccanismi per supportare diversi tipi di sincronizzazione.

Synchronizers in Java

Cooperation across threads often requires specific synchronization actions, to be performed according to a given pattern.

A construct designed to address a specific synchronization pattern is called "Synchronizer," and it coordinates control flow of threads based on its internal state.

Java provides several classes for common special-purpose synchronization. Among them:

Latches, Barriers, Semaphores, Exchangers ce ne sono anche altri...

Latches

è utile quando si vuole sincronizzare più thread in un punto di controllo specifico e consentire loro di procedere solo quando tutti sono pronti.

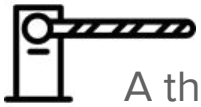
A **latch** is aimed at making a group of threads wait *until it reaches its terminal state*: → all threads pass through.

An instance of a latch can be used only once.

In Java, the `CountDownLatch` class is present: the constructor takes a positive integer to initialize the state ("count" var).
State: updated by `countDown()` (count--); read by `getCount()`

A thread waits until the release point in time by invoking `await()`
- a timed version is available as well.

Barriers



I thread che partecipano alla barriera si riuniscono tutti in un punto di sincronizzazione e aspettano finché tutti i thread non hanno raggiunto quel punto. Una volta raggiunto, tutti i thread vengono rilasciati contemporaneamente.

A **barrier** makes a group of threads meet at a barrier point.
A thread that gets to the barrier waits for all the others to come.
As the last thread arrives, all threads are released to proceed.

The reference Java implementation is `CyclicBarrier`.
A `CyclicBarrier` object can be reused
upon invoking `reset()` on it.

A thread waits until the release point in time by invoking `await()`
- a timed version is available as well.

An optional “barrier action” can be specified as well.

DIFFERENZE Latches VS Barriers:
- le Barrier sono riutilizzanili, i latch NO
- Stato Terminale vs Punto di Sincronizzazione Ciclico: Il CountDownLatch raggiunge uno stato terminale quando il contatore raggiunge zero, mentre la CyclicBarrier sincronizza i thread in un punto specifico e li rilascia ciclicamente per il successivo ciclo di sincronizzazione.

Phasers

A **phaser** is a more sophisticated version of a barrier/latch.

It is a reusable synchronizer.

(number of participating threads)

Differently from CyclicBarrier, the number of parties can change over time: it is designed to operate in successive phases.

For details, refer to the API documentation.

"Parties" è il termine utilizzato per rappresentare le unità che partecipano alla sincronizzazione attraverso la Phaser. Ogni thread che partecipa viene considerato come un "party".

Semaphores

sono esattamente quelli visti a Calcolatori Elettronici

A **semaphore** is aimed to *control the number of accesses* to a resource. It thus supports a generalization of mutual exclusion.

In Java, the `Semaphore` class implements this synchronizer.

The number of accesses is controlled by using **permits**;
the maximum number of available permits is set at initialization time.

- `acquire()` is used to get a permit; if it is not available, it blocks. "wait"
- `release()` is used to give a permit back to the semaphore. "signal"

The corresponding classical names are `P()`/`V()` or `down()`/`up()`

Exchangers

An exchanger is aimed at making two threads wait for each other at a synchronization point, and swap objects. scambiare oggetti tra di loro in un punto di sincronizzazione specifico

In Java, it is implemented by the `Exchanger<V>` class.

V is the type of the object to be exchanged.

Example of code for an exchange operation:

```
myBuffer = exchanger.exchange(myBuffer);
```

Un esempio comune di utilizzo di Exchanger è nei problemi di produzione e consumo, dove un thread produce dati (ad esempio, in un buffer) e l'altro thread li consuma. L'Exchanger consente ai thread di sincronizzare il passaggio dei dati tra di loro in modo efficiente.

Thread-safe Data Structures

“Synchronized” Collections

collection := struttura dati che può contenere un insieme di elementi (ArrayList, HashMap, ecc...)

A first, trivial attempt to define a thread-safe collection is to make its methods “synchronized.”* This can be done in a systematic way by using the Decorator pattern.

Una collezione è considerata thread-safe se può essere utilizzata simultaneamente da più thread senza causare problemi di concorrenza come race condition.

A standard way to decorate an ordinary collection can let us obtain *a corresponding thread-safe version*.

Methods like `isEmpty()` or `size()` are not really meaningful in a concurrent setting, as the returned value might not be necessarily accurate just at the subsequent operation.

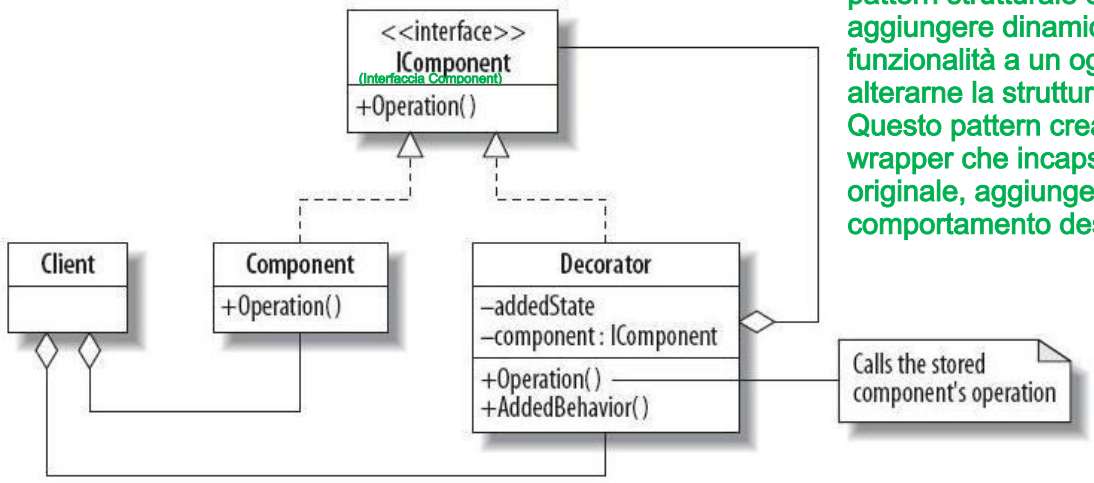
Ad esempio, un altro thread potrebbe modificare la collezione subito dopo che il metodo size() ha restituito il valore, rendendo il risultato non affidabile.

* Questo viene fatto utilizzando la parola chiave synchronized nella definizione dei metodi stessi. Così facendo, solo un thread alla volta può eseguire un metodo sincronizzato della stessa istanza dell'oggetto.

I can use the *decorator pattern* to redefine all the methods, making them synchronized.
By labeling them as "synchronized" and, within the synchronized method, calling the original, non-synchronized method, I create a simple trick to ensure everything is synchronized.
This approach provides a corresponding thread-safe version of the collection.

The Decorator Pattern

▲ Lui stesso di questa slide dice poco o nulla...



Il Decorator Pattern è un design pattern strutturale che consente di aggiungere dinamicamente nuove funzionalità a un oggetto senza alterarne la struttura.
Questo pattern crea una classe wrapper che incapsula l'oggetto originale, aggiungendo il nuovo comportamento desiderato.

"Initially, you have your Component that you want to enhance with additional behavior. In this case, I want to make every method synchronized.

Any client code can use either the original component or the decorated version of the component. In a concurrent setting, however, only the decorated version will be used."

Getting a Synchronized Collection

Specific static factory methods in class `java.util.Collections` of the form `.synchronizedXXX(XXX<T> ds)` return synchronized versions of the plain data structure `ds`.

`XXX` stands for the interface `Collection`, or for `List`, `Map`, etc..

Iteration on synchronized collections needs explicit synchronization *on the whole data structure*, e.g.

```
Collection<T> c = Collections.synchronizedCollection(myCol);
synchronized(c) {
    for (T item : c) foo(item);
}
```

(In this example, foo is a function that locks only a single element of the collection, not the entire collection)

Making the methods synchronized is likely the most straightforward way to handle thread safety with collections. However, using single synchronized methods is often insufficient for ensuring the collection itself is thread-safe.

For example, if you want to iterate over a synchronized collection, which is a common operation for lists, you need to obtain an explicit lock on the entire data structure. In contrast, individual synchronized methods typically operate on single elements.

PROBLEM: despite improvements in the latest versions of Java and the JVM, working with synchronized blocks is still not very efficient.

Addressing Performance

A "synchronized" collection shows poor performance *under high contention*.

Main reason: coarse-grained synchronization level.

Solution:
reduction of synchronization overhead, applying *any possible trick*.

"Concurrent" versions of "Synchronized" collections have been developed: e.g.,
`ConcurrentHashMap` is an improvement of `SynchronizedMap`

ConcurrentHashMap

It implements the `ConcurrentMap<K,V>` interface, which includes also atomic read+write operations, e.g.:

`boolean replace(K key, V oldValue, V newValue)`
- equivalent to the atomic execution of

```
if (map.containsKey(key) && Objects.equals(map.get(key), oldValue)){
   map.put(key, newValue);
   return true;
} else return false;
```

Null-safe equals, since Java 1.7

Iteration over Concurrent Collections

Iterating over a concurrent collection is risky:
other threads may access the data structure at the same time!

The same problems arise with “for-each” loops,
which are compiled to the corresponding iterator-based form.

So, every time we are in a concurrent setting, we must specify new semantics for an iterator to describe what happens in the event of possible interferences during access to the data structure.

It is necessary *specifying new semantics for a concurrent iterator*,
so to fully describe its behavior in case of possible interferences.

In various constructs, we will encounter different possible semantics for iterators.
The simplest semantics for a concurrent iterator is the so-called *fail-fast* semantics.

Fail-fast Iterators

Provided by most of non-concurrent collections.

Used to traverse the *actual* collection:
no modification to the collection is allowed.

As soon as a modification is detected (on a best effort basis),
`ConcurrentModificationException` is thrown.

Modifications are detected by inspecting specific counters.

Note: it is safe to remove an element via the `remove()` method
of the iterator - not the `remove()` of the collection!

Weakly-consistent Iterators

Called also “fail safe.” Provided by most of concurrent collections.

According to the official docs, such iterators:

- may proceed concurrently with other operations
- will never throw `ConcurrentModificationException`
- are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

VANTAGGIO: Consentono modifiche alla collezione durante l'iterazione, migliorando la flessibilità e riducendo la necessità di bloccare l'accesso alla collezione.

L'iteratore potrebbe riflettere o meno le modifiche fatte alla collezione dopo la sua creazione. Questo significa che gli elementi aggiunti, rimossi o modificati durante l'iterazione potrebbero o non potrebbero essere visti dall'iteratore. Non c'è una garanzia che tutte le modifiche saranno visibili. Alcune modifiche potrebbero essere visibili, altre no, dipendendo dall'implementazione specifica della collezione e del suo iteratore.

Le collezioni "Copy-On-Write" (COW), sono una forma di collezioni thread-safe progettate per supportare la lettura simultanea e la scrittura senza l'uso esplicito di meccanismi di sincronizzazione come lock o mutex.

Copy-On-Write: Il termine "Copy-On-Write" indica che quando si effettua una modifica alla collezione, viene creata una copia della collezione esistente, e la modifica viene apportata solo sulla copia. Questo assicura che la versione originale della collezione rimanga immutata durante le letture e le modifiche.

Scritture Costose: Le operazioni di scrittura sono relativamente costose, poiché coinvolgono la creazione di una copia della collezione. Tuttavia, questo design è ottimizzato per scenari in cui le letture sono molto più frequenti delle scritture.
Poiché le letture possono essere eseguite simultaneamente senza problemi di sincronizzazione, le collezioni COW sono adatte a scenari in cui molteplici thread effettuano frequenti operazioni di lettura e le operazioni di scrittura sono meno frequenti.

Snapshot Iterators

Used with copy-on-write collections like `CopyOnWriteArrayList`.

The iterator provides a snapshot of the state of the collection when the iterator was constructed.

No synchronization is needed while traversing the iterator.

The iterator does NOT support the `remove()` method.

In such collections, all “mutative operations” are implemented by making a fresh copy of the underlying data structure, i.e. *Copy-On-Write*.

MODIFICA DELLA COLLEZIONE ORIGINALE
La collezione originale non viene mai modificata direttamente. Ogni volta che una scrittura avviene, viene creata una nuova copia, e questa nuova copia viene usata per tutte le operazioni future. Quindi, tecnicamente, la collezione originale rimane immutata. Tuttavia, l'oggetto che rappresenta la collezione può essere aggiornato per puntare alla nuova copia.

Blocking Queues

High-performance, thread-safe version of *bounded buffer*, designed according to the `BlockingQueue<E>` interface.

Methods are present to support insertion, extraction, and inspection in different fashions (blocking, throwing exceptions, polled, timed): see API documentation.

Blocking queues support the producers/consumers pattern.

Blocking queues make resource management more reliable.

Impl. of the classic bounded buffer: `ArrayBlockingQueue<E>`

Blocking queues make resource management much more reliable and, consequently, reasonably efficient. Therefore, the best idea to manage the resource might be to make use of a blocking queue.

Memory Consistency Rules (... for Java)

Example: Lazy Singleton Initialization

A "singleton" is a design pattern that ensures only a single instance of a particular class exists.

A popular design pattern to have one single instance of a class.

A static factory method has to return the single instance, which is usually setup with *lazy initialization*.

(the object is created the first time it is requested)

The solution aside **is not suited to a multithreaded setting!**

```
class SingletonPlace { //Single-threaded version
  private static Helper ref;
  public static Helper getUniqueInstance() {
    if (ref == null) { //lazy initialization
      ref = new Helper();
    }
    return ref;
  }
// other functions and members...
}
```

Helper è una classe fittizia: un esempio di una classe che si desidera gestire come singleton.

Singleton: Double-Checked Locking!

Idea: let's try to shrink the synchronized portion!

This way, it does not work properly: WHY?

"there is a very very subtle bug"

Cause: Unsafe publication of "ref," because of **visibility of memory updates...**

```
class SingletonPlace { //multithreaded version
  private static Helper ref;
  public static Helper getUniqueInstance() {
    if (ref == null) {
      synchronized(this) {
        if (ref == null) {
          ref = new Helper();
        }
      }
    }
    return ref;
  }
// other functions and members...
}
```

PROBLEMA: "Unsafe publication" significa che la pubblicazione della nuova istanza di Helper non è sicura e non garantisce che altri thread vedano la variabile ref completamente inizializzata immediatamente dopo la sua creazione.

Quando un thread esce da un blocco synchronized, in teoria, tutte le modifiche fatte all'interno del blocco dovrebbero essere visibili agli altri thread. Tuttavia, nel caso del double-checked locking, abbiamo due problemi specifici legati alla visibilità della memoria:
1) Riorganizzazione delle istruzioni: Il compilatore e la CPU possono riorganizzare le istruzioni per ottimizzare le prestazioni. Questo potrebbe portare a situazioni in cui la variabile ref viene assegnata prima che l'oggetto Helper sia completamente inizializzato.
2) Cache del thread: Ogni thread può avere una cache locale delle variabili, il che significa che le modifiche apportate da un thread potrebbero non essere immediatamente visibili agli altri thread.

Correct Double-Checked Locking

We need to get sure the "ref" publication is subsequent to the construction of "Helper."

Proper *precedence constraints* must be forced.

This has been done by modifying the semantics of "volatile."

```
class SingletonPlace { //multithreaded version
  private static volatile Helper ref;
  public static Helper getUniqueInstance() {
    if (ref == null) {
      synchronized(this) {
        if (ref == null) {
          ref = new Helper();
        }
      }
    }
    return ref;
  }
// other functions and members...
}
```


Behavior of Java Volatile Variables

It has been decided to give *volatile* variables additional properties:

- **Visibility guarantee** - read/writes are directly issued in main memory
- **Happens before guarantees** - to address problems due to reordering of instructions by compiler/CPU, a volatile operation sets a sort of *memory barrier*: with a W_{vol}, no preceding R/W to *other* vars in the same thread can ever be postponed after it;
  with a R_{vol}, no subsequent R/W can be executed before it.

In general, it is important to specify how the language constructs behave with regards to memory operations.

(Memory) Consistency Models

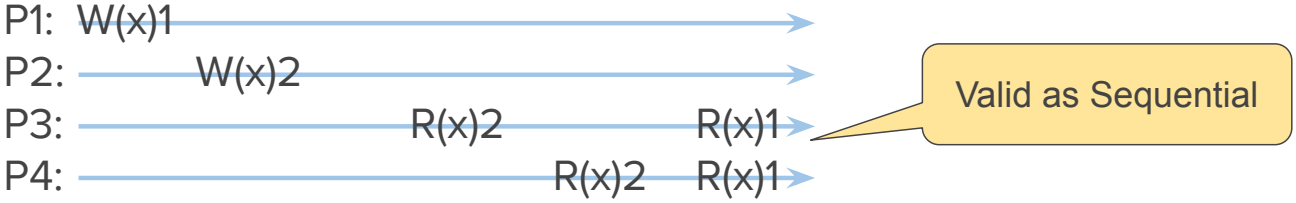
The goal of a memory consistency model is to characterize, with a set of rules, the way operations towards memory locations are visible to processes. Studied for multiprocessors (SMPs).

The consistency model specifies a *contract* between the programmer and the system, so that the programmer can count on some guarantees about the behavior of memory operations.

The less stringent the consistency model is, the easier it will be to implement it in HW (and SW) components, *in a more efficient way*.

Example: Sequential Consistency

In this model, a write to a variable does not have to be seen instantaneously, but writes to variables by different processes have to be seen *in the same order by all processors*.

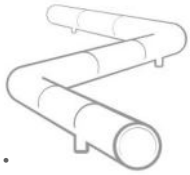


Generalized in a weaker model named “Causal consistency”
(only causally-related writes must be seen in the same order by all processes)

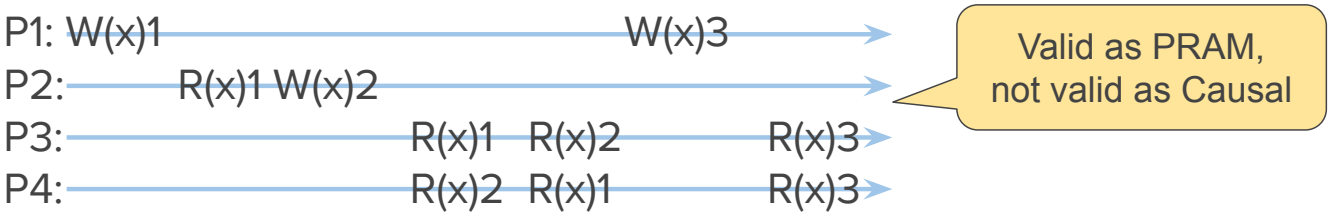
The two write operations, even if the second one appears to be executed later, are actually concurrent in practice. Even if the resulting behavior seems a bit strange, it is not an issue because the important thing here is that all processes see the same order of updates to the variable x.

Example: PRAM Consistency

All processes see memory writes *from one process*
in the order they have been executed by the process
(as though they were "moved" to RAM through a pipeline).



No guarantee exists about the order in which different processes see writes from other different process (apart the "pipeline" rule).



Java Memory Model (I)

Describes legal behaviours in a multi-threaded Java code with respect to the shared memory.

JSR-133 substituted (in 2005) the previous flawed model.

Requirements for the JMM:

- Making common optimizations possible.
- If a program is correctly synchronized, then all its executions will appear to be sequentially consistent.

Related subtleties not easy to be fully understood.

Java Memory Model (II)

Basic idea: formally describing the precedence constraints ("happens before") to be guaranteed in a valid execution, considering the Java constructs involved with synchronization.

In practice, this leads to the definition of rules;
any JVM implementation (+ compiler) must comply with these rules.
I.e., JMM implicitly specifies the set of all admissible executions;
a JVM (+ compiler) must support a subset of these, possibly all.

Real-world JVMs sometimes do not strictly observe all limitations.

Theoretically, it would be ideal if the JVM could support all these rules. However, in practice, the JVM does not adhere to all the constraints described by the rules of the Java memory model.

Java Memory Model - Rules (I)

Program order rule - Each action in a thread happens-before every action in that thread that comes later in the program order.

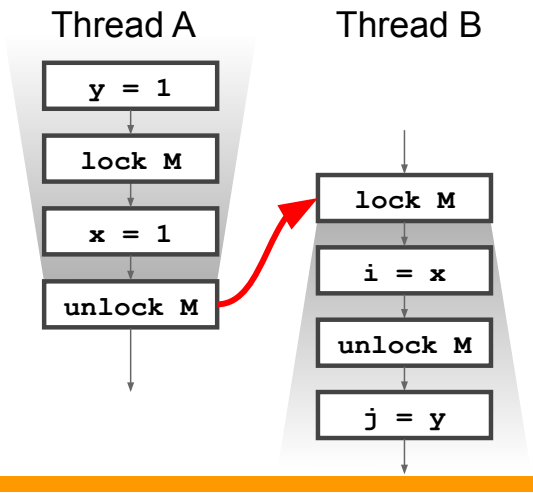
poi magari il PC farà delle ottimizzazioni cambiando l'ordine di esecuzione di alcune istruzioni, ma l'importante è che il risultato sia quello scritto nel codice

In practice, a *within-thread as-if-serial* semantics is suitable for optimizations/out of order executions: the runtime is free to introduce any useful execution optimizations as long as the result of the thread *in isolation* is guaranteed to be exactly the same as it would have been had all the statements been executed in the order the statements occurred in the program (also called *program order*).

Java Memory Model - Rules (II)

Monitor lock rule - An unlock on a monitor lock happens-before every subsequent lock
on that same monitor lock;
the same holds for explicit locks.

Volatile variable rule - A write to a volatile field happens-before every subsequent read of that same field;
the same memory semantics
holds for atomic variables.



Java Memory Model - Rules (III)

Thread start rule - A call to `Thread.start()` on a thread happens-before every action in the started thread.

Thread termination rule - Any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from `Thread.join()` or by `Thread.isAlive()` returning false.

To deal with *safe publication* issues, the synchronization semantics of `final` has been better defined (see official docs).

Java Memory Model - Rules (IV)

Interruption rule - A thread calling `interrupt()` on another thread happens-before the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted()` or `interrupted()`).

Finalizer rule - The end of a constructor for an object happens-before the start of the finalizer for that object.

Transitivity - If A happens-before B, and B happens-before C, then A happens-before C.

! Notare che tutte queste regole possono sembrare davvero banali, ma non è banale in un contesto concorrente a causa della propagazione dell'aggiornamento attraverso l'intera gerarchia della memoria.

Recap: Java Synch Mechanisms

- Volatile variables
- Atomic variables
- Synchronized blocks (implicit locking/monitor)
- Explicit locks
- Concurrent collections

Not covered in these slides; Please check official docs.