

INTRODUZIONE ALLA PROGRAMMAZIONE MULTITHREAD IN JAVA

- PARTE II -

Alessandro Ricci
a.ricci@unibo.it

SOMMARIO

- Interazione e coordinazione fra thread in Java
 - competizione - blocchi/metodi synchronized
 - sincronizzazione - meccanismo wait/notify/notifyAll
- Multithreading e GUI - seconda parte
 - come realizzare interfacce grafiche senza corse critiche
- Uno sguardo oltre la programmazione multi-threaded
 - Task ed Executors
 - Virtual threads
 - cenno ad altri modelli

INTERAZIONE E COORDINAZIONE FRA THREAD IN JAVA

INTERAZIONE E COMUNICAZIONE

- Nel precedente modulo: introduzione alla programmazione concorrente in Java
 - in particolare: programmazione multithread
 - metodologia task-oriented
- Qualsiasi programma (sistema) non banale => più thread che devono interagire/comunicare/cooperare per fornire le funzionalità dell'applicazione nel suo complesso
 - interazione e coordinazione come aspetti fondamentali nella progettazione di un sistema
- Si distinguono due categorie di meccanismi
 - meccanismi per gestire la **competizione**
 - meccanismi per realizzare **cooperazione**

COMPETIZIONE E COOPERAZIONE

- **Competizione**

- interazioni non volute ma necessarie per realizzare il corretto funzionamento dell'insieme dei processi (o thread)
 - **mutua esclusione**
 - accesso mutuamente esclusivo a risorse
 - **sezioni critiche**
 - esecuzione mutuamente esclusiva di blocchi di azioni da parte dei processi

- **Cooperazione**

- interazioni volute e necessarie per realizzare il corretto funzionamento dell'insieme dei processi (o thread) interagenti
 - **comunicazione**
 - scambio di informazioni
 - **sincronizzazione**
 - meccanismi per forzare un ordine temporale fra le azioni dei processi

MUTUA ESCLUSIONE E SINCRONIZZAZIONE IN JAVA: MECCANISMI DI BASE

- **Mutua esclusione (sincronizzazione implicita)**
 - blocchi e metodi synchronized
 - realizzare classi thread-safe
- **Sincronizzazione esplicita e coordinazione**
 - primitive wait, notify, notifyAll

COMPETIZIONE E MODELLI DI MEMORIA

- Il modello di memoria adottato in Java fornisce un insieme molto ristretto di garanzie in merito alla semantica dell'accesso concorrente in lettura/scrittura o solo in scrittura a variabili (e.g. campi di un oggetto) condivise
 - l'accesso puramente in lettura non crea ovviamente problemi
- In particolare:
 - accesso a campi di tipo boolean, char, int e a campi che contengono il riferimento ad un oggetto è garantito essere atomico
 - accesso a campi di tipo long, double non è garantito essere atomico
- Nel prosieguo di questo modulo si considerano tuttavia idiomi e pattern che evitano il più possibile l'accesso concorrente R/W ai medesimi campi

METODI/BLOCCHI synchronized

- Dichiarando un metodo synchronized si vincola l'esecuzione del metodo ad un solo thread per volta.
- I thread che ne richiedono l'esecuzione mentre già uno sta eseguendo vengono automaticamente sospesi dalla JVM
 - in attesa che il thread in esecuzione esca dal metodo
- Dichiarando più metodi synchronized il vincolo viene esteso a tutti i metodi in questione
 - se un thread sta eseguendo un metodo synchronized, ogni thread che richiede l'esecuzione di un qualsiasi altro metodo synchronized viene sospeso e messo in attesa.
 - i metodi synchronized sono dunque mutuamente esclusivi
- Tale vincolo non vale nei confronti dei metodi non synchronized
 - il fatto che un thread sta eseguendo un metodo synchronized, non vieta ad altri thread di eseguire concorrentemente eventuali metodi non synchronized dell'oggetto stesso

PRIMO ESEMPIO

```
public class ResourceUser extends Thread {
    private Resource res;

    public ResourceUser(String name, Resource res) {
        super(name);
        this.res = res;
    }

    public void run() {
        log("before invoking op");
        res.op();
        log("after invoking op");
    }

    private void log(String msg) {
        System.out.println("[ " + Thread.currentThread() + " ] " + msg);
    }
}
```

```
public class Resource {
    public synchronized void op() {
        System.out.println("[Resource] Thread " + Thread.currentThread() + " entered.");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {}
        System.out.println("[Resource] - Thread " + Thread.currentThread() + " exited.");
    }
}
```

TEST

- Nel test creiamo due thread (di classe ResourceUser) accedono concorrentemente ad un oggetto condiviso di classe Resource, invocando il metodo synchronized op

```
package oop.concur;

public class TestResourceUsers {
    public static void main(String[] args) {
        Resource res = new Resource();
        ResourceUser userA = new ResourceUser("pippo", res);
        ResourceUser userB = new ResourceUser("pluto", res);
        userA.start();
        try {
            Thread.sleep(500);
        } catch (Exception ex) {
        }
        userB.start();
    }
}
```

```
[Thread[pippo,5,main]] before invoking op
[Resource] Thread Thread[pippo,5,main] entered.
[Thread[pluto,5,main]] before invoking op
[Resource] - Thread Thread[pippo,5,main] exited.
[Thread[pippo,5,main]] after invoking op
[Resource] Thread Thread[pluto,5,main] entered.
[Resource] - Thread Thread[pluto,5,main] exited.
[Thread[pluto,5,main]] after invoking op
```

- Eseguendo il test è possibile verificare come che userB riesce ad entrare nel metodo op solo quando userA è uscito

CLASSI NON THREAD-SAFE

- L'accesso concorrente ad oggetti passivi da parte di più thread può essere causa di **corse critiche** e quindi malfunzionamenti del programma
 - una classe potrebbe avere un comportamento corretto se usata in contesti sequenziali, mentre dare origine a problemi se usata in contesti MULTITHREAD
 - classe **non thread-safe**
- Esempio semplice nel materiale: contatore condiviso
 - classico contatore, usato concorrentemente da 2 thread (CounterUser) che lo incrementano N volte ciascuno
 - nel caso in cui il contatore non sia thread-safe (UnsafeCounter), al termine della computazione da parte dei due thread il contatore può non avere il valore corretto che ci si aspetta (ovvero 2N)
 - non-determinismo

CORSE CRITICHE: ESEMPIO

```
package oop.concur;

public class CounterUser extends Thread {
    private long ntimes;
    private Counter counter;

    public CounterUser(String name, Counter c, long n) {
        super(name);
        ntimes = n;
        counter = c;
    }

    public void run() {
        log("starting - counter value is " + counter.getValue());
        for (long i = 0; i < ntimes; i++) {
            counter.inc();
        }
        log("completed - counter value is " + counter.getValue());
    }

    private void log(String msg) {
        System.out.println("[COUNTER USER " + getName() + "] " + msg);
    }
}
```

CORSE CRITICHE: ESEMPIO

- Lancio di due thread (istanze della classe CounterUser)

```
package oop.concur;
public class TestConcurrentCounter {
    public static void main(String[] args) throws Exception {
        Counter c = new UnsafeCounter();
        long ntimes = Long.parseLong(args[0]);
        CounterUser agentA = new CounterUser(c, ntimes);
        CounterUser agentB = new CounterUser(c, ntimes);
        agentA.start();
        agentB.start();
        agentA.join();
        agentB.join();
        System.out.println("Count value: " + c.getValue());
    }
}
```

A causa di corse critiche, il valore finale può non essere corretto (non determinismo) - più è grande il valore, più è probabile non sia corretto

```
% java -cp bin oop.concur.MainConcurrentUnsafeCounter 10000
[COUNTER USER agent-B] starting - counter value is 0
[COUNTER USER agent-A] starting - counter value is 0
[COUNTER USER agent-B] completed - counter value is 10270
[COUNTER USER agent-A] completed - counter value is 11513
Count value: 11513
```

SOLUZIONE: CLASSI **THREAD-SAFE**

- Una classe si dice ***thread-safe*** quando mantiene il proprio comportamento corretto anche se usata in un contesto multithread
- Per rendere thread-safe una classe ed evitare le corse critiche appena viste, è sufficiente fare in modo che l'accesso all'oggetto (mediante l'esecuzione di un qualsiasi metodo pubblico) avvenga in modo *mutuamente esclusivo*
- Un modo semplice per farlo in Java: *dichiarare tutti i metodi pubblici synchronized*

```
package oop.concur;

public class SafeCounter {
    private int cont;

    public SafeCounter () { cont = 0; }

    public synchronized void inc(){
        cont++;
    }

    public synchronized void dec(){
        cont--;
    }

    public synchronized int getValue(){
        return cont;
    }
}
```

ESEMPIO DEL CONTATORE THREAD-SAFE

- Lancio di due thread (istanze della classe CounterUser)

```
package oop.concur;

public class TestConcurrentCounter {
    public static void main(String[] args) throws Exception {
        Counter c = new SafeCounter();
        long ntimes = Long.parseLong(args[0]);
        CounterUser agentA = new CounterUser(c, ntimes);
        CounterUser agentB = new CounterUser(c, ntimes);
        agentA.start();
        agentB.start();
        agentA.join();
        agentB.join();
        System.out.println("Count value: " + c.getValue());
    }
}
```

In questo caso il valore finale è sempre pari a $2N$.

```
% java -cp bin oop.concur.MainConcurrentSafeCounter 100000000
[COUNTER USER agent-B] starting - counter value is 0
[COUNTER USER agent-A] starting - counter value is 0
[COUNTER USER agent-B] completed - counter value is 199362753
[COUNTER USER agent-A] completed - counter value is 200000000
Count value: 200000000
```

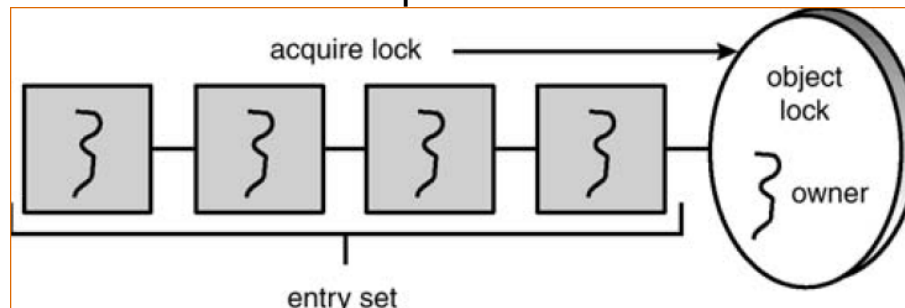
COSTO DELLA THREAD-SAFETY

```
% java -cp bin oop.concur.TestConcurrentUnsafeCounterWithTime 100000000  
[COUNTER USER agent-B] starting - counter value is 0  
[COUNTER USER agent-A] starting - counter value is 0  
[COUNTER USER agent-B] completed - counter value is 100011617  
[COUNTER USER agent-A] completed - counter value is 100011617  
Count value: 100011617  
Time elapsed: 61ms.
```

```
% java -cp bin oop.concur.TestConcurrentSafeCounterWithTime 100000000  
[COUNTER USER agent-B] starting - counter value is 0  
[COUNTER USER agent-A] starting - counter value is 0  
[COUNTER USER agent-A] completed - counter value is 199783206  
[COUNTER USER agent-B] completed - counter value is 200000000  
Count value: 200000000  
Time elapsed: 4798ms.
```


METODI SYNCHRONIZED: FUNZIONAMENTO

- A livello di implementazione il meccanismo di coordinazione synchronized è realizzato mediante un lock associato nativamente ad ogni oggetto.
 - l'esecuzione di un metodo synchronized comporta prima l'acquisizione del lock:
 - se il lock è già posseduto da un altro thread, il richiedente è bloccato / sospeso e inserito nella lista dei thread in attesa, chiamata entry set del lock.
 - se il lock è disponibile invece, il thread diviene proprietario del lock dell'oggetto ed esegue il metodo.
 - quando il proprietario rilascia il lock - o perché ha terminato l'esecuzione del metodo o perché deve esser sospeso, in seguito all'esecuzione di una wait - se l'entry set non è vuota, viene rimosso uno thread e ad esso viene assegnato il lock, ripristinandone l'esecuzione.
- La JVM non specifica alcuna politica di gestione dell'entry set
 - tipicamente viene utilizzata una politica FIFO



BLOCCHI synchronized

- Il meccanismo synchronized può essere utilizzato anche a livello di blocchi di codice, con una granularità quindi più fine rispetto al caso dei metodi

```
AnyClass myLockObject;  
...  
synchronized (myLockObject){  
    <statements>  
}
```

- Semantica del costrutto
 - prima di eseguire le istruzioni specificate all'interno dello blocco (statements), viene acquisito il lock sull'oggetto specificato
 - nel caso in cui il lock sia già stato acquisito, il thread corrente viene sospeso nell'entry set dell'oggetto specificato (myLockObject)
 - al termine dell'esecuzione delle istruzioni, in uscita dal blocco synchronized, viene automaticamente rilasciato il lock

SEZIONI CRITICHE CON BLOCCHI SYNCHRONIZED

- Nell'esempio, due thread utilizzano un blocco synchronized su un oggetto condiviso per realizzare due sezione critiche

```
class MyWorkerA extends Thread {
    private Object lock;
    public MyWorkerA(Object lock){
        this.lock = lock;
    }
    public void run(){
        while (true){
            System.out.println("a1");
            synchronized(lock){
                System.out.println("a2");
                System.out.println("a3");
            }
        }
    }
}
```

```
class MyWorkerB extends Thread {
    private Object lock;
    public MyWorkerB(Object lock){
        this.lock = lock;
    }
    public void run(){
        while (true){
            synchronized(lock){
                System.out.println("b1");
                System.out.println("b2");
            }
            System.out.println("b3");
        }
    }
}
```

```
public class TestCS {
    public static void main(String[] args) {
        Object lock = new Object();
        new MyWorkerA(lock).start();
        new MyWorkerB(lock).start();
    }
}
```

SINCRONIZZAZIONE ESPLICITA

- Metodi **wait**, **notify**, **notifyAll**
 - metodi pubblici definiti nella root class `java.lang.Object`
 - quindi ereditati da qualsiasi nuova classe che scriviamo
- Semantica
 - wait
 - l'invocazione della `wait` provoca la sospensione del thread corrente (con rilascio del lock sull'oggetto), fino a quando un altro thread non esegua una `notify` o `notifyAll` sul medesimo oggetto
 - i thread sospesi vengono incluso in un insieme chiamato wait set
 - notify
 - provoca il risveglio di uno degli eventuali thread sospesi sul medesimo oggetto con la `wait`
 - notifyAll
 - provoca il risveglio di tutti i thread sospesi con la `wait`
- **NOTA IMPORTANTE:** *Per poter chiamare uno qualsiasi di questi metodi è necessario avere prima ottenuto il lock sull'oggetto*
 - tipicamente vengono chiamati all'interno di metodi `synchronized`

BUONA PRATICA DI UTILIZZO: “MONITOR” PATTERN

- **Monitor pattern**
 - proprietà di **mutua esclusione**: 1 solo thread alla volta può essere in esecuzione di metodi del monitor
 - sincronizzazione fra metodi: uso di meccanismi di sincronizzazione (**condition variables**) per sincronizzare metodi dello stesso oggetto
- Due possibili schemi implementativi
 - mediante costrutti base: synchronized + wait/notify/notifyAll
 - una classe con tutti i metodi pubblici synchronized
 - quindi esecuzione metodi mutuamente esclusiva
 - uso di wait/notify/notifyAll all'interno dei metodi synchronized per sincronizzare i diversi flussi di controllo/thread che hanno chiamato i metodi diversi
 - mediante classi fornite a livello di libreria (java.util.concurrent)
 - classe ReentrantLock - per la mutua esclusione
 - oggetti Condition - come condition variable

ESEMPIO: SAFE COUNTER MONITOR

- Esempio già visto - SafeCounter è un monitor

```
package oop.concur;

public class Counter {
    private int cont;

    public Counter () { cont = 0; }

    public synchronized void inc(){
        cont++;
    }

    public synchronized void dec(){
        cont--;
    }

    public synchronized int getValue(){
        return cont;
    }
}
```

UN SEMPLICE “SINCRONIZZATORE”

- Esempio di semplice “sincronizzatore”
 - per sincronizzare azioni di thread distinti
 - costruito come monitor

```
public class SimpleSynchronizer {  
    private boolean signalArrived;  
  
    public SimpleSynchronizer() {  
        signalArrived = false;  
    }  
  
    public synchronized void waitForSignal() throws InterruptedException {  
        while (!signalArrived) {  
            this.wait();  
        }  
        signalArrived = false;  
    }  
  
    public synchronized void signal() {  
        signalArrived = true;  
        notifyAll();  
    }  
}
```

PRIMO ESEMPIO: SYNCH FRA AZIONI

- TestSynchUsers
 - thread SynchUserA e SynchUserB sincronizzati
 - azione b2 di SynchUserB deve avvenire dopo azione a2 di SynchUserA

```
public class SynchUserA extends Thread {  
    private SimpleSynchronizer sync;  
  
    public SynchUserA(SimpleSynchronizer sync) {  
        this.sync = sync;  
    }  
  
    public void run() {  
        a1();  
        a2();  
        sync.signal();  
        a3();  
    }  
    ...  
}
```

```
public class SynchUserB extends Thread {  
    private SimpleSynchronizer sync;  
  
    public SynchUserB(SimpleSynchronizer sync) {  
        this.sync = sync;  
    }  
  
    public void run() {  
        b1();  
        try {  
            sync.waitForSignal();  
        } catch (Exception ex) {}  
        b2();  
        b3();  
    }  
    ...  
}
```


SECONDO ESEMPIO: PING PONG

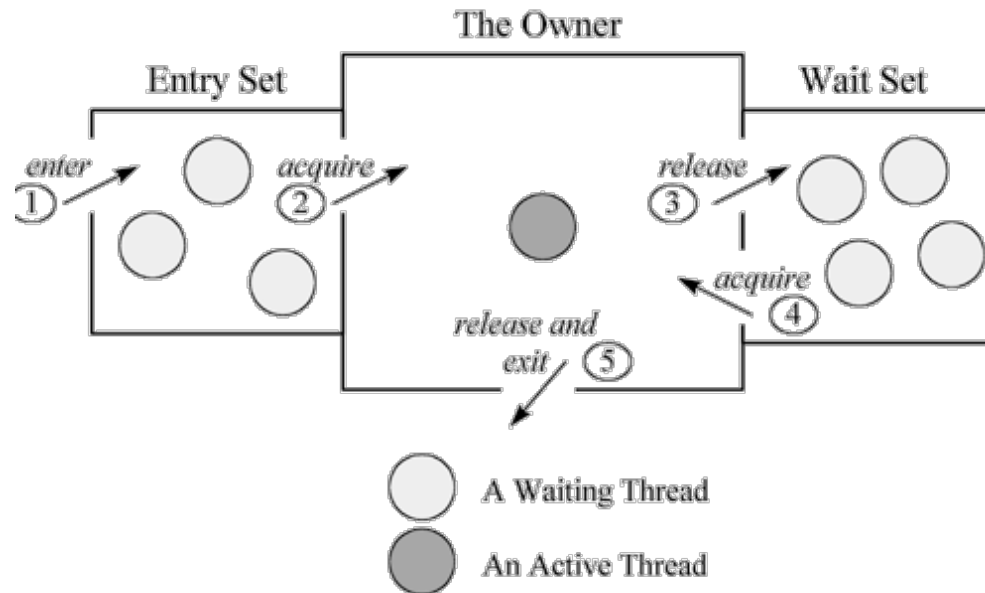
- TestPingPong
 - due thread (Pinger e Ponger) usano due synch per alternarsi nelle azioni (di ping e pong)

```
public class Pinger extends Thread {  
    private SimpleSynchronizer mySync, otherSync;  
  
    public Pinger(SimpleSynchronizer mySync,  
                  SimpleSynchronizer otherSync) {  
        this.mySync = mySync;  
        this.otherSync = otherSync;  
    }  
  
    public void run() {  
        try {  
            while (true) {  
                mySync.waitForSignal();  
                log("ping");  
                otherSync.signal();  
            }  
        } catch (Exception ex) {}  
    }  
    ...  
}
```

```
public class Ponger extends Thread {  
    private SimpleSynchronizer mySync, otherSync;  
  
    public Ponger(SimpleSynchronizer mySync,  
                  SimpleSynchronizer otherSync) {  
        this.mySync = mySync;  
        this.otherSync = otherSync;  
    }  
  
    public void run() {  
        try {  
            while (true) {  
                mySync.waitForSignal();  
                log("pong");  
                otherSync.signal();  
            }  
        } catch (Exception ex) {}  
    }  
    ...  
}
```

FUNZIONAMENTO WAIT/NOTIFY: ENTRY SET E WAIT SET

- Step
 - invocazione metodo synchronized
 - si ottiene il lock
 - esecuzione di una wait, con rilascio del lock
 - notifica da parte del thread attivo e ottenimento del lock
 - uscita dal metodo synchronized



LIBRERIA `java.util.concurrent`

- La libreria `java.util.concurrent` include un ricco insieme di costrutti di alto livello per semplificare lo sviluppo di applicazioni concorrenti, tra cui:
 - **Concurrent Collections**
 - implementazione efficiente e thread-safe di strutture dati come liste, mappe, code, stack da utilizzare in contesti concorrenti
 - **Synchronizers**
 - implementazione di meccanismi e costrutti classici per la coordinazione di thread (semaphores, latches, barriers,....)

CONCURRENT COLLECTIONS

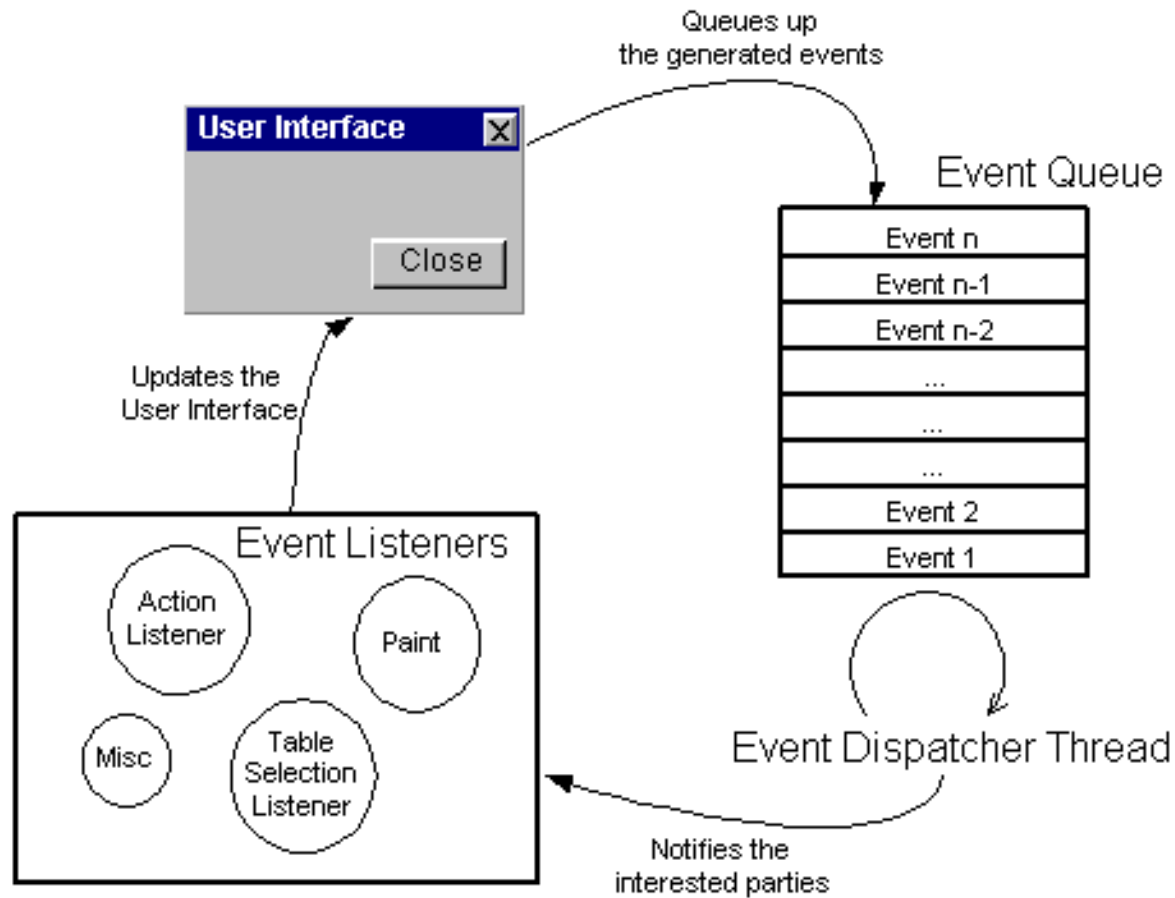
- Implementazione delle collections di Java appositamente pensata per essere efficace nel caso di accessi concorrenti da più thread
 - thread-safe
 - ottimizzata in modo da minimizzare le parti sequenziali
- Fra le classi principali:
 - **ConcurrentHashMap**
 - versione concorrente delle hash map
 - **Queue and BlockingQueue**
 - interfacce che rappresentano code, con diverse implementazioni
 - alcune sono utili per implementare direttamente dei bounded-buffer
 - **CopyOnWriteArrayList**
 - versione concorrente di ArrayList

SYNCHRONIZERS

- Nel terminologia adottata dalla libreria, un synchronizer è un qualsiasi oggetto passivo che serve per coordinare il flusso di controllo dei thread che vi interagiscono
- Sono i componenti di base che incapsulano funzionalità di coordinazione classiche, utili in tutte le applicazioni
- Tipi principali inclusi nella libreria:
 - **Locks**
 - **Semaphores**
 - **Latches**
 - **Barriers**
- Proprietà generali synchronizer:
 - incapsulano uno stato che determina quando il thread che li invoca può proseguire oppure essere bloccato
 - forniscono metodi per manipolare tale stato
 - forniscono metodi che permettono di attendere in modo efficiente il verificarsi di tale stato - eventualmente bloccando il flusso di controllo del thread chiamante

MULTITHREADING E INTERFACCE GRAFICHE - SECONDA PARTE

RICHIAMI: GUI EVENT LOOP



[Figura tratta da <https://www.javaworld.com/article/2073477/swing-gui-programming/customize-swingworker-to-improve-swing-guis.html>]

RICHIAMI: DESIDERATA NELLO SVILUPPO DI GUI

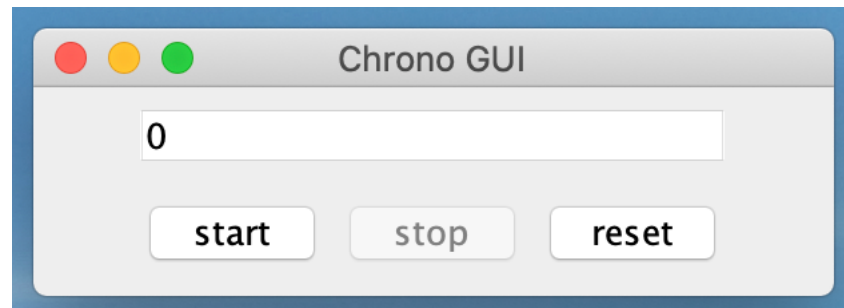
1. Avere GUI reattive/responsive (prima parte)
- 2. Avere GUI senza corse critiche**

COME REALIZZARE GUI SENZA CORSE CRITICHE

- I componenti della GUI sono tutti implementati con classi non thread-safe, poiché si presuppone che l'unico thread che vi acceda sia quella della GUI
 - Per cui, per avere una **GUI senza corse critiche**, è necessario delegare al thread della GUI qualsiasi compito che riguardi la modifica di stato di un qualsiasi componente della GUI
- Come farlo
 - metodi statici `SwingUtilities.invokeLater` e `SwingUtilities.invokeAndWait`
 - entrambi accodano il task specificato come argomento (un `Runnable`) alla coda degli eventi del thread della GUI, come nuovo task da eseguire
 - il secondo metodo oltre ad accodare blocca il thread chiamante fin quando il compito è stato eseguito dal thread della GUI
- Esempio sul repo:
 - `oop.concur.gui_reactive_and_safe`

UN ESEMPIO COMPLETO: COSTRUIAMO UN CRONOMETRO...

- Problema
 - si vuole pilotare un conteggio mediante una interfaccia grafica dotata di tre pulsanti (start, stop, reset) e un edit box dove visualizzare il conteggio
 - alla pressione di start, il conteggio deve partire e venire incrementato ogni decimo di secondo, alla pressione di stop il conteggio si ferma e reset lo riporta a zero



ARCHITETTURA BASATA SU MVC + THREAD

- Adottiamo un'architettura basata sul pattern model view controller (MVC) estesa per integrare opportunamente i thread per mantenere la reattività dell'interfaccia
 - Model
 - ChronoCount
 - View
 - ChronoView
 - ChronoFrame (Swing specific)
 - Controller
 - Controller (interfaccia) e ChronoController (impl.)
 - ChronoAgent (thread)
- Due soluzioni proposte
 - nella prima soluzione (oop.concur.chrono) la responsabilità dell'aggiornamento della View è a carico del controller
 - nella seconda soluzione (oop.concur.chrono2), viene applicato in modo completo il pattern MVC e la View funge da osservatore del model

OLTRE LA PROGRAMMAZIONE MULTITHREAD

OLTRE I THREAD...

- La programmazione con thread è l'approccio alla programmazione concorrente più “antico” e di basso livello. Nel tempo sono stati studiati e introdotti altri approcci, sia nella pratica, sia in ambito di ricerca.
- A seguire in questo modulo se ne considerano alcuni ampiamente utilizzati nello sviluppo di sistemi in Java. In particolare:
 - Approccio a **task** con **Executors**
 - **Virtual Threads**
 - **Attori**

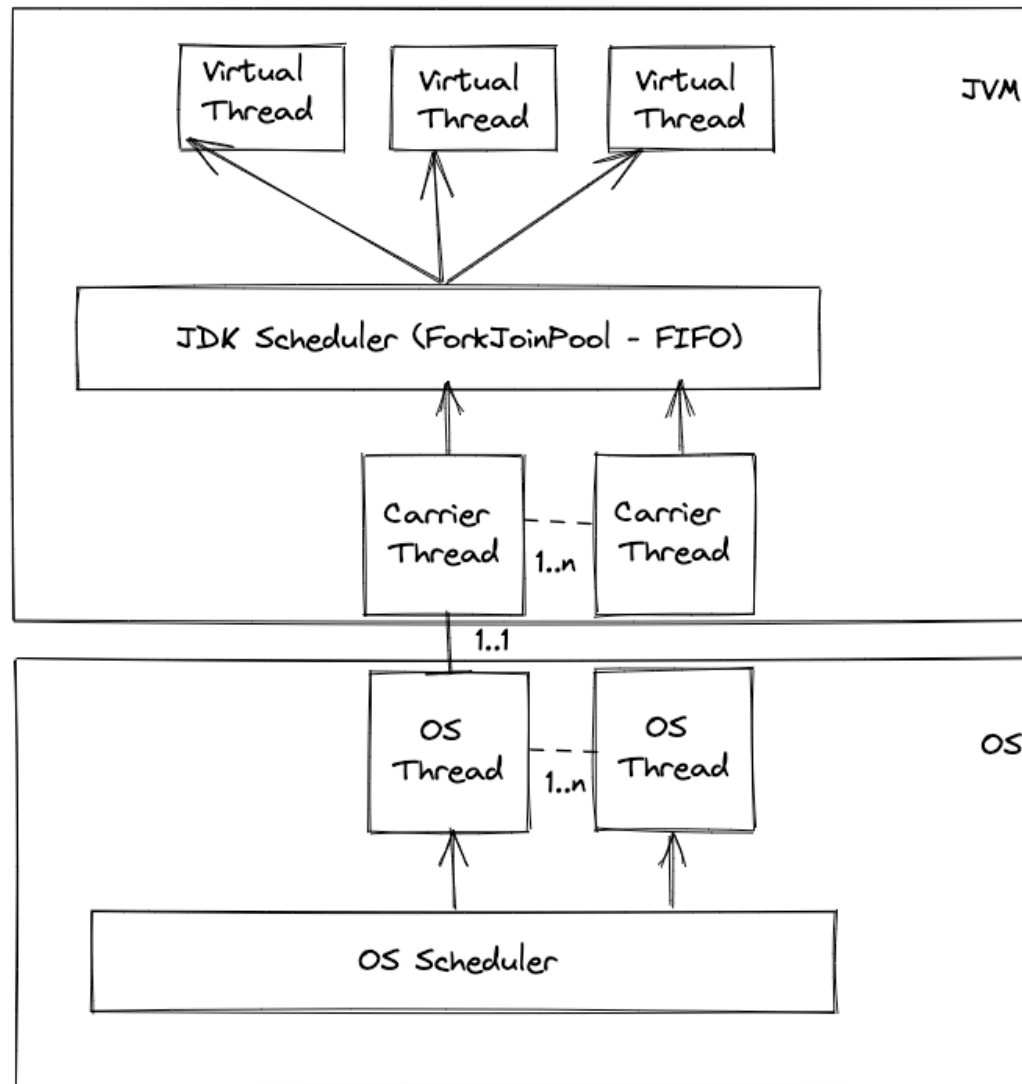
DAI THREAD FISICI AI TASK

- In fase di analisi di un problema e progettazione, tipicamente si individuano i compiti o task con cui suddividere il lavoro, come attività singole che possano essere eseguite autonomamente
 - principio di decomposizione *logica* che fa riferimento alla natura del problema, indipendentemente dai dettagli fisici relativi all'hardware concorrente a disposizione (es: numero core)
- Il framework **Executors** in Java è stato introdotto già a partire dalla versione JDK1.5 per supportare questo livello
 - permette ad uno sviluppatore/sviluppatrice di definire i task indipendenti con cui decomporre un problema
 - classi che implementano interfacce `Runnable` o `Callable`
 - L'esecuzione dei task viene delegata ad un esecutore (executor) che incapsula le strategie usate per mappare i vari task su un pool di 1 o più thread
- Tipi diversi di executor possono implementar strategie diverse..
 - `FixedThreadPool`, `CachedThreadPool`, `SingleThreadPool`,...
- Esempio sul repo:
 - `oop.concur.ext.TestExecutors`

VIRTUAL THREAD

- “Implementazione leggera” dei thread (user thread), così definiti logici o *virtuali*
 - gestiti parzialmente dalla JVM, come semplici oggetti con footprint di memoria (stack) dimensionabile dinamicamente di creare e mandare in esecuzione
 - disponibili in modo stabile a partire dal JDK 21
- Idea di base: usare thread fisici come *carrier* dove la JVM possa in modo opportuno “montare” e “smontare” thread virtuali all’occorrenza, in modo efficiente, evitando overhead del context switch
 - possibilità di creare e mandare in esecuzione un numero molto elevato di thread (virtuali), eseguiti da un pool di thread fisici
- Aspetto importante
 - **la JVM riesce a smontare e montare thread virtuali solo se/ quando richiedono operazioni bloccanti**
 - nell’implementazione corrente per alcune operazioni bloccanti della JVM il meccanismo non funziona - i virtual thread sono *pinned* ai carrier thread [*]

VIRTUAL THREAD: FUNZIONAMENTO



VIRTUAL THREAD API

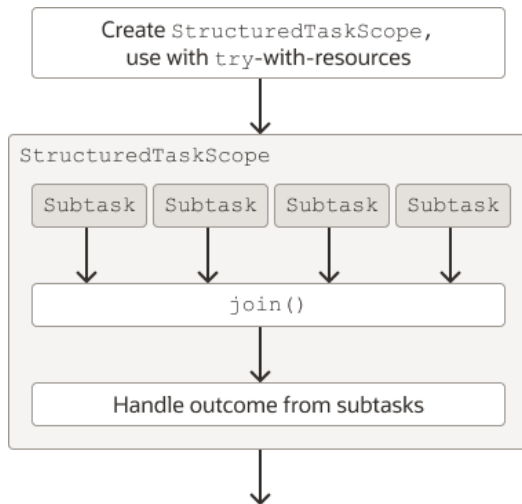
- API pressoché invariata rispetto a `java.lang.Thread`
 - quindi semanticamente uguali ai Thread “fisici”
- Ciò che cambia è l’API con cui vengono creati
 - `Thread.ofVirtual()`
- Esempio sul repo:
 - `oop.concur.ext.TestVirtualThreads`

IMPORTANZA DEI VIRTUAL THREAD

- I virtual thread permettono di *disaccoppiare* i thread come costruito a livello *logico* per progettare e realizzare programmi concorrenti da aspetti e vincoli *fisici*, legati al sistema operativo e all'HW della macchina
 - possiamo usarli direttamente per implementare un *task logico* individuato in fase di progettazione, oppure un *componente attivo* (“agente”) a cui sono assegnati uno o più task

STRUCTURED CONCURRENCY

- Obiettivo: dare struttura alla programmazione multithread, permettendo di trattare *gruppi* di task correlati in esecuzione in thread diversi come una singola unità di lavoro
 - per la gestione di errori, cancellazioni, migliorando robustezza e osservabilità
 - pattern fork-join



```
Callable<String> task1 = ...  
Callable<Integer> task2 = ...
```

```
try (var scope = new StructuredTaskScope<Object>()) {  
    Subtask<String> subtask1 = scope.fork(task1);  
    Subtask<Integer> subtask2 = scope.fork(task2);  
    scope.join();  
    ... process results/exceptions ...  
} // close
```

- Feature in preview

<https://docs.oracle.com/en/java/javase/21/core/structured-concurrency.html>

OLTRE ALLA PROGRAMMAZIONE MULTITHREAD

- In letteratura e nella pratica esistono vari approcci alternativi alla programmazione multithread, introdotti nel corso del tempo
 - studiati alla magistrale
- Fra i principali esempi
 - **Programmazione asincrona**
 - modelli ad eventi / event-loop
 - Java: Framework Vert.x, Javascript: node.js
 - **Programmazione reattiva**
 - modelli funzionali, data-driven e reattivi
 - Framework Reactive X - in Java: RxJava
 - **Programmazione ad attori**
 - oggetti + concorrenza
 - scambio asincrono di messaggi

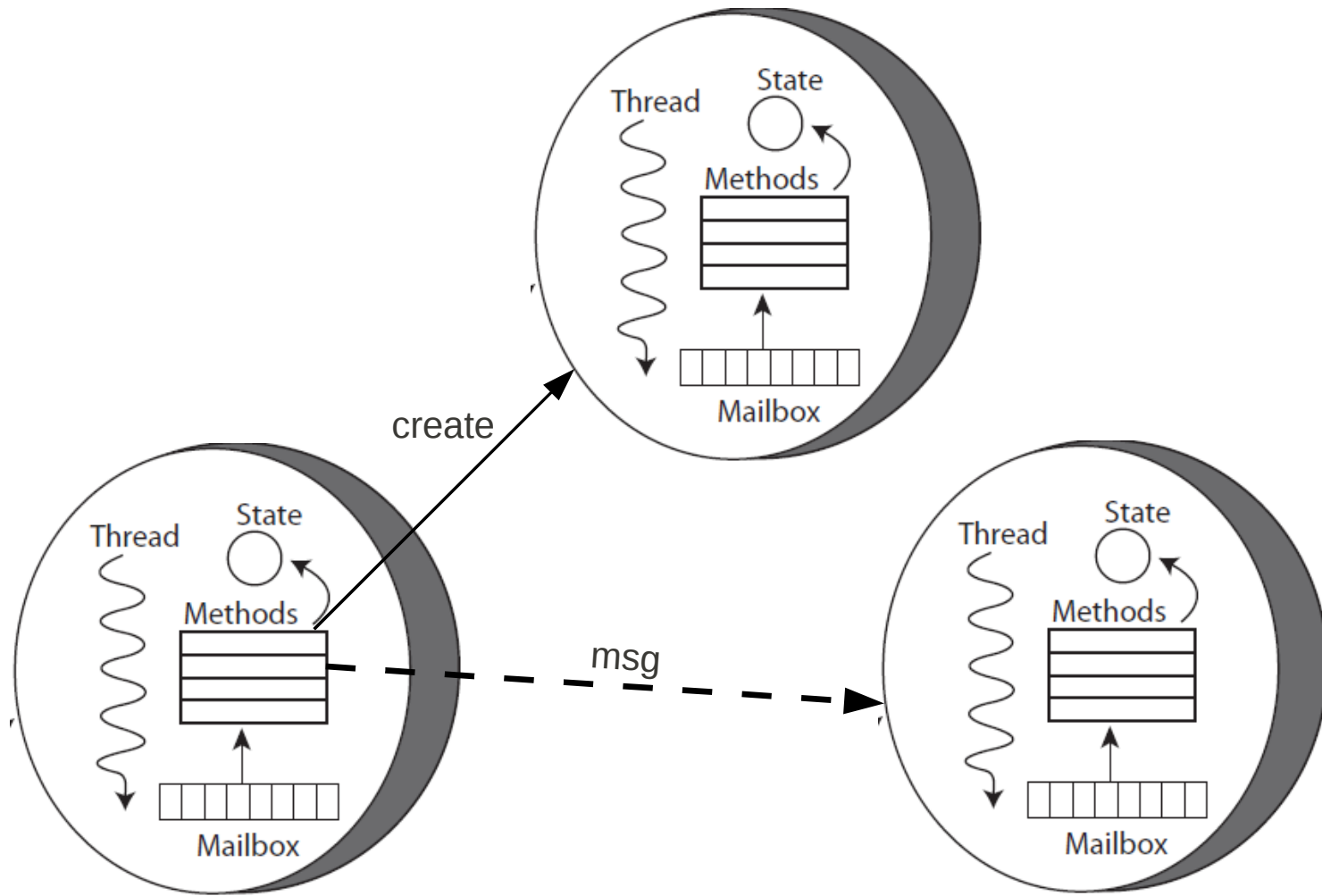
PROGRAMMAZIONE AD ATTORI

- Punto chiave del modello ad attori: interazione esclusivamente basata su **scambio asincrono di messaggi**
 - no memoria condivisa, no operazioni bloccanti
- Concetto/astrazione di **attore**
 - entità attiva o autonoma
 - *dotata di un proprio flusso di controllo logico*
 - reattiva
 - computa quando riceve un messaggio, mandando in esecuzione l'handler associato
 - esecuzione degli handler atomica
- Concettualmente:

ATTORI = OGGETTI + CONCORRENZA

- Vari framework e linguaggi ad attori disponibili
 - es: Framework Akka (akka.io), ActorFoundry (academics)

PROGRAMMAZIONE AD ATTORI



OOP-CONCUR

PROG. MULTI-THREAD IN JAVA

BIBLIOGRAFIA PER APPROFONDIMENTI

- Java Concurrency in Practice - Brian Goetz et al - Addison Wesley
- Concurrent Programming in Java: Design Principles and Pattern, 2/E
- Doug Lea - Addison-Wesley Professional
 - testo di riferimento per la programmazione concorrente in Java
- Corso Programmazione Concorrente e Distribuita - Laurea
Magistrale in Ingegneria e Scienze Informatiche - Università di
Bologna, Campus di Cesena