



Agent and Object Technology Lab  
Dipartimento di Ingegneria dell'Informazione  
Università degli Studi di Parma



Ingegneria del software A

Programmazione concorrente

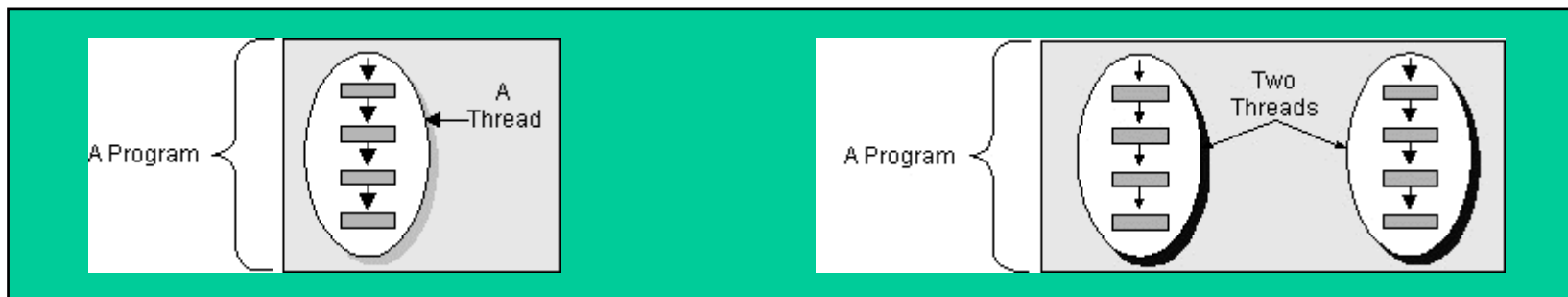
**Prof. Agostino Poggi**

- ◆ **Single-tasking**
  - Primi sistemi: un job alla volta, modalità batch
- ◆ **Multi-tasking**
  - Più processi in esecuzione, contemporaneamente
  - Sia applicazioni utente (word-processor, browser, posta...)
  - .. che processi di sistema
- ◆ **Multi-threading**
  - Sistemi più recenti: più flussi di esecuzione nel contesto di uno stesso processo
  - Una applicazione può eseguire più compiti
    - Un browser scarica diversi file, mentre stampa una pagina...
    - Una applicazione server può gestire più richieste in parallelo

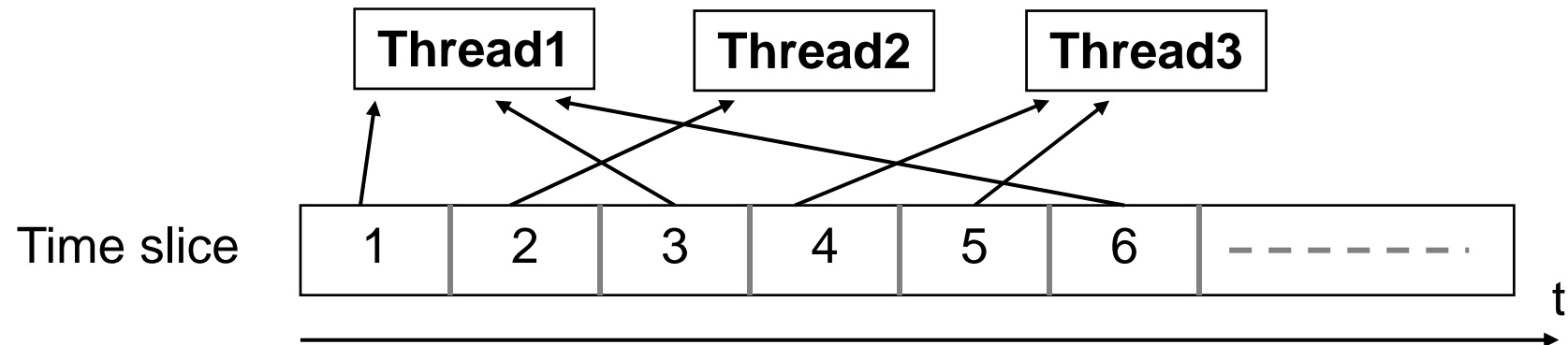
- ◆ Un sistema di elaborazione normalmente ha diversi processi e thread attivi
- ◆ Caso in cui ci sia solo una unità di elaborazione
  - Il parallelismo di processi e thread viene simulato (*time slicing*)
  - Il tempo di elaborazione è suddiviso tra processi e/o thread
  - Allo scadere di ogni unità di tempo, il s.o. opera un cambio di contesto (*context switch*)
- ◆ Sempre più comune avere sistemi paralleli
  - Processori multipli, o *core* multipli
  - In ogni istante di tempo, ci possono essere più processi o thread fisicamente in esecuzione

- ◆ Ambiente di esecuzione completo
  - Insieme completo e privato di risorse run-time di base
  - In particolare, proprio spazio di memoria
- ◆ Spesso sinonimo di programma o applicazione
  - Ma una applicazione può essere un insieme di processi cooperanti
- ◆ Meccanismi *IPC (Inter Process Communication)*
  - Forniti dai s.o. per facilitare la comunicazione tra processi
  - *Pipe* – Processi sulla stessa macchina
  - *Socket* – Anche su macchine diverse
- ◆ Virtual machine implementata come singolo processo
  - La classe `ProcessBuilder` permette di lanciare altri process

- ◆ Singolo flusso di controllo sequenziale all'interno di un programma
- ◆ Definito anche processo leggero (*lightweight process*)
  - Sia i processi che i thread forniscono un ambiente di esecuzione
  - Creazione e switch di thread richiedono meno risorse e tempo
- ◆ I thread esistono all'interno di un processo: ogni processo ne ha almeno uno
  - I thread condividono le risorse del processo (il suo contesto), compresa la memoria e i file aperti
  - Ma hanno il proprio stack, il proprio program counter...
  - Comunicazione efficiente ma potenzialmente problematica



- ◆ Sistema *non preemptive*
  - Cambio di contesto solo quando il thread in esecuzione:
    1. Interrompe la propria esecuzione volontariamente
    2. O si pone in attesa di un evento
  - Windows 3.1 è un sistema non preemptive
  
- ◆ Sistema *preemptive*
  - Allo scadere del time slice la piattaforma interrompe forzatamente il thread e opera il cambio di contesto
  - La *schedulazione*, ossia la scelta del thread da attivare, può avvenire secondo diversi algoritmi
    - Round-robin, con priorità...



- ◆ Un programma Java non dovrebbe dipendere dall'algoritmo di schedulazione sottostante
- ◆ Il flusso di esecuzione può essere controllato con vari meccanismi di *sincronizzazione*

- ◆ Esecuzione multi-threaded: caratteristica essenziale della piattaforma Java
- ◆ Ogni applicazione ha diversi thread “di sistema”
  - Gestione della memoria e gestione degli eventi
- ◆ Il lavoro inizia sempre con un solo thread: il thread principale
  - Possibilità per il programmatore di creare altri thread

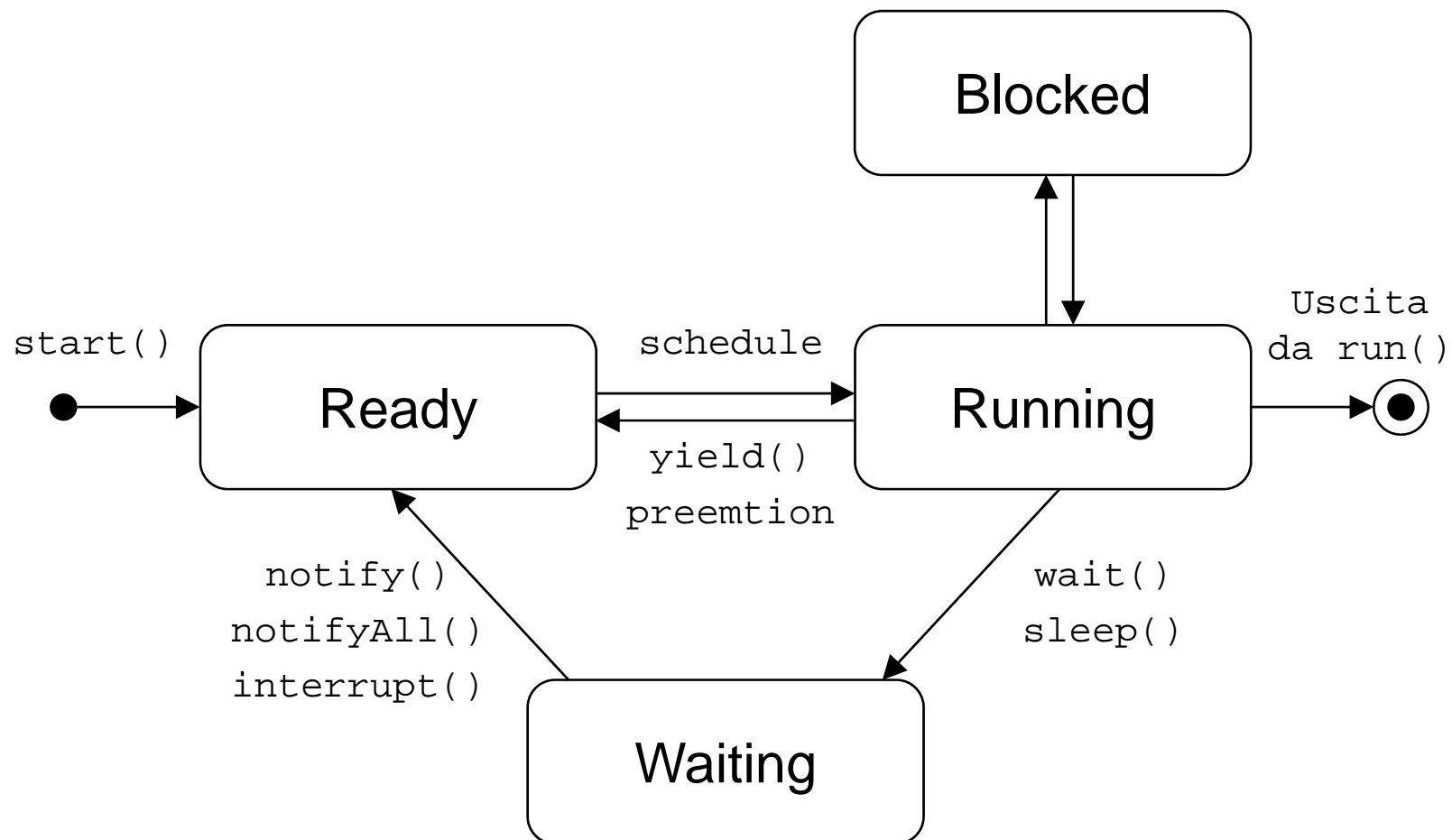


- ◆ Ogni thread è associato ad una istanza di Thread
- ◆ Due diverse strategie per creare una applicazione concorrente usando oggetti thread:
  1. Controllo diretto di creazione e gestione del thread
    - Semplicemente si istanzia Thread ogni volta che l'applicazione deve iniziare un compito asincrono
  2. Passaggio dei compiti dell'applicazione ad un esecutore
    - Si astrae la gestione dei thread dal resto dell'applicazione

- ◆ La gestione di thread da Java è indipendente dalla piattaforma sottostante
  - La macchina virtuale può appoggiarsi ai thread nativi
  - O può simularli
  
- ◆ Ogni thread è caratterizzato da:
  - Un corpo (*thread body*)
  - Uno stato (*thread state*)
  - Un gruppo di appartenenza (*thread group*)
  - Una priorità (*thread priority*)

- ◆ Per specificare il corpo di un thread, bisogna scrivere l'implementazione del suo metodo `run`
- ◆ Un thread viene avviato invocando il metodo `start`
  - In seguito, la macchina virtuale genera la biforcazione del flusso
  - Ad un certo punto lo scheduler eseguirà il metodo `run`

*Non bisogna invocare direttamente `run`*



- ◆ Bisogna fornire il codice da eseguire nel secondo thread
- ◆ Due maniere:

## 1. Realizzare un oggetto con interfaccia Runnable

- Runnable definisce il solo metodo `run`, che contiene il codice...
- L'oggetto Runnable viene passato al costruttore di Thread

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        Runnable hello = new HelloRunnable();  
        Thread t = new Thread(hello);  
        t.start();  
    }  
}
```

## 2. Realizzare una sottoclasse di Thread

- Thread implementa Runnable, ma il metodo `run` non fa niente
- Bisogna fornire l'implementazione di `run`

*Nota: entrambi gli esempi invocano `Thread.start` per avviare il nuovo thread*

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

- ◆ Quale dei due idiomi usare?
- ◆ Secondo idioma più facile da usare in app semplici
  - Limitato dal fatto che la classe del task deve discendere da `Thread`
- ◆ L'uso di un oggetto `Runnable` è più generale e flessibile: l'oggetto può estendere qualsiasi classe
  - Applicabile anche alle API di alto livello per la gestione di thread
- ◆ La classe `Thread` definisce vari metodi utili per la gestione di thread
  - Dei metodi di classe forniscono informazioni sul thread chiamante, o ne modificano lo stato
  - Dei metodi di istanza sono invocati per gestire l'oggetto thread chiamato

- ◆ `Thread.sleep` provoca la sospensione dell'esecuzione del thread corrente per un periodo specificato
  - Rendere il tempo del processore disponibile ad altri thread dell'applicazione, o di altre applicazioni che girano su un computer
  - Usato per fare operazioni periodiche
  - O aspettare un altro thread che svolge compiti con noti requisiti di tempo
  
- ◆ Due versioni (overloaded) di `sleep`
  - Una specifica il tempo in millisecondi
  - L'altro in nanosecondi
  
  - Non c'è garanzia sulla loro precisione
  - Limitate dalle caratteristiche del S.O. sottostante
  - Il periodo di “sonno” può essere terminato da interruzioni
  - In ogni caso, non si può assumere che il thread resterà sospeso precisamente per il periodo di tempo specificato



```
public class SleepMessages {  
    public static void main(String args[])  
        throws InterruptedException {  
        String importantInfo[] = {  
            "Mares eat oats",  
            "Does eat oats",  
            "Little lambs eat ivy",  
            "A kid will eat ivy too"  
        };  
        for (int i = 0; i < importantInfo.length; i++) {  
            //Pause for 4 seconds  
            Thread.sleep(4000);  
            //Print a message  
            System.out.println(importantInfo[i]);  
        }  
    }  
}
```

- ◆ Il metodo `main` dichiara `InterruptedException`
  - Quando un altro thread interrompe il thread attuale, l'operazione corrisponde a lanciare una `InterruptedException`
  - Nell'esempio precedente, non c'è bisogno di gestire l'eccezione, perché non c'è un thread che causi l'interruzione
- ◆ Un thread può interrompere un altro thread invocando il metodo `interrupt` sull'oggetto thread da interrompere

- ◆ In caso di interruzione, il thread dovrebbe smettere di fare quello che sta facendo e gestire l'interruzione
- ◆ In che modo bisogna gestire (reagire ad) una interruzione?
  - Spesso il thread termina (consigliato)
  - Ma è il programmatore a decidere la semantica

- ♦ I metodi che lanciano `InterruptedException` in genere escono da run dopo aver catturato l'eccezione
  - Ad esempio il metodo, `sleep`, in caso di interruzione lancia una `InterruptedException`

```
for (int i = 0; i < importantInfo.length; i++) {  
    try {  
        Thread.sleep(4000); // Pause for 4 seconds.  
    } catch (InterruptedException e) {  
        return; // We've been interrupted: no more messages.  
    }  
    System.out.println(importantInfo[i]);  
}
```

- ◆ Cosa fare per gestire le interruzioni se un thread esegue un metodo che non lancia `InterruptedException`?
  - Deve invocare periodicamente `Thread.interrupted`, che rivela la ricezione di una interruzione
  - Es. il codice controlla `interrupted` ed esce da run quando riceve una interruzione

```
for (int i = 0; i < inputs.length; i++) {  
    heavyCrunch(inputs[i]);  
    if (Thread.interrupted()) {  
        //We've been interrupted: no more crunching.  
        return;  
    }  
}
```

- ◆ In applicazioni più complesse, può avere senso lanciare una `InterruptedException`
- ◆ Così il codice di gestione delle interruzioni viene centralizzato in una clausola `catch`

```
if (Thread.interrupted()) {  
    throw new InterruptedException();  
}
```

- ◆ Il meccanismo delle interruzioni è implementato usando un flag interno
- ◆ L'invocazione del metodo di classe `Thread.interrupt` imposta questo flag
  - Quando un thread controlla se c'è una interruzione invocando il metodo statico `Thread.interrupted`, allora lo stato delle interruzioni viene cancellato
- ◆ Il metodo di istanza `Thread.isInterrupted` non cambia lo stato delle interruzioni
  - Usato da un thread per controllare lo stato di un altro thread
- ◆ Per convenzione, ogni metodo che esce lanciando una `InterruptedException` cancella in quel momento lo stato delle interruzioni

- ◆ Il metodo `join` permette ad un thread di aspettare il completamento di un altro
- ◆ Se `t` è un oggetto `Thread` in esecuzione, `t.join()` provoca la pausa del thread corrente fino alla terminazione di `t`
- ◆ Esiste una versione `overloaded` che permette di specificare un periodo d'attesa
  - Tuttavia, come `sleep`, anche `join` dipende dal S.O. per il tempo, quindi non si può assumere che l'attesa sarà lunga quanto specificato
- ◆ Come `sleep`, `join` risponde ad una interruzione generando una `InterruptedException`



- ◆ Esempio che mette assieme alcuni dei concetti visti fin qui; ci sono due thread
  - Il thread principale, crea un nuovo thread da un oggetto Runnable chiamato MessageLoop,
  - Il thread principale aspetta che il nuovo thread finisca
  - Se il thread MessageLoop ci mette troppo a finire, il thread principale lo interrompe
  - Il thread MessageLoop scrive una serie di messaggi
  - Se interrotto prima di aver scritto tutti i messaggi, il thread MessageLoop scrive un altro messaggio ed esce

```
public class SimpleThreads {

    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + ": " + message);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {"Mares eat oats", "Does eat oats",
                "Little lambs eat ivy", "A kid will eat ivy too"};
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    Thread.sleep(4000); //Pause for 4 seconds
                    threadMessage(importantInfo[i]); //Print a message
                }
            } catch (InterruptedException e) { threadMessage("I wasn't done!"); }
        }
    }

    // ...
}
```

```
// ...
public static void main(String args[]) throws InterruptedException {
    // Delay before we interrupt MessageLoop thread (default 1 hour).
    long patience = 1000 * 60 * 60;
    // If command line argument present, gives patience in seconds.
    if (args.length > 0) { patience = Long.parseLong(args[0]) * 1000; }
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop()); t.start();
    threadMessage("Waiting for MessageLoop thread to finish");
    while (t.isAlive()) { // loop until t exits.
        threadMessage("Still waiting...");
        t.join(1000); // Wait (max 1 sec) for t to finish.
        if (((System.currentTimeMillis() - startTime) > patience) &&
            t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            t.join(); //Shouldn't be long now -- wait indefinitely
        }
    }
    threadMessage("Finally!");
} // main
} // SimpleThreads
```

- ◆ I thread comunicano principalmente attraverso la condivisione di oggetti in memoria
- ◆ Comunicazione efficiente, ma possibili errori

### 1. Interferenza tra thread

- Errori introdotti quando thread multipli accedono a dati condivisi

### 2. Errori di **inconsistenza** di memoria

- Viste inconsistenti della memoria condivisa

### ◆ **Sincronizzazione**: strumento per prevenire questi errori

- **Metodi sincronizzati** – Semplice paradigma che può efficacemente prevenire le due forme d' errore
- **Lock intrinseci** e sincronizzazione – Paradigma più generale, come si può ottenere sincronizzazione sulla base di lock impliciti
- **Accesso atomico** – Si basa sull'idea che ci siano operazioni su cui altri thread non possono influire

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```

- ♦ L'esempio `Counter` è scritto in modo tale che:
  - Ogni invocazione di `increment` aggiunge 1 a `c`
  - Ogni invocazione di `decrement` sottrae 1 da `c`
- ♦ Però, se un oggetto `Counter` è referenziato da più thread, l'interferenza tra thread può impedire che questo si verifichi come previsto
- ♦ L'interferenza capita quando due operazioni, in esecuzione in thread diversi ma agenti sugli stessi dati, si intrecciano
- ♦ Questo significa che le due operazioni consistono di diversi passi, e le sequenze di passi si sovrappongono

- ◆ Potrebbe non sembrare possibile che operazioni su istanze di `Counter` si intreccino, dato che entrambe le operazioni su `c` sono istruzioni singole e semplici
- ◆ Però anche semplici istruzioni possono essere tradotte in più passi dalla macchina virtuale
- ◆ Ad esempio, la singola espressione `c++` può essere decomposta in tre passi:
  1. Recupera il valore attuale di `c`
  2. Incrementa il valore letto di 1
  3. Memorizza il valore incrementato di nuovo in `c`
- ◆ L'espressione `c--` può essere decomposta in maniera simile, eccetto che il secondo passo è un decremento

- ◆ Es. Il thread A invoca `increment` quasi nello stesso istante in cui il thread B invoca `decrement`
  - Se il valore iniziale di `c` è 0, le loro azioni possono sovrapporsi seguendo questo ordine:
    1. Thread A: Legge `c`
    2. Thread B: Legge `c`
    3. Thread A: Incrementa il valore letto; il risultato è 1
    4. Thread B: Decrementa il valore letto; il risultato è -1
    5. Thread A: Memorizza il risultato in `c`; `c` vale 1
    6. Thread B: Memorizza il risultato in `c`; `c` vale -1
  - Il risultato del thread A è perso, sovrascritto da B
  - Questo particolare intreccio è solo una delle possibilità
  - In circostanze diverse potrebbe andar perso il risultato di B, oppure potrebbe non esserci alcun errore
  - Intrecci imprevedibili: bug difficili da rilevare e correggere



- ◆ Diversi thread hanno viste inconsistenti di quelli che dovrebbero essere gli stessi dati
- ◆ Chiave per evitare errori di consistenza della memoria è capire la relazione *succede-prima*
- ◆ Garanzia che le scritture in memoria di una specifica istruzione siano visibili per un'altra istruzione
  
- ◆ Semplice campo intero definito e inizializzato:
  - `int counter = 0;`
  - Il campo counter è condiviso tra due thread, A e B
  - A incrementa counter:
  - `counter++;`
  - Subito dopo, il thread B scrive il valore di counter:
  - `System.out.println(counter);`

- ◆ Se le due istruzioni fossero eseguite nello stesso thread, il valore letto sarebbe sicuramente “1”
- ◆ Ma se le due istruzioni fossero eseguite in thread separati, il valore letto potrebbe anche essere “0”
- ◆ Non c'è garanzia che le modifiche del thread A al contatore siano visibili per il thread B
- ◆ A meno che il programmatore non stabilisca una relazione *succede-prima* tra queste due istruzioni
- ◆ Tra i diversi meccanismi che creano relazioni *succede-prima*, uno dei più importanti è la sincronizzazione

- ◆ Due azioni che creano relazioni di *succede-prima*
- 1. Quando una istruzione invoca `Thread.start...`
  - ◆ Ogni istruzione che ha una relazione *succede-prima* con quella istruzione ha la stessa relazione con ogni istruzione eseguita dal nuovo thread
  - ◆ Gli effetti del codice che ha portato alla creazione del nuovo thread sono visibili al nuovo thread
- 2. Quando un thread termina e provoca il ritorno da un `Thread.join` in un altro thread...
  - ◆ Allora tutte le istruzioni eseguite dal thread terminato hanno una relazione *succede-prima* con tutte le istruzioni seguenti alla chiamata a `join`
  - Gli effetti del codice nel thread sono ora visibili al thread che ha eseguito il `join`

- ◆ Due paradigmi di base per la sincronizzazione
  - *Metodi sincronizzati*
  - *Istruzioni sincronizzate*
- ◆ Per rendere un metodo sincronizzato, si aggiunge la parola `synchronized` alla sua dichiarazione
- ◆ Oggetto visibile a più thread: tutte le letture e scritture dei campi quell'oggetto devono essere fatte attraverso metodi sincronizzati
  - Semplice strategia per prevenire l'interferenza tra thread e gli errori di consistenza di memoria
  - Strategia efficace, ma (vedremo che) può presentare problemi di *liveness*
  - Eccezione importante: i campi costanti (`final`) possono essere letti tranquillamente attraverso metodi non sincronizzati

- ♦ Se supponiamo di rendere sincronizzati i metodi di `Counter`, abbiamo due effetti
  1. Non è possibile che due invocazioni di metodi sincronizzati sullo stesso oggetto si intreccino
    - Un thread sta eseguendo un metodo sincronizzato su `obj`...
    - Allora tutti gli altri thread che invocano metodi sincronizzati su `obj` si bloccano...
    - Fin quando il primo thread non avrà finito le operazioni su `obj`
  2. L'esecuzione di un metodo sincronizzato stabilisce una relazione *succede-prima*
    - ♦ Rispetto a qualsiasi invocazione successiva di un metodo sincronizzato sullo stesso oggetto
    - ♦ Garanzia che il nuovo stato sia visibile a tutti i thread

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

- ♦ *I costruttori **non** possono essere sincronizzati*
  - Usare la parola riservata `synchronized` con un costruttore è un errore di sintassi
- ♦ Sincronizzare i costruttori non ha molto senso
  - Solo il thread che crea un oggetto dovrebbe avervi accesso
- ♦ Però... quando si costruisce un oggetto condiviso, bisogna prestare attenzione che non ci siano delle “perdite” del riferimento a `this`
  - Es. si vuole tenere una lista (`static`) delle istanze di una classe
  - `instances.add(this);`
  - Se si inserisce questa istruzione nel costruttore, altri thread potrebbero accedere, tramite `instances`, all’oggetto durante la sua fase di costruzione

- ◆ Ogni oggetto ha associato un lock intrinseco, o **monitor**
- ◆ Un thread che necessita un accesso esclusivo e consistente ai campi di un oggetto...
  - Deve **acquisire** il lock intrinseco dell'oggetto prima di accedervi...
  - E poi **rilasciare** il lock intrinseco quando ha finito di operarvi
  - Si dice che un thread **possiede** il lock intrinseco nell'intervallo di tempo tra acquisizione e rilascio del lock
- ◆ Fin tanto che un thread possiede un lock...
  - Nessun altro thread può acquisire lo stesso lock
  - L'altro thread si bloccherà quando tenterà di acquisirlo
- ◆ Quando un thread rilascia un lock intrinseco
  - Si stabilisce una relazione *succede-prima* tra quell'azione e ogni successiva acquisizione dello stesso lock



- ◆ Quando un thread invoca un *metodo sincronizzato...*
  - Acquisisce automaticamente il lock intrinseco dell'oggetto
  - E lo rilascia quando il metodo termina
  - Il rilascio del lock si verifica anche se la terminazione è provocata da una eccezione non catturata
  
- ◆ Che succede quando viene invocato un metodo di classe (`static`) sincronizzato?
  - Un metodo `static` è associato ad una classe
  - In questo caso, il thread acquisisce il lock intrinseco dell'oggetto `Class`, che rappresenta la classe (Reflection API)
  - L'accesso ai campi `static` di una classe è controllato da un lock distinto dal lock di qualsiasi sua istanza

- ◆ Codice sincronizzato, ma occorre specificare l'oggetto che fornisce il lock intrinseco
  - Differenza rispetto ai metodi sincronizzati
- ◆ Es. Il metodo `addName` necessita di sincronizzare i cambi a `lastName` e `nameCount`
  - Ma deve evitare di sincronizzare le invocazioni di metodi di altri oggetti
  - (Invocare metodi di altri oggetti da codice sincronizzato può creare problemi di *liveness*)
  - Senza istruzioni sincronizzate è necessario un metodo in più

```
public void addName(String name) {  
    ...  
    synchronized(this) {  
        lastName = name; nameCount++;  
    }  
    ...  
}
```

- ♦ La classe `TwoLocks` ha due campi mai usati assieme
  - Gli aggiornamenti dei singoli campi devono essere sincronizzati
  - Ma ok se gli aggiornamenti di `c1` si intrecciano con quelli di `c2`
  - Altrimenti, si riduce la concorrenza creando blocchi non necessari
  - Pattern: non metodi sincronizzati, ma due oggetti come lock

```
public class TwoLocks {  
    private long c1 = 0, c2 = 0;  
    private Object lock1 = new Object(), lock2 = new Object();  
    public void inc1() {  
        synchronized(lock1) { c1++; }  
    }  
    public void inc2() {  
        synchronized(lock2) { c2++; }  
    }  
}
```

- ◆ Un thread non può acquisire un lock posseduto da un altro thread
- ◆ Ma un thread *può* acquisire un lock che già possiede (*reentrant synchronization*)
- ◆ Quando codice sincronizzato invoca, direttamente o indirettamente, altro codice sincronizzato
- ◆ Ed entrambi i frammenti di codice usano lo stesso lock
- ◆ Senza sincronizzazione rientrante, un thread causerebbe facilmente il suo stesso blocco

- ♦ Azione atomica: viene eseguita tutta assieme
  - Una azione atomica non può fermarsi a metà: o avviene completamente, o non avviene affatto
  - Non ci sono effetti collaterali di una azione atomica visibili prima che essa sia completa
- ♦ Una espressione di incremento (es. `c++`) non è una azione atomica; idem per diverse semplici espressioni
- ♦ Alcune azioni che la VM assicura come atomiche:
  - Lettura e scrittura di riferimenti e dati primitivi (eccetto `long` e `double`)
  - Lettura e scrittura di variabili `volatile` (anche `long` o `double`)

- ♦ Le azioni atomiche non possono intrecciarsi
  - Nessun timore di interferenza
- ♦ Non elimina il bisogno di sincronizzare azioni atomiche
  - Ancora possibili errori di consistenza di memoria
- ♦ Accesso a variabili atomiche efficiente
  - Più che codice sincronizzato
  - Ma richiede più attenzione per assicurare consistenza
- ♦ `java.util.concurrent`: classi con metodi atomici

- ◆ `int get()`
- ◆ `void set(int newValue)`
- ◆ `int addAndGet(int delta)`
- ◆ `boolean compareAndSet(int expect, int update)`
- ◆ `int decrementAndGet()`
- ◆ `int getAndAdd(int delta)`
- ◆ `int getAndDecrement()`
- ◆ `int getAndIncrement()`
- ◆ `int getAndSet(int newValue)`
- ◆ `int incrementAndGet()`

- ♦ **Liveness**: capacità di una applicazione concorrente di eseguire in maniera tempestiva
  - Problema più comune: *deadlock*
  - Altri problemi: *starvation*, *livelock*
- ♦ **Deadlock**: situazione in cui due o più thread rimangono bloccati per sempre, l'uno in attesa dell'altro



- ◆ Es. Alfonso e Gastone sono due amici molto cortesi
- ◆ Rigida regola di cortesia: quando ti inchini ad un amico, devi rimanere inchinato finchè il tuo amico non ha una possibilità di restituire l'inchino
- ◆ Ma se i metodi `bow` vengono invocati assieme...
- ◆ Entrambi i thread si bloccheranno invocando `bowBack`
- ◆ Nessuno dei due blocchi avrà fine, perchè ogni thread aspetta l'altro

```
public class Deadlock {  
    public static void main(String[] args) {  
        final Friend alphonse = new Friend("Alphonse");  
        final Friend gaston = new Friend("Gaston");  
  
        new Thread(new Runnable() {  
            public void run() { alphonse.bow(gaston); }  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() { gaston.bow(alphonse); }  
        }).start();  
    }  
}
```

```
public class Friend {
    private final String name;

    public Friend(String name) {
        this.name = name;
    }

    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s has bowed to me!%n",
            this.name, bower.getName());
        bower.bowBack(this);
    }

    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s has bowed back to me!%n",
            this.name, bower.getName());
    }
}
```

- ◆ Starvation e livelock sono problemi molto meno comuni del deadlock, ma bisogna tenerne conto
- ◆ Starvation: un thread non ottiene accesso regolare alle risorse condivise e non riesce ad avanzare
- ◆ Risorse rese non disponibili da thread “avidì” (*greedy*)
- ◆ Es. Un oggetto fornisce un metodo sincronizzato che richiede molto tempo per essere eseguito
- ◆ Se un thread invoca il metodo frequentemente, gli altri thread saranno spesso bloccati

- ◆ Un thread può rispondere ad azioni di un altro thread
- ◆ Il livelock si può verificare come *ciclo* se l'azione di origine è anch'essa una reazione a qualche altro thread
- ◆ I thread in livelock sono incapaci di avanzare
- ◆ Ma i thread non sono bloccati, a differenza del deadlock
- ◆ Sono troppo occupati a risponderci l'un l'altro per poter continuare il lavoro
- ◆ Caso simile a due persone di fronte in un corridoio
  - Alfonso si sposta alla sua destra per far passare Gastone
  - Gastone si sposta alla sua sinistra per far passare Alfonso...

- ◆ I thread spesso devono coordinare le loro azioni
- ◆ Pattern più comune: blocchi con guardia
  - Controllo di una condizione: vera prima di proseguire
  - Ci sono diversi passi da seguire per fare questo correttamente
- ◆ Es. una variabile condivisa `joy` dev'essere settata da un altro thread, prima di proseguire
  - Ciclo (con **polling**) finchè la condizione non è verificata
  - **Attesa attiva**: spreco CPU , esecuzione continua durante l'attesa

```
public void guardedJoy() {  
    // Simple loop guard. Wastes processor time. Don't do this!  
    while (!joy) {}  
    System.out.println("Joy has been achieved!");  
}
```

- ◆ `Object.wait` per sospendere il thread corrente
  - `wait` non ritorna finchè un altro thread non notifica qualche evento
  - Possibile che non sia l'evento che il thread attende
  - Invocare `wait` sempre in un ciclo che ricontrolla la condizione
  - Non assumere che la notifica sia per la particolare condizione su cui si aspetta, o che la condizione sia ancora vera

```
public synchronized guardedJoy() {  
    while (!joy) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and efficiency have been achieved!");  
}
```

1. Quando viene invocato `wait`, il thread ***rilascia il lock*** e sospende la sua esecuzione
2. In seguito, un altro thread acquisirà lo stesso lock e invocherà `notifyAll`, che informa tutti i thread sospesi su quel lock
3. Qualche tempo dopo che il secondo thread rilascia il lock, il primo thread riacquisisce il lock e riprende l'esecuzione, dato che il metodo `wait` ritorna

```
public synchronized notifyJoy() {  
    joy = true;  
    notifyAll();  
}
```



- ◆ Perché il metodo `guardedJoy` è sincronizzato?
  - Quando un thread invoca `obj.wait`, deve possedere il lock intrinseco di `obj`
  - Altrimenti si ha un errore a run-time
  - Invocare `wait` in un metodo sincronizzato è un modo semplice per acquisire il lock
- ◆ Come molti metodi che sospendono l'esecuzione, `wait` può lanciare `InterruptedException`
- ◆ Secondo metodo di notifica, `notify`: sveglia un solo thread
  - Ma non c'è modo di specificare quale thread svegliare
  - Utile in applicazioni massivamente parallele
  - Gran numero di thread simili, non importa quale viene svegliato

- ◆ I blocchi con guardia possono essere usati per creare applicazioni di tipo produttore-consumatore
- ◆ Due thread comunicano usando un oggetto condiviso
  - Il produttore deposita i dati nell'oggetto condiviso
  - Il consumatore li preleva per farci qualcosa
- ◆ La coordinazione è essenziale
  - Il thread consumatore non deve leggere i dati prima che siano depositati dal produttore
  - Il thread produttore non deve depositare i dati prima che il consumatore abbia prelevato quelli vecchi

- ◆ Esempio: i dati sono una serie di messaggi, condivisi attraverso un oggetto di tipo `Drop`
- ◆ Il thread produttore, di tipo `Producer`, invia una serie di messaggi
- ◆ La stringa “DONE” indica che tutti i messaggi sono stati inviati
- ◆ Il thread produttore aspetta per un intervallo di tempo casuale tra due messaggi
- ◆ Il thread consumatore, di tipo `Consumer`, preleva i messaggi e li scrive, finchè non legge la stringa “DONE”
- ◆ Anche questo thread aspetta per intervalli casuali

- ◆ Il thread principale avvia il thread produttore e quello consumatore
- ◆ Nota: la classe `Drop` è solo un esempio
- ◆ Per evitare di re-inventare la ruota meglio cercare prima nel Collections Framework

```
public class ProducerConsumerExample {  
    public static void main(String[] args) {  
        Drop drop = new Drop();  
        (new Thread(new Producer(drop))).start();  
        (new Thread(new Consumer(drop))).start();  
    }  
}
```

```
public class Drop {
    private String message;
    private boolean empty = true; // Should consumer wait?
    public synchronized String take() {
        while (empty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        empty = true;
        notifyAll(); // Notify producer that status has changed.
        return message;
    }
    public synchronized void put(String message) {
        while (!empty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        empty = false; this.message = message;
        notifyAll(); // Notify consumer that status has changed.
    }
}
```

```
public class Producer implements Runnable {
    private Drop drop;
    public Producer(Drop drop) {    this.drop = drop;    }

    public void run() {
        String importantInfo[] = { "Mares eat oats", "Does eat oats",
            "Little lambs eat ivy", "A kid will eat ivy too" };
        java.util.Random random = new java.util.Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

```
public class Consumer implements Runnable {
    private Drop drop;
    public Consumer(Drop drop) { this.drop = drop; }

    public void run() {
        java.util.Random random = new java.util.Random();
        for (String message = drop.take(); !message.equals("DONE");
            message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

*Non possono cambiare stato dopo la costruzione*

- ◆ In applicazioni concorrenti, gli oggetti immutabili non possono essere corrotti da interferenze tra thread o osservati in stato inconsistente
- ◆ Programmatori riluttanti ad usare oggetti immutabili
  - Preoccupazione per i costi di creazione di un nuovo oggetto anzichè aggiornare un oggetto già esistente
  - Impatto della creazione di oggetti spesso sovrastimata
  - Minor overhead per garbage collection e nessun codice per proteggere dalla corruzione
- ◆ Es. la classe `SynchRGB` rappresenta colori
  - Ogni oggetto rappresenta il colore come tre interi e una stringa che contiene il nome del colore



```
public class SynchRGB {
    private int red; //Values must be between 0 and 255.
    private int green;
    private int blue;
    private String name;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255 || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public SynchRGB(int red, int green, int blue, String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }
    // ...
}
```

```
// ...
public void set(int red, int green, int blue, String name) {
    check(red, green, blue);
    synchronized (this) {
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }
}
public synchronized int getRGB() {
    return ((red << 16) | (green << 8) | blue);
}
public synchronized String getName() { return name; }
public synchronized void invert() {
    red = 255 - red;  green = 255 - green;  blue = 255 - blue;
    name = "Inverse of " + name;
}
}
```

```
SynchRGB color = new SynchRGB(0, 0, 0, "Pitch Black");  
// ...  
int myColorInt = color.getRGB();  
String myColorName = color.getName();
```

- ◆ Accedere con cautela, per evitare di osservare uno stato inconsistente
- ◆ Se un altro thread invoca `set` dopo `getRGB` ma prima di `getName`, il valore RGB non corrisponderà al nome
- ◆ Per evitare ciò, le due istruzioni devono essere legate assieme

```
synchronized (color) {  
    int myColorInt = color.getRGB();  
    String myColorName = color.getName();  
}
```

- ◆ Inconsistenze possibili solo per oggetti mutabili
- ◆ Non sono un problema per la versione immutabile

- ◆ Strategia per definire oggetti immutabili
  1. Non fornire metodi “setter”, che modificano i campi o oggetti riferiti
  2. Marcare tutti i campi con `final` e `private`
  3. Non permettere alle sottoclassi di sovrascrivere metodi
    - ◆ Classe `final`
    - ◆ Oppure costruttore privato e pattern *factory*
  4. Se i campi di istanza includono riferimenti a oggetti mutabili, non permettere che vengano modificati
    - ◆ Non fornire metodi che modificano tali oggetti
    - ◆ Non fornire riferimenti agli oggetti mutabili
    - ◆ Non memorizzare oggetti esterni e mutabili passati al costruttore (se necessario, creare copie)
    - ◆ Creare copie degli oggetti interni mutabili quando necessario fornirli all'esterno tramite metodi `getter`

- ◆ Ci sono due metodi setter nella classe
  - Il primo, `set`, è da cancellare!
  - Il secondo, `invert`, si può sistemare facendogli restituire un nuovo oggetto, invece che modificare quello esistente
- ◆ Tutti i campi sono già `private`, ma li definiamo anche come `final`
- ◆ La classe stessa è definita `final`
- ◆ Solo un campo fa riferimento ad un oggetto (`String`), ma quell'oggetto è immutabile: ok!
- ◆ Dopo le modifiche, abbiamo `ImmutableRGB`

```
public final class ImmutableRGB {  
    //Values must be between 0 and 255.  
    final private int red;  
    final private int green;  
    final private int blue;  
    final private String name;  
  
    public ImmutableRGB(int red, int green, int blue, String name) {  
        check(red, green, blue);  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
        this.name = name;  
    }  
    // ...  
}
```

```
// ...
private void check(int red, int green, int blue) {
    if (red < 0 || red > 255
        || green < 0 || green > 255
        || blue < 0 || blue > 255) {
        throw new IllegalArgumentException();
    }
}

public int getRGB() { return ((red<<16) | (green<<8) | blue); }
public String getName() { return name; }

public ImmutableRGB invert() {
    return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
        "Inverse of " + name);
}
}
```

- ◆ Gli oggetti `Lock` funzionano in maniera molto simile ai lock impliciti usati per sincronizzare il codice
  - Come per i lock impliciti, solo un thread alla volta può possedere un oggetto `Lock`
  - Gli oggetti `Lock` supportano anche un meccanismo di attesa/notifica attraverso degli oggetti `Condition` a loro associati
- ◆ Il principale vantaggio degli oggetti `Lock` è la possibilità di rinunciare ad un tentativo di acquisizione di un lock
  - Il metodo `tryLock` ritorna subito se il lock non è disponibile, oppure se non si libera entro un certo tempo
  - Il metodo `lockInterruptibly` ritorna se un altro thread invia una interruzione prima che il thread sia acquisito

*Permettono di evitare/risolvere situazioni di deadlock*



- ◆ Fornisce un solo metodo, `execute`, per sostituire la creazione di thread di basso livello

```
// (new Thread(r)).start();  
e.execute(r); // r is a Runnable
```

- ◆ Definizione meno specifica, comportamento dipendente dall'implementazione
  - Semplice creazione nuovo thread e schedulazione
  - Uso di un worker thread esistente, possibilmente in un pool
  - Inserimento del compito in una coda

- ◆ `newFixedThreadPool`
  - Crea un esecutore basato su un numero fisso di thread, cui vengono sottoposti i compiti
  - Se ci sono più compiti che thread, una coda interna conserva i compiti in eccesso fino alla liberazione di un thread
  
- ◆ `newCachedThreadPool`
  - Crea un esecutore con un thread pool espandibile, adatto ad applicazioni che possono avere molti compiti da eseguire rapidamente
  
- ◆ `newSingleThreadExecutor`
  - Crea un esecutore basato su un singolo thread, che esegue un compito alla volta

- ◆ Coda con ulteriori metodi per aspettare che...
- ◆ La coda diventi non-vuota quando si prende un elemento
- ◆ Ci sia spazio quando si aggiunge un elemento
  
- ◆ `ArrayBlockingQueue`
- ◆ `LinkedBlockingQueue`
- ◆ ...

```
class Producer implements Runnable {  
    private final BlockingQueue<String> queue;  
    Producer(BlockingQueue<String> q) { queue = q; }  
  
    public void run() {  
        try {  
            while (true) { queue.put(produce()); }  
        } catch (InterruptedException ex) { ... handle ...}  
    }  
    String produce() { ... }  
}
```

```
class Consumer implements Runnable {  
    private final BlockingQueue<String> queue;  
    Consumer(BlockingQueue<String> q) { queue = q; }  
  
    public void run() {  
        try {  
            while (true) { consume(queue.take()); }  
        } catch (InterruptedException ex) { ... handle ...}  
    }  
    void consume(String x) { ... }  
}
```

```
class Setup {  
    void main() {  
        BlockingQueue<String> q =  
            new LinkedBlockingQueue<String> ();  
        Producer p = new Producer(q);  
        Consumer c1 = new Consumer(q);  
        Consumer c2 = new Consumer(q);  
        new Thread(p).start();  
        new Thread(c1).start();  
        new Thread(c2).start();  
    }  
}
```