Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 8: Thread

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Multitasking e multithreading · Thread e processi · Runnable e Thread
start e run · Interleaving e sleep · Variabili nei thread
Ciclo di vita · Priorità · join e interrupt · synchronized e monitor
Deadlock · wait e notify · Produttore e consumatore

Indice

1 Multitasking e multithreading 1

2 Thread e processi 1

3 Runnable e Thread 2

4 Interleaving e `sleep` 4

5 Variabili nei thread 5

6 Il ciclo di vita di un thread 7

7 Schedulazione e priorità 8

8 Aspettare un thread: `join` 9

9 Interrompere un thread: `interrupt` 11

10 Risorse condivise e `synchronized` 13

11 Il monitor 15

12 Deadlock e thread demoni 16

13 Rientranza 19

14 `wait`, `notify` e `notifyAll` 19

15 Produttore e consumatore 20

16 Esercizi 25

17 Soluzioni 27

1 Multitasking e multithreading

Definizione – Tre modi di eseguire

- **Monotasking**: si esegue un'applicazione alla volta, in rigorosa sequenza.
- **Multitasking**: più applicazioni diverse avanzano «contemporaneamente», ciascuna nella propria area di memoria.
- **Multithreading**: dentro una stessa applicazione avanzano «contemporaneamente» più esecuzioni, i **thread**, che condividono l'area di memoria dell'applicazione.

Un processore esegue un'istruzione alla volta, ma in tempi brevissimi. Il sistema operativo e la macchina virtuale danno il processore a turno ai flussi da eseguire: mentre uno aspetta, per esempio un'azione dell'utente o il disco, il processore lavora per un altro. I turni sono così brevi che i flussi sembrano procedere insieme. Questa alternanza si chiama **interleaving**.

Con più core, flussi diversi girano davvero insieme su core diversi; quando i flussi sono più dei core, si alternano comunque. Quando e come avvenga il passaggio da un flusso all'altro lo decidono il sistema operativo e la macchina virtuale, non il programma.

Definizione – Thread

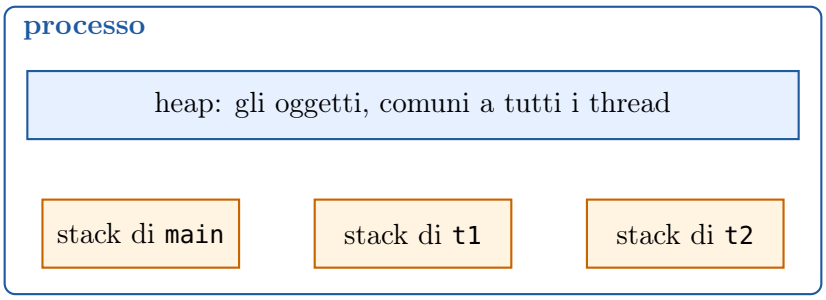
Un **thread** è un flusso di esecuzione sequenziale e indipendente dentro un programma. È detto anche *lightweight process*, processo leggero.

2 Thread e processi

Proposizione – Che cosa è di un thread e che cosa è comune

- Ogni thread ha solo ciò che serve a un flusso indipendente: il suo contesto (fra cui il punto dell'esecuzione) e il suo **stack**.
- Lo spazio di indirizzamento, compreso lo **heap**, e le altre risorse sono comuni a tutti i thread del processo.

Per questo creare un thread è più semplice e più rapido che creare un processo.



I thread di un processo hanno ciascuno il suo stack, con le variabili dei metodi in esecuzione, e condividono lo heap.

Anche un programma Swing ha più thread. Il `main` costruisce la finestra e termina, ma la finestra resta attiva: gli eventi e il disegno li gestisce un altro thread, che continua dopo la fine del `main` (Capitolo 7).

Nota – A che cosa servono i thread

- A simulare attività simultanee.
- A sfruttare più processori per lo stesso programma.
- A usare bene le risorse lente, come il disco e l'utente.
- A tenere reattiva l'interfaccia: le operazioni lunghe vanno a thread dedicati, e il thread che gestisce l'utente resta libero.

3 Runnable e Thread

Definizione – Eseguibile ed esecutore

- L'**eseguibile** è un oggetto di una classe che implementa l'interfaccia `Runnable`: il lavoro da fare sta nel suo metodo `public void run()`.
- L'**esecutore** è un oggetto `Thread`, che riceve l'eseguibile nel costruttore e lo esegue in un thread nuovo.

`Thread` e `Runnable` stanno in `java.lang`, che non si importa. Un oggetto `Thread` non è il thread: è lo strumento con cui si chiede alla macchina virtuale, l'unica che crea davvero i thread, di avviarne uno, di interromperlo o di aspettarne la fine. Creare l'oggetto non crea nessun thread.

```java
public class MioRunnable implements Runnable {
    private String nome;

    public MioRunnable(String n) {
        nome = n;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(nome + ": " + i);
        }
        System.out.println(nome + ": DONE! ");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        MioRunnable ja = new MioRunnable("Jamaica");
        Thread mioThread = new Thread(ja);
        mioThread.start();
        System.out.println("main: fine");
    }
}
```

Proposizione – Lo schema in tre istruzioni

1. si crea l'eseguibile, un oggetto `Runnable`;
2. si crea l'esecutore, `new Thread(eseguibile)`;
3. si chiama `start()` sull'esecutore.

Prima di `start()` c'è un solo flusso di esecuzione; da lì in poi ce ne sono due.



L'esecutore tiene il riferimento all'eseguibile: `start()` esegue `ja.run()` in un thread nuovo.

Proposizione – `start` e `run`

- `mioThread.start()` crea il thread e vi esegue `run()` dell'eseguibile; il chiamante prosegue subito.
- `ja.run()` chiamato direttamente è una chiamata come le altre: `run()` gira nel thread che la fa, e nessun thread nuovo parte.

Il programma stampa di solito prima `main: fine` e poi `Jamaica: 0`, ..., `Jamaica: 9`, `Jamaica: DONE!`: dopo `start()` il `main` ha ancora il processore e scrive subito. Ma l'ordine non è garantito: lo decide chi assegna i turni. Con `Thread.sleep(100)` prima della stampa del `main` (Sezione 4), il `main` cede il processore e la sua riga esce per ultima.

Osservazione – Estendere `Thread`

Si può anche scrivere una classe che estende `Thread` e ridefinisce `run()`, e chiamare `start()` su un suo oggetto. Il risultato è lo stesso, ma la classe non può più estendere un'altra classe, perché Java ha l'ereditarietà singola, e mescola il lavoro da fare con il meccanismo che lo esegue. Si usa sempre il primo modo: `Runnable` più `Thread`.

4 Interleaving e `sleep`

Due thread stampano cento volte un carattere ciascuno, uno `0` e l'altro `1` (Esercizio 1). Spesso escono prima tutti gli `0` e poi tutti gli `1`: i thread girano in modo concorrente, ma il primo è così veloce da finire prima che il secondo ottenga il turno.

Definizione – `Thread.sleep`

```
public static void sleep(long millisec)
```

sospende per `millisec` millisecondi il thread che esegue l'istruzione: il thread diventa non eseguibile e, scaduto il tempo, torna eseguibile. Può lanciare `InterruptedException`, che va gestita con `try` e `catch` oppure rilanciata.

Scaduto il tempo il thread è di nuovo pronto, ma non riparte per forza subito: tocca a chi assegna i turni. `sleep` non serve a rispettare scadenze precise. Il tempo è in millisecondi: due secondi sono `2000`.

```java
public class PrintExecutable implements Runnable {
    private final char c;

    public PrintExecutable(char c) {
        this.c = c;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.print(c);
            try {
                Thread.sleep((long) (Math.random() * 10));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Dopo ogni carattere il thread dorme qualche millisecondo, e in quel tempo il processore passa all'altro. Ora `0` e `1` si alternano, in modo diverso a ogni esecuzione:

```
0101011010001010101001110101001011011001011001100101101011...
```

Un thread che chiama `sleep` cede il processore di sua iniziativa: l'alternanza è più regolare, ma il numero di `0` e `1` consecutivi resta imprevedibile.

5 Variabili nei thread

Proposizione – Chi vede quale variabile

- Le variabili di un metodo, parametri compresi, sono locali al thread che esegue il metodo: stanno nel suo stack, e ogni thread ha le sue.
- Un campo di istanza è comune a tutti i thread che usano quell'oggetto.
- Un campo `static` è comune a tutti i thread del programma.

Più thread che accedono alla stessa variabile hanno bisogno di regole di **sincronizzazione**.

`Thread.currentThread()` restituisce l'oggetto `Thread` che sta eseguendo l'istruzione, e `getName()` il suo nome, che si assegna con `setName`:

```java
public class RunExecutable implements Runnable {
    public void run() {
        for (int i = 0; i < 100; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running " + i);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()
                + " is running");
        Thread alpha = new Thread(new RunExecutable());
        Thread beta = new Thread(new RunExecutable());
        alpha.setName("Alpha thread");
        beta.setName("Beta thread");
        alpha.start();
        beta.start();
        System.out.println(Thread.currentThread().getName()
                + " stops running");
    }
}
```

Il programma stampa subito `main is running` e `main stops running`: il thread del `main` si chiama `main`, avvia gli altri due e termina. Poi i due conteggi si intercalano, ma ciascuno va in ordine da 0 a 99: `i` è una variabile del metodo, e ogni thread ha la sua.

Un contatore condiviso

`i` diventa un campo di `RunExecutable`, e il ciclo diventa `for (i = 0; i < 100; i++)`. Gli esiti dipendono da com'è dichiarato il campo:

- `private int i;`: i due thread hanno due oggetti `RunExecutable` diversi, quindi due campi diversi, e il programma si comporta come prima;
- `private static int i;`: il campo è uno solo, e i due thread lo incrementano entrambi.

Con il campo `static` l'uscita perde l'ordine. In un'esecuzione, per esempio, Beta stampa 0, 1 e 2, poi Alpha stampa 0 e subito dopo 4. I due thread insieme fanno circa 100 giri invece di 200, e a ogni esecuzione la divisione cambia (82 e 19, 37 e 64, . . .).

Il motivo è l'interleaving. Un thread aumenta `i` e, prima di stamparlo, perde il turno; l'altro lo aumenta ancora. Il primo stampa poi un valore che non è più quello che aveva calcolato, e alcuni valori non li stampa nessuno. Ogni esecuzione dà un risultato diverso.

Osservazione – Serve la sincronizzazione

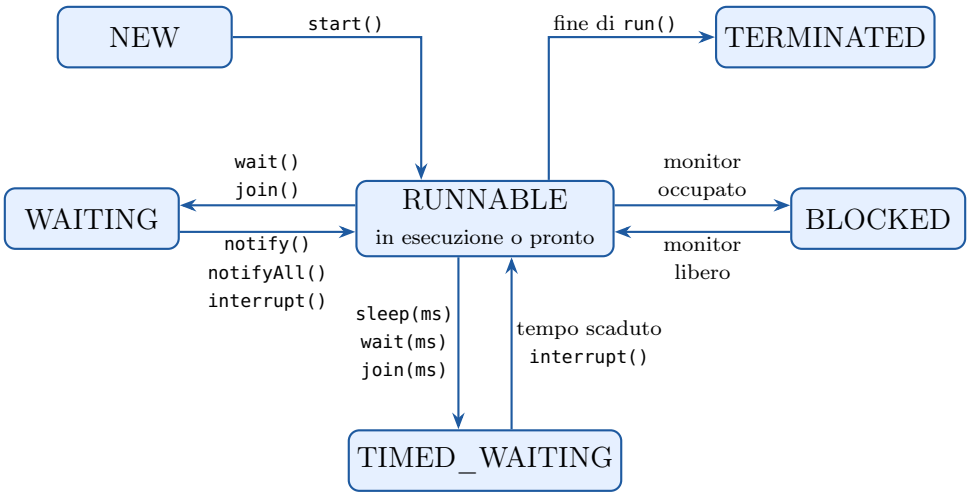
Un campo modificato da più thread senza coordinamento fa perdere il controllo del programma: non è più programmazione sequenziale. Servono meccanismi che lascino accedere un thread alla volta, a costo di farli aspettare; in Java sono i metodi `synchronized` (Sezione 10).

6 Il ciclo di vita di un thread

Un thread passa per pochi stati, e solo certe operazioni lo spostano dall'uno all'altro. Il diagramma degli stati qui sotto li riassume: i rettangoli sono gli stati, le frecce i passaggi, con l'operazione o l'evento che li provoca.

Definizione – Stati di un thread

- **NEW**: l'oggetto `Thread` è stato creato, ma `start()` non è ancora stata chiamata.
- **RUNNABLE**: il thread è avviato e può eseguire `run()`; a volte ha il processore (*running*), a volte aspetta il suo turno.
- **WAITING** e **TIMED_WAITING**: il thread è sospeso finché non accade un evento; nel secondo caso anche finché non scade un tempo.
- **BLOCKED**: il thread aspetta di entrare in un metodo `synchronized` occupato (Sezione 11).
- **TERMINATED**: `run()` è finito. È uno stato finale: il thread non riparte più.



Gli stati di un thread. Da WAITING, TIMED_WAITING e BLOCKED il thread torna sempre in RUNNABLE; solo la fine di `run()` lo porta in TERMINATED.

Un thread lascia il processore anche quando aspetta un'operazione di input e output, per esempio un tasto, il disco o un messaggio dalla rete. Ogni attesa ha un evento preciso che la chiude: scade il tempo, un altro thread lo avvisa, l'operazione si completa.

Definizione – `isAlive` e `yield`

- `public boolean isAlive()` restituisce `true` se il thread è stato avviato e non è ancora terminato, `false` se non è mai stato avviato o è terminato. È un'informazione grossolana: non distingue i due casi di `false`, né un thread in esecuzione da uno in attesa.
- `public static void yield()` suggerisce di cedere il processore ad altri thread pronti. È solo un suggerimento: il thread può riprendere subito.

7 Schedulazione e priorità

Chi assegna il processore ai thread, cioè lo **scheduler**, può seguire due politiche:

- **non preemptive**: un thread in esecuzione non viene mai interrotto, e cede il processore solo di sua iniziativa (input e output, `sleep`, `yield`);
- **preemptive**: lo scheduler può togliere il processore a un thread, per esempio quando ne è pronto uno di priorità più alta, o allo scadere di un intervallo di tempo (**time slicing**).

In Java lo scheduler è preemptive, e privilegia i thread di priorità più alta. Se e come si alternino i thread con la stessa priorità dipende dalla piattaforma: la specifica della macchina virtuale non dice come i thread Java corrispondano a quelli del sistema operativo. Per questo Java non garantisce tempi di risposta: non si usa per i programmi in tempo reale.

Definizione – Priorità

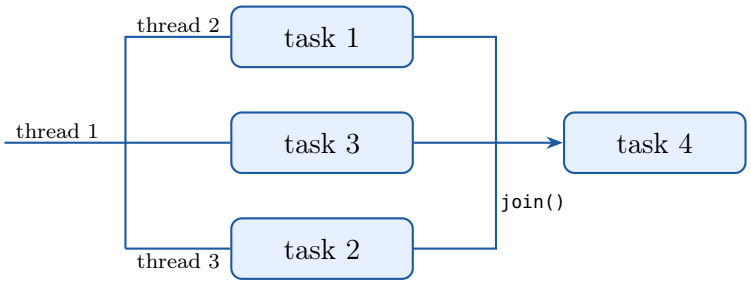
Ogni thread ha una priorità fra `Thread.MIN_PRIORITY` (1) e `Thread.MAX_PRIORITY` (10); il valore normale è `Thread.NORM_PRIORITY` (5). Si legge con `getPriority()` e si cambia con `setPriority(int)`. Indica solo un ordine di preferenza, non un tempo, e alcuni sistemi la ignorano.

Nota – Regole per programmi portabili

- Un ciclo lungo, o infinito, contiene operazioni che cedono il processore: input e output, `sleep`, attese di coordinamento. Un thread che fa solo calcoli (*CPU bound*) potrebbe altrimenti non cederlo mai.
- Non si fa affidamento sul time slicing né sulle priorità per ottenere un ordine di esecuzione.

8 Aspettare un thread: `join`

Dopo `start()` il programma ha due flussi. A volte un flusso deve aspettare che un altro finisca prima di continuare: le due linee di esecuzione si ricongiungono.



Il thread 1 avvia i thread 2 e 3, svolge il task 3 e aspetta la fine degli altri due prima del task 4.

Definizione – `join`

Se il thread corrente esegue `t.join()`, si sospende finché il thread `t` non termina. Le varianti `join(long millis)` e `join(long millis, int nanos)` aspettano al massimo il tempo indicato. Tutte possono lanciare `InterruptedException`.

Il primo esempio è un conto alla rovescia. Ogni oggetto `CountDown` stampa il suo nome e i numeri da 10 a 1, poi `Go!`, con una pausa di 100 millisecondi fra una riga e l'altra:

```
public class CountDown implements Runnable {
    private String id;
    private int countDown;

    public CountDown(String nome, int countDownPartenza) {
        id = nome;
        countDown = countDownPartenza;
    }

    private String status() {
        return id + ": " + (countDown > 0 ? countDown : "Go!") + "\n";
    }

    public void run() {
        while (countDown >= 0) {
            System.out.print(status());
            countDown--;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

In `status` l'espressione `countDown > 0 ? countDown : "Go!"` vale `countDown` se la condizione è vera, altrimenti `"Go!"`: è un `if-else` dentro un'espressione. Il ciclo rispetta le regole della Sezione 7: ogni giro stampa e dorme.

Il primo `main` chiama direttamente `h.run()` e poi `c.run()`: come nella Sezione 3, non parte nessun thread. Esce tutto il conto di `Huston`, poi tutto quello di `CapeCanaveral`, poi `Si parte!!!`.

Il secondo `main` avvia due thread, ma non li aspetta:

```
Thread t1 = new Thread(h);
Thread t2 = new Thread(c);
t1.start();
t2.start();
System.out.println("Si parte!!!");
```

I due conti si alternano, ma `Si parte!!!` esce subito, prima della fine dei conti: il `main` non ha aspettato nessuno.

```java
public class MainConcorrenteJoin {
    public static void main(String[] args) {
        CountDown h = new CountDown("Huston", 10);
        CountDown c = new CountDown("CapeCanaveral", 10);
        Thread t1 = new Thread(h);
        Thread t2 = new Thread(c);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            // il main è stato interrotto mentre aspettava
        }
        System.out.println("Si parte!!!");
    }
}
```

Ora il `main` aspetta `t1` e poi `t2`. I due conti si alternano come prima, ma `Si parte!!!` esce per ultima, dopo i due `Go!`. Il `join` sta in un blocco `try` perché può lanciare `InterruptedException`: il blocco `catch` dice che cosa fare se il `main` viene interrotto mentre aspetta (Sezione 9).

9 Interrompere un thread: `interrupt`

Definizione – `interrupt`

`t.interrupt()` chiede al thread `t` di interrompersi: imposta in `t` un indicatore, il *flag di interruzione*. È una richiesta, non un ordine.

- Se `t` è in esecuzione, prosegue. Si ferma solo se il suo codice controlla l'indicatore, con il metodo statico `Thread.interrupted()`, che lo legge e lo azzera.
- Se `t` è sospeso in `sleep`, `wait` o `join`, o lo diventa dopo, si risveglia: il metodo lancia `InterruptedException` e l'indicatore torna `false`. Il thread prosegue nel `catch`.

```java
public class SleepInterrupt implements Runnable {
    public void run() {
        try {
            System.out.println("in run()");
            Thread.sleep(20000);
            System.out.println("in run() - woke up");
        } catch (InterruptedException x) {
            System.out.println("in run() - interrupted while sleeping");
            return;
        }
        System.out.println("fine");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException x) {
        }
        System.out.println("in main() - interrupting other thread");
        t.interrupt();
        System.out.println("in main() - leaving");
    }
}
```

Il thread `t` stampa `in run()` e si addormenta per 20 secondi. Il `main` dorme 2 secondi e poi lo interrompe. L'uscita è:

```
in run()
in main() - interrupting other thread
in main() - leaving
in run() - interrupted while sleeping
```

Il programma finisce dopo circa 2 secondi, non 20: `sleep` lancia `InterruptedException`, e `t` esce dal `catch` con `return`, senza stampare `fine`. La riga del `main` dopo `interrupt()` esce prima: `interrupt()` non aspetta che `t` reagisca.

Osservazione – Interrupt non sincronizza

`interrupt` serve a gestire situazioni anomale, per esempio annullare un'attesa, non a coordinare l'ordine dei thread: non dice quando l'altro thread reagirà. Per aspettare un thread si usa `join`.

10 Risorse condivise e `synchronized`

Spesso più thread usano la stessa **risorsa condivisa**, per esempio lo stesso oggetto. Alcuni la scrivono, altri la leggono, e le operazioni si possono intercalare. I thread devono allora coordinarsi: devono **sincronizzarsi**.

```
public class Contatore {
    private int valore;

    public void incrementa() {
        int tmp = valore;
        try {
            Thread.sleep(10);
        } catch (InterruptedException ex) {
        }
        valore = tmp + 1;
        System.out.println(Thread.currentThread().getName()
                + ": il contatore segna " + valore);
    }

    public int getValore() {
        return valore;
    }
}
```

`incrementa` legge il valore, aspetta un poco e scrive il valore più uno: la `sleep` simula un'operazione che dura, e lascia il turno all'altro thread. Due thread `ProcessoConta` incrementano dieci volte lo stesso contatore:

```
public class ProcessoConta implements Runnable {
    private Contatore contatore;

    public ProcessoConta(Contatore c) {
        contatore = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            contatore.incrementa();
            try {
                Thread.sleep(10);
            } catch (InterruptedException ex) {
            }
        }
    }
}
```

```
Contatore cnt = new Contatore();
Thread a = new Thread(new ProcessoConta(cnt));
Thread b = new Thread(new ProcessoConta(cnt));
a.start();
b.start();
```

Ci si aspetterebbe che il contatore arrivi a 20. Invece ogni valore esce due volte:

```
Thread-1: il contatore segna 1
Thread-0: il contatore segna 1
Thread-0: il contatore segna 2
Thread-1: il contatore segna 2
...
```

I due thread leggono lo stesso valore in `tmp`, dormono entrambi e scrivono entrambi `tmp + 1`: un incremento va perso. Questa situazione, in cui il risultato dipende da come si alternano i thread, si chiama **race condition**.

Definizione – Metodo `synchronized`

Un metodo dichiarato `synchronized` viene eseguito da un solo thread alla volta sullo stesso oggetto. Se un thread chiama un metodo `synchronized` di un oggetto mentre un altro ne sta eseguendo uno, aspetta che l'altro finisca.

Basta aggiungere `synchronized` ai due metodi di `Contatore`:

```
public synchronized void incrementa() { ... }
public synchronized int getValore() { ... }
```

Ora lettura e scrittura di `incrementa` non si intercalano più, e il contatore arriva a 20. Spesso i thread si alternano, uno con i valori dispari e l'altro con i pari, ma l'alternanza non è garantita: è garantito solo che nessun incremento vada perso.

11 Il monitor

Definizione – Monitor

Ogni oggetto Java ha un **monitor**, un lucchetto che un solo thread alla volta può possedere. Un thread che entra in un metodo `synchronized` acquisisce il monitor dell'oggetto e lo rilascia all'uscita. Gli altri thread che chiedono lo stesso monitor aspettano in coda, nello stato BLOCKED. Ne risulta la **mutua esclusione**: le parti `synchronized` di uno stesso oggetto non sono mai eseguite da due thread insieme.

Proposizione – Tre forme di `synchronized`

- Un metodo di istanza `synchronized` usa il monitor dell'oggetto `this`.
- Un metodo `static synchronized` usa il monitor della classe.
- Un blocco `synchronized (obj) { ... }` usa il monitor dell'oggetto `obj`, solo per le istruzioni del blocco.

La sincronizzazione rallenta il programma, perché fa aspettare i thread. Conviene quindi sincronizzare solo le istruzioni che lo richiedono: per questo esiste il blocco.

Osservazione – Metodi non sincronizzati

Un metodo senza `synchronized` non chiede il monitor: un thread lo esegue anche se un altro thread sta eseguendo un metodo `synchronized` dello stesso oggetto. La mutua esclusione vale solo fra le parti sincronizzate.

12 Deadlock e thread demoni

Definizione – Deadlock

Si ha un **deadlock**, o stallo, quando due o più thread si aspettano a vicenda: ciascuno possiede un monitor che serve all'altro, e nessuno può proseguire.

```
public class Risorsa {
    private String nome;
    private int valore;

    public Risorsa(String id, int v) {
        nome = id;
        valore = v;
    }

    public synchronized int getValore() {
        return valore;
    }

    public synchronized String getNome() {
        return nome;
    }

    public synchronized Risorsa maxVal(Risorsa ra) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        int va = ra.getValore();
        if (valore >= va)
            return this;
        else
            return ra;
    }
}
```

```java
public class Confronto implements Runnable {
    private Risorsa r1;
    private Risorsa r2;

    public Confronto(Risorsa r1, Risorsa r2) {
        this.r1 = r1;
        this.r2 = r2;
    }

    public void run() {
        String nome = Thread.currentThread().getName();
        System.out.println(nome + " begins");
        Risorsa rm = r1.maxVal(r2);
        String rn = rm.getNome();
        System.out.println(nome + ": la risorsa più grande è " + rn);
        System.out.println(nome + " ends");
    }
}
```

```java
public class MainDeadlock {
    public static void main(String[] args) throws InterruptedException {
        Risorsa r1 = new Risorsa("alfa", 5);
        Risorsa r2 = new Risorsa("beta", 20);
        System.out.println(Thread.currentThread().getName() + " begins");
        Thread t1 = new Thread(new Confronto(r1, r2));
        Thread t2 = new Thread(new Confronto(r2, r1));
        t1.start();
        t2.start();
        Thread.sleep(10000);
        System.out.println(Thread.currentThread().getName() + " ends");
    }
}
```

Con un solo thread `Confronto` il programma stampa che la risorsa più grande è `beta` e termina. Con due thread, che confrontano le stesse risorse in ordine opposto, si blocca:

1. `t1` chiama `alfa.maxVal(beta)`, e così prende il monitor di `alfa`; nello stesso tempo `t2` chiama `beta.maxVal(alfa)`, e prende il monitor di `beta`;
2. entrambi dormono 100 millisecondi;
3. `t1` chiama `beta.getValore()`, che è `synchronized`: aspetta il monitor di `beta`, che ha `t2`;
4. `t2` chiama `alfa.getValore()` e aspetta il monitor di `alfa`, che ha `t1`.

I due thread restano in BLOCKED per sempre. Il `main` stampa `main ends`, ma il programma non termina.

Definizione – Thread demone

Un thread **demone** lavora al servizio degli altri. La macchina virtuale termina quando restano solo thread demoni. Un thread diventa demone con `setDaemon(true)`, chiamato prima di `start()`.

13 Rientranza

Proposizione – I monitor sono rientranti

Un thread che possiede già il monitor di un oggetto può chiamare altri metodi `synchronized` dello stesso oggetto senza aspettare: il monitor è già suo. Lo rilascia quando esce dal primo metodo sincronizzato.

Così un metodo `synchronized` può usare gli altri metodi sincronizzati della classe, e il codice si divide in metodi piccoli senza rischiare che un thread blocchi sé stesso. Per esempio `getValore()` di `Contatore` si può chiamare dentro `incrementa()`.

14 `wait`, `notify` e `notifyAll`

Il monitor impedisce che due thread usino una risorsa insieme, ma non basta a coordinarli. Nello schema **produttore e consumatore** un thread produce dati e un altro li consuma: il consumatore deve aspettare che ci sia un dato, e il produttore che ci sia posto per metterlo.

Definizione – `wait`, `notify`, `notifyAll`

Sono metodi di `Object`, e si chiamano dentro una parte `synchronized` sull'oggetto di cui si possiede il monitor.

- `wait()` sospende il thread nello stato WAITING e **rilascia il monitor**, così altri thread possono entrare. Può lanciare `InterruptedException`.
- `notify()` risveglia uno dei thread in attesa su quell'oggetto, scelto in modo arbitrario.
- `notifyAll()` li risveglia tutti.

Un thread risvegliato deve riacquisire il monitor prima di proseguire.

Proposizione – Attesa di una condizione

Dentro un metodo `synchronized`, un thread che deve aspettare una condizione scrive:

```
while (!condizione)
    wait();
```

e chi rende vera la condizione chiama `notifyAll()`. Il `while`, non un `if`, ricontrolla la condizione dopo ogni risveglio, perché nel frattempo un altro thread può averla resa di nuovo falsa.

Osservazione – Perché `notifyAll`

Con `notify()` si risveglia un solo thread, e può essere quello sbagliato, per esempio un altro produttore quando serviva un consumatore. Quel thread trova la condizione falsa e torna ad aspettare, e nessuno sveglia gli altri: il programma può bloccarsi. Con `notifyAll()` ogni thread in attesa ricontrolla la propria condizione.

15 Produttore e consumatore

Un magazzino riceve 80 pezzi al minuto, uno ogni 750 millisecondi, e ha posto per 20. Due officine li consumano: una fa 6 tavoli al minuto con 9 pezzi ciascuno, l'altra 6 sedie al minuto con 6 pezzi ciascuna. Quando il magazzino è pieno i fornitori aspettano; quando è vuoto aspettano le officine. Il magazzino è una memoria condivisa limitata.

Nel codice il magazzino è una coda di capacità fissa, `BoundedQueue`. Usa un'`ArrayList`, una lista di lunghezza variabile della libreria Java: `add(x)` aggiunge in fondo, `remove(0)` toglie e restituisce il primo elemento, `size()` conta gli elementi. `BoundedQueue<T>` è una coda di elementi di tipo `T`: `T` sta per il tipo scelto da chi crea la coda, qui `String`.

```java
import java.util.ArrayList;
import java.util.List;

public class BoundedQueue<T> {
    private List<T> elements;
    private int maxSize;

    public BoundedQueue(int capacity) {
        elements = new ArrayList<T>();
        maxSize = capacity;
    }

    public synchronized T remove() throws InterruptedException {
        while (elements.size() == 0)
            wait();
        T r = elements.remove(0);
        notifyAll();
        return r;
    }

    public synchronized void add(T newElem) throws InterruptedException {
        while (elements.size() == maxSize)
            wait();
        elements.add(newElem);
        notifyAll();
    }
}
```

`remove` aspetta finché la coda è vuota, `add` finché è piena; ciascuno, dopo aver cambiato la coda, sveglia tutti gli altri. I due metodi sono `synchronized` sulla stessa coda, quindi non si intercalano.

```
public class Producer implements Runnable {
    private String greeting;
    private BoundedQueue<String> queue;
    private int greetingCount;
    private static final int DELAY = 10;

    public Producer(String aGreeting, BoundedQueue<String> aQueue, int count) {
        greeting = aGreeting;
        queue = aQueue;
        greetingCount = count;
    }

    public void run() {
        try {
            int i = 1;
            while (i <= greetingCount) {
                queue.add(i + ": " + greeting);
                i++;
                Thread.sleep((int) (Math.random() * DELAY));
            }
        } catch (InterruptedException exception) {
        }
    }
}
```

```
public class Consumer implements Runnable {
    private BoundedQueue<String> queue;
    private int greetingCount;
    private static final int DELAY = 10;

    public Consumer(BoundedQueue<String> aQueue, int count) {
        queue = aQueue;
        greetingCount = count;
    }

    public void run() {
        try {
            int i = 1;
            while (i <= greetingCount) {
                String greeting = queue.remove();
                System.out.println(greeting);
                i++;
                Thread.sleep((int) (Math.random() * DELAY));
            }
        } catch (InterruptedException exception) {
        }
    }
}
```

```
public class ThreadTester {
    public static void main(String[] args) {
        BoundedQueue<String> queue = new BoundedQueue<String>(10);
        final int GREETING_COUNT = 1000;
        Thread t1 = new Thread(new Producer("Hello, World!", queue,
    GREETING_COUNT));
        Thread t2 = new Thread(new Producer("Goodbye, World!", queue,
    GREETING_COUNT));
        Thread t3 = new Thread(new Consumer(queue, GREETING_COUNT));
        Thread t4 = new Thread(new Consumer(queue, GREETING_COUNT));
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Due produttori mettono 1000 saluti ciascuno in una coda da 10 posti, e due consumatori ne tolgono 1000 ciascuno. Escono tutte le 2000 righe, in un ordine diverso a ogni esecuzione, e il programma termina: nessun saluto va perso e nessuno è letto due volte.

Nota – Le classi pronte di `java.util.concurrent`

La libreria Java offre strutture già sincronizzate. L'interfaccia `BlockingQueue`, con la classe `LinkedBlockingQueue`, è una coda come `BoundedQueue`: `put` aspetta se è piena, `take` se è vuota. `ConcurrentMap`, con `ConcurrentHashMap`, è una mappa che più thread possono usare insieme.

Riepilogo

- **Thread**: il lavoro in `run()` di un `Runnable`; `new Thread(r).start()` lo esegue in un flusso nuovo, con il suo stack e lo heap in comune.
- **Ordine**: lo decide lo scheduler, e cambia a ogni esecuzione; priorità, `yield` e time slicing non danno garanzie.
- **Stati**: NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCKED, TERMINATED.
- **sleep e join**: `Thread.sleep(ms)` sospende il thread corrente, e `t.join()` aspetta che `t` finisca.
- **interrupt**: è solo una richiesta; un thread fermo in `sleep`, `wait` o `join` si sveglia con una `InterruptedException`.
- **synchronized**: un thread alla volta per monitor; i monitor sono rientranti, i metodi non sincronizzati li ignorano.
- **Deadlock**: thread che si aspettano a vicenda i monitor; i demoni non tengono in vita la macchina virtuale.
- **wait e notifyAll**: `while (!cond) wait();` e `notifyAll()` dopo ogni modifica, come in `BoundedQueue`.

16 Esercizi

Esercizio 1 – Zeri e uni

Scrivere una classe `Runnable` che stampa 100 volte, senza andare a capo, un carattere ricevuto nel costruttore, e un `main` che avvia due thread: uno stampa `0`, l'altro `1`. Le sequenze sono uguali a ogni esecuzione?

Soluzione a pagina 27 →

Esercizio 2 – run al posto di start

Nel `main` della Sezione 3 si scrive `ja.run()` al posto di `mioThread.start()`. Che cosa stampa il programma, e quanti thread usa?

Soluzione a pagina 27 →

Esercizio 3 – Si parte troppo presto

Con la classe `CountDown` della Sezione 8, il `main` crea `h` (`"Huston"`, 10) e `c` (`"CapeCanaveral"`, 10), li avvia in due thread `t1` e `t2` e stampa `Si parte!!!`. Dove esce `Si parte!!!`? Come si corregge il `main` perché esca per ultima?

Soluzione a pagina 28 →

Esercizio 4 – Togliere lo stallo

Modificare `maxVal` di `Risorsa` (Sezione 12) in modo che `MainDeadlock` termini, senza togliere `synchronized` da `getValore` e `getNome`. Perché lo stallo scompare?

Soluzione a pagina 28 →

Esercizio 5 – notify al posto di notifyAll

In `BoundedQueue` (Sezione 15) si sostituisce `notifyAll()` con `notify()` in `add` e in `remove`. La coda ha capacità 1, e la usano due produttori `P1` e `P2` e due consumatori `C1` e `C2`. Mostrare una sequenza in cui tutti e quattro i thread restano in attesa.

Soluzione a pagina 29 →

17 Soluzioni

Soluzione dell'Esercizio 1 – Zeri e uni

```
public class PrintExecutable implements Runnable {
    private final char c;

    public PrintExecutable(char c) {
        this.c = c;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.print(c);
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new PrintExecutable('0'));
        Thread t2 = new Thread(new PrintExecutable('1'));
        t1.start();
        t2.start();
    }
}
```

Le sequenze non sono garantite uguali: l'ordine dipende da come vengono assegnati i turni. Spesso però escono cento `0` e poi cento `1`, perché il primo thread finisce prima che il secondo ottenga il turno. Con la `sleep` della Sezione 4 si alternano.

← *Torna all'Esercizio 1, pagina 25*

Soluzione dell'Esercizio 2 – run al posto di start

`ja.run()` è una chiamata di metodo come le altre e gira nel thread del `main`. Il programma usa un solo thread e stampa sempre, in quest'ordine, `Jamaica: 0`, …, `Jamaica: 9`, `Jamaica: DONE!` e per ultima `main: fine`. L'oggetto `mioThread` resta inutilizzato.

← *Torna all'Esercizio 2, pagina 25*

Soluzione dell'Esercizio 3 – Si parte troppo presto

`Si parte!!!` esce all'inizio, prima o subito dopo le prime righe dei conti: dopo le due `start()` il `main` prosegue senza aspettare, mentre i due thread stampano e dormono 100 millisecondi a ogni giro. Il conto dura più di un secondo, la stampa del `main` un istante.

Per correggerlo il `main` aspetta i due thread prima di stampare:

```
try {
    t1.join();
    t2.join();
} catch (InterruptedException e) {
}
System.out.println("Si parte!!!");
```

Ora `Si parte!!!` esce dopo `Huston:  Go!` e `CapeCanaveral:  Go!`.

← *Torna all'Esercizio 3, pagina 26*

Soluzione dell'Esercizio 4 – Togliere lo stallo

Si toglie `synchronized` da `maxVal` e si legge anche il proprio valore con `getValore()`:

```
public Risorsa maxVal(Risorsa ra) {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
    }
    int v = getValore();
    int va = ra.getValore();
    if (v >= va)
        return this;
    else
        return ra;
}
```

Ora ogni thread possiede al massimo un monitor alla volta: `getValore()` prende il monitor di una risorsa e lo rilascia subito. Nessun thread aspetta un monitor tenendone un altro, quindi l'attesa a vicenda non può nascere. Entrambi i thread stampano che la risorsa più grande è `beta` e terminano.

← *Torna all'Esercizio 4, pagina 26*

Soluzione dell'Esercizio 5 – notify al posto di notifyAll

Una sequenza possibile, partendo dalla coda vuota:

1. `C1` e `C2` chiamano `remove`, trovano la coda vuota e vanno in `wait`.
2. `P1` aggiunge un elemento e con `notify()` sveglia `C1`.
3. Prima che `C1` riprenda il monitor, `P2` entra in `add`, trova la coda piena e va in `wait`; lo stesso fa `P1` con il suo elemento successivo.
4. `C1` toglie l'elemento e con `notify()` sveglia `C2`, un altro consumatore, invece di un produttore. Poi richiama `remove`, trova la coda vuota e va in `wait`.
5. `C2` trova la coda vuota e torna in `wait` senza notificare nessuno.

Ora `P1`, `P2`, `C1` e `C2` sono tutti in `wait`, e nessuno può svegliarli. Con `notifyAll()`, al passo 4 si svegliano anche i produttori, e uno di loro trova posto nella coda.

← *Torna all'Esercizio 5, pagina 26*