# 1 Lezione del 21-10-25

Continuiamo a parlare della sincronizzazione, approfondendo il funzionamento dei metodi relativi.

### 1.0.1 Metodo `wait()`

Il metodo `wait()` è definito come `public final void wait() throws InterruptedException`. Il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.

Esiste una versione con timeout di `wait()`. Questa è definita come `public final void wait(long timeout) throws InterruptedException`. In questo caso il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Aspetta i millisecondi di timeout: se li supera esce forzatamente riacquisendo i lock;
- Riacquisisce automaticamente il lock quando esece dalla wait.Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.

La `wait(timeout)` con `timeout` $= 0$ corrisponde alla `wait()` (timeout infinito).

### 1.0.2 Metodo `notify()`

Il metodo `notify()` ha firma `public final void notify()`. Quando viene invocato su un oggetto il primo fra i thread che erano in attesa su tale oggetto (cioè erano nella sua *wait list*, dove si entra invocando `wait()`) viene svegliato.

Il `notifyAll()` è una variante di `notify()`, con firma `public final void notifyAll()`, che sveglia tutti i thread in attesa nella *wait list*.

Notiamo che `wait()`, `notify()` e `notifyAll()` non devono essere per forza chiamati sulla classe madre, ma possono anche essere chiamati su altri oggetti, con la prerogativa che si possieda un lock su tali oggetti.

## 1.1 Produttori e consumatori

Vediamo come applicare questo schema ha uno degli esempi più noti della programmazione concorrente: quello del paradigma dei **produttori** e **consumatori**.

Ipotizziamo una situazione dove esistono $n$ thread, $P_0$, $P_1$, ..., $P_n$ detti *produttori*, ed $m$ thread, $C_0$, $C_1$, ..., $C_m$ detti *consumatori*. Fra produttori e consumatori c'è un certo *buffer*. Il buffer è *condiviso* tra tutti i produttori e i consumatori, ed è in grado di contenere un numero limitato di valori.

Ogni produttore ciclicamente:

- Produce un nuovo valore;
- Lo immette nel buffer.

Chiaramente, se il buffer è pieno, il produttore deve bloccarsi.

Ogni consumatore ciclicamente:

- Preleva un valore dal buffer;
- Lo consuma.

Come sopra, se il buffer è vuoto, il consumatore deve bloccarsi.

Vediamo quindi l'implementazione di questo pattern:

```
class Buffer {
  final int buf[];
  final int size;
  int beg;
  int end;

  Buffer(int size) {
    this.size = size;
    buf = new int[size];
  }

  boolean isFull() {
    return (end + 1) % size == beg;
  }

  boolean isEmpty() {
    return end == beg;
  }

  synchronized void insert(int e) {
    while(isFull()) {
      System.out.println("insert() bloccato da buffer pieno");
      try {
        wait();
      } catch(InterruptedException ex) {}
    }

    System.out.println("insert() di " + e);
    buf[end] = e;
    end = (end + 1) % size;
    notifyAll();
  }

  synchronized int extract() {
    while(isEmpty()) {
```

```
36        System.out.println("extract() bloccato da buffer vuoto");
37        try {
38          wait();
39        } catch(InterruptedException ex) {}
40      }
41
42      int e = buf[beg];
43      beg = (beg + 1) % size;
44      System.out.println("extract() di " + e);
45      notifyAll();
46
47      return e;
48    }
49  }
50
51  class Producer extends Thread {
52    static int c = 0;
53
54    Buffer b;
55    String name;
56
57    Producer(String name, Buffer b) {
58      this.name = name;
59      this.b = b;
60    }
61
62    public void run() {
63      while(true) {
64        // aspetta un tempo casuale
65        try {
66          sleep((long) (Math.random() * 1000));
67        } catch(InterruptedException ex) {}
68
69        // inserisci
70        System.out.println(name + " ha prodotto " + c);
71        synchronized(Producer.class) {
72          b.insert(c++);
73        }
74      }
75    }
76  }
77
78  class Consumer extends Thread {
79    Buffer b;
80    String name;
81
82    Consumer(String name, Buffer b) {
83      this.name = name;
84      this.b = b;
85    }
86
87    public void run() {
88      while(true) {
89        // aspetta un tempo casuale
90        try {
91          sleep((long) (Math.random() * 1000));
92        } catch(InterruptedException ex) {}
93
94        // estrai
95        int c = b.extract();
```

```
96        System.out.println(name + " ha consumato " + c);
97      }
98    }
99  }
100
101 class Main {
102   static void main() {
103     // crea buffer
104     Buffer buf = new Buffer(10);
105
106     // popola produttori
107     for(int i = 0; i < 25; i++) {
108       new Producer("Produttore " + i, buf).start();
109     }
110
111     // popola consumatori
112     for(int i = 0; i < 15; i++) {
113       new Consumer("Consumatore " + i, buf).start();
114     }
115
116     // simula per un po'
117     try {
118       Thread.sleep(5000);
119     } catch(InterruptedException ex) {}
120
121     // termina
122     System.exit(0);
123   }
124 }
```

### 1.2 Differenza fra `notify()` e `notifyAll()`

Approfondiamo la differenza fra i metodi per il risveglio di thread, cioè `notify()` e `notifyAll()`.

Su un oggetto possono essere in attesa più thread (alternativamente, in una *wait list* possono esserci più di un thread).

Se questi thread sono in attesa di condizioni diverse, è opportuno usare la `notifyAll()`. In tal situazione, se si facesse diversamente (usando una semplice `notify()`) si potrebbe infatti avere la sveglia di un solo thread che però non vede le proprie condizioni soddisfatte: nessuno viene effettivamente svegliato, si può incorrere in deadlock.

Se tutti i thread bloccati attendono la stessa condizione, o se solo uno (senza specificare quale) dei thread bloccati può trarre beneficio dalla situazione creatasi, conviene invece usare `notify()`.

Nell'esempio sopra, sul buffer si usa `notifyAll()` anziché `notify()` in quanto produttori e consumatori possono entrambi bloccarsi sui thread, ma aspettando condizioni diverse.