**THREADS – JAVA**

• Multi threading è l’abilità di eseguire diverse parti di codice nello stesso momento

• In java ci sono **2 modi per creare un nuovo thread**:

**1. Creare una classe che estenda la classe thread**:

public class Multithread exend Thread {

}

Questo non basta. Per rendere questa classe “multitrheadable” dobbiamo **eseguire un**   
**@Override del metodo “run” della classe Thread:**

public class Multithread exend Thread {

  //codice

  @Override

  public void run() {

    //codice da eseguire

  }

}

|  |
| --- |
| **METODO RUN**:  Il metodo run() in una classe che estende Thread o implementa l'interfaccia Runnable definisce il comportamento del thread quando viene avviato. Quando un thread viene avviato tramite il metodo start(), il sistema operativo eseguirà il metodo run() di quel thread.  Pertanto, il metodo run() è il punto di ingresso principale per il codice eseguito da un thread. All'interno del metodo run(), puoi definire le attività che il thread deve eseguire, come ad esempio calcoli, operazioni di I/O, o qualsiasi altra operazione che desideri eseguire in modo concorrente. |

Esempio: Supponiamo di avere una classe che estenda thread in cui il metodo run stampi i numeri da 1 a 5  
 attendendo un secondo fra una stampa e l’altra

public class Multithread exend Thread {

  @Override

  public void run() {

    for (int i = 1; i <= 5; i++) {

      System.out.println(i);

      try {

        Thread.sleep(1000);

      } catch (InterruptedExeption e) {

        System.out.println(e);

      }

}

  }

}

Vediamo ora come implementarla nel nostro main method:

public class Program {

  public static void main(String[] args) {

    // Creo un oggetto di tipo Multithread

    Multithread t1 = new Multithread();

    /\*NOTA: Potrei pensare che, avendo Multithread un metodo "run" mi basti

     \*      scrivere ti.run(). Invece no, perchè questo NON farebbe partire il

     \*      metodo in un thread separato, ma lo eseguirebbe nel thread principale.

     \*     Per far partire il metodo in un thread separato, devo usare il metodo:

     \*

    Questo metodo fa partire il metodo "run" in un thread separato

    NOTA: Una volta avviato il secondo thread, il thread principale continuerà

          l'esecuzione con le linee successive di codice, mentre il secondo

          eseguirà il metodo "run" definito nella classe Multithread.\*/

     t1.start();

     //altro codice

  }

}

Proviamo ora ad aggiungere diversi threads con un ciclo for

public class Program {

  public static void main(String[] args) {

    for (int 1 = 0; i < 5; i++) {

      Multithread t = new Multithread();

      t.start();

    }

  }

}

|  |
| --- |
| NOTA: Andando a stampare quale thread viene eseguito durante l’esecuzione del programma ci   accorgeremmo che l’ordine di esecuzione sarà diverso dall’ordine di scrittura  (il terzo potrà eseguire prima del secondo etc)  NOTA: Se uno dei threads dovesse fallire nell’esecuzione e lanciare un execption, questo non avrà alcun  effetto sugli altri. Solo quel singolo thread smetterebbe di essere eseguito.  Questo vale anche se l’exception venisse lanciata nel thread principale (il main method). Gli   altri continuerebbero la loro normale esecuzione fino alla conclusione |

**1. Creare una classe che implementi l’interfaccia “Runnable”:**

• In questo caso, l’unica cosa da fare, sarà creare una nostra implementazione del metodo “run”

public class Multithread implement Runnable {

  @Override

  public void run() {

    for (int i = 1; i <= 5; i++)

      System.out.println(i);

      try {

        Thread.sleep(1000);

      } catch (InterruptedExeption e) {

        System.out.println(e);

      }

  }

}

• Questa volta però, per poterla usare, non potrò chiamare il metodo “start” perché non sto estendendo   
 “Thread”. Quello che dovrò fare sarà:

- Creare il mio oggetto di tipo Multithread  
 - Creare un nuovo oggetto della classe predefinita Thread, passando al suo costruttore l’oggetto   
 che implementa la classe Runnable  
 - su questo oggetto thread lanciare il metodo start

public class Program {

  public static void main(String[] args) {

    for (int 1 = 0; i < 5; i++) {

      Multithread t = new Multithread();

      Thread myThread = new Thread(t);

      myThread.start();

    }

  }

}

NOTA: Perché dovrei implementare Runnable invece di estendere Thread ?  
 Perché estendendo Thread non potrei estendere nessun’altra classe (mentre posso implementare   
 diverse interfacce).

**JOIN method**

Esempio:

public class Program {

  public static void main(String[] args) {

    for (int 1 = 0; i < 5; i++) {

      Multithread t = new Multithread();

      Thread myThread = new Thread(t);

      myThread.start();

      myThread.join();

    }

  }

}

Quello che fa il metodo join è interrompere l’esecuzione del thread in cui è stato chiamato fino  
al completamento del thread chiamante  
(in questo caso il main method aspetterà l’esecuzione del thread myThread prima di continuare con  
la sua esecuzione)

**ISALIVE method**

Esempio:

public class Program {

  public static void main(String[] args) {

    for (int 1 = 0; i < 5; i++) {

      Multithread t = new Multithread();

      Thread myThread = new Thread(t);

      myThread.start();

      myThread.isAlive();

    }

  }

}

Restituisce un booleano che indica se il thread è ancora in esecuzione oppure no

**ALTRI METODI**

**Thread.activeCount()**: Mi dice quanti threads sono attivi

**Thread.currentThread().getName()**: Mi restituisce il nome del thread attivo  
 Questo va usato all’interno di ogni classe thread se voglio  
 in nome di quel thread

Es:

public class Main {

  public static void main(String[] args) {

      // Avvio di un nuovo thread

      Thread otherThread = new Thread(new MyRunnable());

      otherThread.start();

      // Chiamata a Thread.currentThread() nel metodo main

      Thread mainThread = Thread.currentThread();

      System.out.println("Thread principale: " + mainThread.getName());

  }

}

class MyRunnable implements Runnable {

  public void run() {

      // Chiamata a Thread.currentThread() all'interno di un altro thread

      Thread otherThread = Thread.currentThread();

      System.out.println("Altro thread: " + otherThread.getName());

  }

}

In questo esempio, la chiamata a   
**Thread.currentThread()** all'interno di **main()** restituirà il thread principale, mentre la chiamata a **Thread.currentThread()** all'interno di **run()** restituirà il thread appena avviato (non il thread principale).

**Thread.currentThread().getPriority()**: Mi restituisce la priorità del thread in cui viene chiamato  
 Più è alto il valore restituito maggiore sarà la priorità

**Thread.currentThread().setPriority(**numero da 1 a 10**)**: Mi restituisce la priorità del thread in cui viene  
 chiamato Più è alto il valore restituito maggiore   
 sarà la priorità

**CUNCURRENCY & SYNCHRONIZATION**

• Come abbiamo visto, facendo partire diversi threads, si potrebbe entrare in condizioni di race condition  
 o semplice esecuzione simultanea con risultati inattesi sulla precedenza dell’esecuzione del singolo thread

• Supponiamo ora di voler creare un programma in cui ogni thread possa accedere ed eseguire la parte   
 critica del codice (la parte di codice che vogliamo venga eseguita un thread alla volta) solo dopo che un   
 thread ne abbia già concluso l’esecuzione.  
 In sostanza ho bisogno di un modo per verificare che, se un thread sta eseguendo l’operazione, un altro  
 possa cominciare solo dopo che il primo l’ha portata a termine.

• Vediamo diversi modi per singronizzare i threads:

**1. SYNCHRONIZED** – inserire la keyword “synchronized” nel blocco di codice da proteggere

1) La prima cosa da fare è capire quale è la sezione critica del codice (quella che vogliamo proteggere).  
2) Aggiungere la keyword “synchronized”

public class Restaurant {

    static class Kitchen implements Runnable {

        public static synchronized void preparedFood() { //critical Section

            System.out.println("Food preparation started");

            System.out.println(Thread.currentThread().getName() + "is preparing);

            try {

                Thread.sleep(2000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

            System.out.println("Food preparation completed");

        } //fine della parte critica

            @Override

            Public void run() {

                System.out.println("\n\nOrder taken");

                preparedFood();

            }

        }

        public static void main(String[] args) {

            Ktichen Kitchen = new Kitchen();

            Thread thread1 = new Thread(Kitchen);

            thread1.start();

            Kitchen Kitchen2 = new Kitchen();

            Thread thread2 = new Thread(Kitchen2);

            thread2.start();

            Kitchen Kitchen3 = new Kitchen();

            Thread thread3 = new Thread(Kitchen3);

            thread3.start();

        }

    }

- In questo esempio stiamo simulando una cucina. La parte da proteggere del codice è il metodo che simula la cucina, in cui un solo thread alla volta può accedere. Per proteggerlo usero “synchronized”:

public static synchronized void preparedFood() {

Syncrhronized aggisce da “monitor” e fa si che un thread possa accedere a quella specifica parte di codice   
solo quando un altro ne ha terminato l’esecuzione.

Nello specifico avrò come output:

Order taken (dal thread 0)  
Food preparation started  
Thread 0 is preparing

Order taken (dal thread 1)

Order taken (dal thread 2)  
Food preparation **complete** (dal thread 0)  
Food preparation started  
Thread 2 is preparing  
Food preparation **complete** (dal thread 2)  
Food preparation started  
Thread 1 is preparing  
Food preparation **complete** (dal thread 1)

Possiamo Vedere che tutto avviene simpultanamente TRANNE la parte di codice che abbiamo protetto .  
Infatti

“Order taken” viene eseguito in simultanea

Mentre la food preparation started viene lanciata solo quando il precedente ha stampato “complete”

**1. SYNCHRONIZED** – Creare un blocco di codice “synchronized”

A differenza del metodo precedente, un secondo modo per sincronizzare i thread, è isolare solo una parte  
di codice critico e non l’intero metodo

public class Restaurant {

    static class Kitchen implements Runnable {

static String lock = "lock"

        public static void preparedFood() { //critical Section

            System.out.println("Whitin the kitchen");

            synchronized(lock) { //inizio dello snippet sinconizzato <<<<<<<<<<

                System.out.println(Thread.currentThread().getName() + "preparing);

                try {

                    Thread.sleep(2000);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

                System.out.println("Food preparation completed");

            } //fine dello snippet sincronizzato <<<<<<<<<<<<<<<<<<<<

        }

            @Override

            Public void run() {

                System.out.println("\n\nOrder taken");

                preparedFood();

            }

        }

        public static void main(String[] args) {

            (...)

        }

    }

Possiamo vedere come in questo caso non abbiamo sincronizzato l’intero blocco, ma solo una sua parte.  
Ad esempio System.out.println("Whitin the kitchen"); non sarà protetto dal multitreading  
come neanche la prima chiamata al metodo run.

Tutti i trheads potranno accedere al metodo preparedFood (chiamato da run) simultaneamente ma  
solo una alla volta nella parte inclusa in   
synchronized() {

…

}

• **Argomento del “synchronized”: “monitor lock”**   
 - Quando “circondiamo” un blocco di codice con il meodo “synchronized” dobbiamo passare a questa   
 funzione un argomento di tipo **oggetto**

Di solito**, l'oggetto lock** viene scelto in base al contesto dell'applicazione**. Può essere qualsiasi oggetto Java valido**, anche se è comune utilizzare oggetti come this (se il blocco di codice sincronizzato è all'interno di un metodo di istanza) o oggetti specifici creati appositamente per la sincronizzazione. **L'importante è che tutti i thread che devono sincronizzare l'accesso al blocco di codice utilizzino lo stesso oggetto lock**.

Quando un thread entra in un blocco di codice sincronizzato, acquisisce il lock associato a quell'oggetto specifico o a quell'oggetto specifico di cui è stato chiamato il metodo sincronizzato. Questo significa che il thread ha il controllo esclusivo su quel blocco di codice e nessun altro thread può accedervi finché il lock non viene rilasciato.

Se un altro thread tenta di accedere allo stesso blocco di codice sincronizzato mentre il lock è già acquisito da un altro thread, viene posto in stato di "attesa". Il thread in attesa rimane in attesa finché il lock non viene rilasciato dal thread che lo detiene. Una volta che il lock viene rilasciato, il sistema notifica il thread in attesa e gli consente di tentare di acquisire il lock e quindi eseguire il blocco di codice sincronizzato.

Quando il thread che ha acquisito il lock esce dal blocco di codice sincronizzato, rilascia il lock. Ciò consente ad altri thread in attesa di acquisire il lock e di eseguire il blocco di codice sincronizzato in modo concorrente. In questo modo, il meccanismo di sincronizzazione in Java garantisce che solo un thread alla volta possa eseguire il blocco di codice sincronizzato, mantenendo la coerenza dei dati condivisi e prevenendo potenziali problemi di concorrenza.

NOTA: In un metodo sincronizzato non statico, al posto di un oggetto specifico, posso usare anche “this”  
 come lock (in questo caso si fa riferimento direttamente all’istanza corrente dell’oggetto)

**COMUNICAZIONDE INTERA-THREAD: CONSUMER PRODUCER PROBLEM**

1. **Wait() / notify()**

• Il problema del produttore-consumatore, noto anche come problema di buffer limitato, è un classico esempio di un problema di sincronizzazione multithreading. Coinvolge due tipi di thread: il produttore, che produce dati, e il consumatore, che consuma dati. Entrambi i thread condividono uno spazio di memoria comune, chiamato buffer o coda, che è limitato in dimensione.

Ecco un esempio di come potrebbe apparire il problema:

1. Il produttore produce dati e li inserisce nel buffer. Se il buffer è pieno, il produttore deve aspettare finché non c'è spazio disponibile nel buffer.
2. Il consumatore consuma dati dal buffer. Se il buffer è vuoto, il consumatore deve aspettare finché non sono disponibili dati nel buffer.

Per risolvere questo problema in modo corretto, è necessario sincronizzare l'accesso al buffer in modo che il produttore e il consumatore non accedano simultaneamente al buffer nello stesso momento. Questo può essere fatto utilizzando tecniche di sincronizzazione come i monitor, i lock o i semafori.

• Un modo per stabilire una comunicazione fra più trheads è quello di implementare la tecnica del  
 **wait()** & **notify()**.

|  |
| --- |
| **QUEUE**  In Java, Queue è un'interfaccia che rappresenta una **struttura dati di tipo coda**. Una coda è una collezione di elementi che segue il principio "First In, First Out" (FIFO): gli elementi vengono inseriti alla fine della coda e vengono rimossi dalla testa della coda (il primo elemento ad arrivare sarà il primo ad essere rimosso)  L'interfaccia Queue estende l'interfaccia Collection e fornisce metodi per l'inserimento, la rimozione e l'ispezione degli elementi nella coda.  Nel codice useremo: Queue<Intiger> queue = new LinkedList<Intiger>():  Questa riga di codice in Java crea una variabile di tipo **Queue** che punta a un'istanza di **LinkedList** contenente oggetti di tipo **Integer**. |

Vediamo un primo esempio di implementazione:

La logica del problema prevede che il producer inizializzerà la Queue con un

 \* valore, mentre il consumer lo preleverà. Se la Queue è vuota il consumer dovrà

 \* attendere che il producer inserisca un valore. Se la Queue è piena il producer

 \* dovrà attendere che il consumer prelevi un valore.

public class ProducerConsumer {

    public static void main(String[] args) {

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Creo un lock visibile a tutti i thread\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/

        Object key = new Object();

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Creo la struttura dati condivisa dai trhead\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/

        Queue<Integer> queue = new LinkedList<Intiger>();

/\*\_\_\_\_\_\_\_\_\_\_Dichiaro la dimensione della struttura dati condivisa\_\_\_\_\_\_\_\_\_\_\_\_ \*/

        final int MAX\_SIZE = 10;

/\*\_\_\_\_\_\_\_\_\_Creiamo due trhead, uno per il producer ed uno per il consumer\_\_\_\_\_\*/

Trhead producer = new Trhead(new Runnable() {

    @Override

    public void run() {

        /\*La logica del producer e consumer si basa su una costante

        \* offerta e richiesta, quindi entrambe le logiche dovranno essere

        \* basate su un ciclo\*/

        while (true){

            /\*variabile che inserirò nel queue \*/

            int count = 0;

            /\*blocco da proteggere per il producer \*/

            synchronized(key) {

                try {

                    /\*se la queue è piena il producer dovrà attendere\*/

                    while (queue.size() == MAX\_SIZE)

                        key.wait();

                    queue.offer(count++); /\*inserisco un valore nella Queue\*/

                    key.notifyAll();    /\*notifico a tutti i thred che

                                          la queue è stata modificata\*/

                    Thread.sleep(1000);   /\*attendo 1 secondo\*/

                    System.out.println("Produced emelents: " + queue.size());

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

});

Trhead consumer = new Thread(new Runnable() {

    @Override

    public void run() {

        while (true) {

            /\*blocco da proteggere per il consumer \*/

            synchronized(key) {

                try {

                    /\*se la queue è vuota il consumer dovrà attendere \*/

                    while (queue.size() == 0)

                        key.wait();     /\*se la queue è vuota il consumer dovrà

                                          attendere\*/

                    queue.poll();       /\*prelevo un valore dalla Queue\*/

                    key.notifyAll();    /\*notifico a tutti i thred che

                                         la queue è stata modificata\*/

                    Trhead.sleep(1000); /\*attendo 1 secondo\*/

                    System.out.println("Element consumed, Remaning: " +   
 queue.size());

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

});

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_lancio i treads\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/

        producer.start();

        consumer.start();

    }

}

**2.**

Un secondo modo per implementare la stessa comunicazione è quello di utilizzare una diversa struttura dati

Supponiamo di voler creare un programma con due threads. Uno che stampi i numeri dispari ed un altro che stampi i pari in maniera alternata e sincronizzata (prima stampa uno poi stampa l’altro)

Dividiamo il problema:  
1. Stampare i numeri in sequenza con l’aiuto del multitrheading

/\*Classe che stampa numeri \*/

class Printer implements Runnable {

    static int counter = 1;

    public void printer() {

        System.out.println(counter++);

    }

    @Override

    public void run() {

        for (int i = 0; i < 10; i++) {

            printer();

        }

    }

}

public class PrintingOddAndEvenNumbers {

    public static void main(String[] args) {

        Printer evenObj = new Printer();

        Printer oddObj = new Printer();

        Thread even = new Thread(evenObj);

        Thread odd = new Thread(oddObj);

        even.start();

        odd.start();

    }

}

- Rendendo la variabile int counter “static” potrò fare in modo che ogni thread stampi un numero solo una volta:

COUNTER NON STATIC  
Thread-0 1  
Thread-1 1  
Thread-1 2  
Thread-0 2  
Thread-1 3  
Thread-0 3  
Thread-1 4  
Thread-0 4  
Thread-1 5  
Thread-0 5  
Thread-1 6  
Thread-0 6  
Thread-1 7  
Thread-0 7  
Thread-1 8  
Thread-0 8  
Thread-1 9  
Thread-0 9  
Thread-1 10  
Thread-0 10

COUNTER STATIC  
Thread-0 1  
Thread-1 2  
Thread-0 3  
Thread-1 4  
Thread-0 5  
Thread-1 6  
Thread-0 7  
Thread-1 8  
Thread-0 9  
Thread-1 10  
Thread-1 11  
Thread-0 11  
Thread-1 12  
Thread-0 13  
Thread-1 14  
Thread-0 15  
Thread-1 16  
Thread-0 16  
Thread-1 17  
Thread-0 18

2. Stampare i numeri pari con il thread pari ed i dispari con il dispari

/\*Classe che stampa numeri \*/

public class Printer implements Runnable {

    static int counter = 1;

    int reminder;

    static Object lock = new Object();

    public void Printer(int reminder) {

        this.reminder = reminder;

    }

    public void printer() {

        System.out.println(Thread.currentThread().getName() + " " + counter++);

        try {

            Thread.sleep(500);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void run() {

        for (int i = 0; i < 10; i++) {

            synchronized (lock) {

                while (counter % 2 != reminder) {

                    try {

                        lock.wait();

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    }

                }

                printer();

                lock.notifyAll();

            }

        }

    }

}

public class PrintingOddAndEvenNumbers {

    public static void main(String[] args) {

        Printer evenObj = new Printer(0);

        Printer oddObj = new Printer(1);

        Thread even = new Thread(evenObj);

        Thread odd = new Thread(oddObj);

        even.start();

        odd.start();

    }

}

- Ho creato due thread che assegnato (con il costruttore) il valore 0 o 1 a “reminder”

        Printer evenObj = new Printer(0);

        Printer oddObj = new Printer(1);

- Supponiamo che counter sia = 4:

- Un thread entra nella classe printer.   
 Arrivo al while e la prima cosa che controllo è se il resto di counter / 2 sia uguale al valore di reminder   
 (inizializato a 0 o 1 a seconda del thread pari o dispari)

- essendo counter = 4, quando il thread dispari entrerà il suo reminderà sarà = 1 🡪 entrerà nel ciclo while  
 (perché 4 % 2 = 0 che è != 1) ed incontrerà il wait() 🡪 rimarrà in attesa

- Qundo il thread pari entra nella classe arriverà anche lui al while ma lo schipperà arrivando a:

                printer();

                lock.notifyAll();

NOTA: Il thread dispari, rimasto in attesa, potrà continuare la sua esecuzione solo dopo che il trhear pari  
 ha eseguito questo notify !

NOTA: **static Object lock**

Nel codice fornito, l'oggetto **lock** viene utilizzato come un monitor per sincronizzare l'accesso al blocco di codice critico all'interno del metodo **run()** della classe **Printer**. Poiché **lock** è utilizzato in un contesto statico, ovvero è dichiarato come **static Object lock**, deve essere statico per essere condiviso tra tutte le istanze della classe **Printer**.

Quando un campo è dichiarato come statico, esiste una sola copia di tale campo per l'intera classe, indipendentemente da quante istanze della classe vengono create. Ciò significa che quando si utilizza un blocco di sincronizzazione come **synchronized(lock)**, questo blocco di sincronizzazione acquisisce il lock associato all'oggetto **lock** che è condiviso da tutte le istanze della classe **Printer**.

Se **lock** non fosse dichiarato come statico, ogni istanza di **Printer** avrebbe il proprio oggetto **lock**, e i blocchi **synchronized(lock)** non avrebbero l'effetto desiderato di sincronizzazione tra le diverse istanze.

In breve, **lock** è dichiarato come statico per garantire che tutte le istanze della classe **Printer** condividano lo stesso oggetto **lock**, consentendo la sincronizzazione corretta tra i thread che utilizzano quelle istanze.