PROGRAMMAZIONE CONCORRENTE

Risorse video per questi appunti :

video 1: https://www.youtube.com/watch?v=zaVL2gK7qkc&list=PL-NrWrNHrd0rte-EP8ZeBlnrHBHcoFkGB&index=5

video 2: https://www.youtube.com/watch?v=HRd0icAJ6Qs&list=PL-NrWrNHrd0rte-EP8ZeBlnrHBHcoFkGB&index=6

video 3: https://www.youtube.com/watch?v=ZsPy_92NM_c&list=PL-NrWrNHrd0rte-EP8ZeBlnrHBHcoFkGB&index=7

video 4: https://www.youtube.com/watch?v=BZXXKmyUDMI&list=PL-NrWrNHrd0rte-EP8ZeBlnrHBHcoFkGB&index=9

video 5: https://www.youtube.com/watch?v=BZXXKmyUDMI&list=PL-NrWrNHrd0rte-EP8ZeBlnrHBHcoFkGB&index=9

Quando si è parlato (il terzo anno) del gestore dei processi di un SO (Sistema Operativo) si è visto che in un SO multithreading un singolo processo può essere formato da più thread. I thread sono parti di un unico processo separati ed attivi contemporaneamente. (Si indica con il termine **multitasking** l'esecuzione parallela di più compiti da parte del processore, il termine **multithreading** indica l'esecuzione parallela ma riferita ai thread dei vari processi in esecuzione). Ricordiamo che, nel caso di piattaforme HW basate su più processori o su processori multicore vi è la possibilità di eseguire in maniera **realmente** parallela i thread (e i processi) diversi con conseguente notevole miglioramento delle prestazioni, invece con un solo processore il parallelismo è "apparente".

Ad esempio: in un word processor un thread può occuparsi della gestione dei dati di input e della formattazione del testo, e un altro thread della correzione automatica. I due thread lavorano sugli stessi dati, utilizzano lo stesso file, ma fanno cose diverse. Sono quindi due "componenti dello stesso processo" che fanno cose diverse, e vengono eseguiti dal processore in modalità multithreading, ossia garantendo l'avanzamento in parallelo di tutti i thread.

Si può rappresentare il legame fra un processo e i suoi thread con una metafora dove il processo è rappresentato come una corda e i singoli fili intrecciati di cui essa è costituita rappresentano i thread. Tutti i thread, insieme, contribuiscono alla corretta esecuzione del processo.

Ma da chi vengono creati i thread? Vengono creati dal programmatore che realizza l'applicazione suddividendo "ciò che deve fare l'applicazione" in parti diverse di codice che costituiscono, appunto, i thread (vedremo in seguito come fare questo in Java).

Nei programmi visti e realizzati finora, le istruzioni venivano sempre eseguite sequenzialmente una dopo l'altra. Quando si programma suddividendo il codice di una stessa applicazione in thread la situazione cambia. Il fatto che i thread condividano delle risorse, ad esempio delle zone di memoria che possono essere scritte o lette da entrambi i thread, genera dei problemi di sincronizzazione nella scrittura di programmi con più thread, in seguito vedremo questi problemi e come risolverli.

PROGRAMMAZIONE MULTITHREAD IN JAVA

A differenza di altri linguaggi di programmazione, JAVA ha avuto sin dalla sua prima versione un supporto nativo per la programmazione multithreading. La programmazione con più thread può essere effettuata grazie **all'interfaccia Runnable**.

Per creare un **thread** in Java è necessario creare una classe che implementa l'interfaccia **Runnable** (L'istruzione da aggiungere accanto alla dichiarazione della classe è **implements Runnable**), chiamiamo questa classe, ad esempio, **ClasseX**. Nella **ClasseX è obbligatorio che venga** ridefinito il metodo **run()** con la seguente firma:

public void run()

Perché è obbligatorio? E' obbligatorio perché la classe implementa l'interfaccia **Runnable**. L'interfaccia serve proprio per questo, per obbligare una classe ad implementare dei metodi, in questo caso il metodo run (). All'interno del metodo run si scrive il codice che si vuole venga eseguito dal thread.
Per creare un thread, nella main class si istanzia un oggetto della classe **Thread** e gli si passa come parametro un'istanza della ClasseX. Nella classe main sarà dunque possibile istanziare più thread ciascuno contenente un'istanza della **ClasseX.** Per avviare il thread nella main class è necessario invocare sul thread il metodo **start**() che, quando viene invocato provoca l'esecuzione del metodo run() della **ClasseX** in modalità **multithreading**. ATTENZIONE: QUINDI PER ESEGUIRE IL THREAD IL METODO DA INVOCARE è start(), non run()!

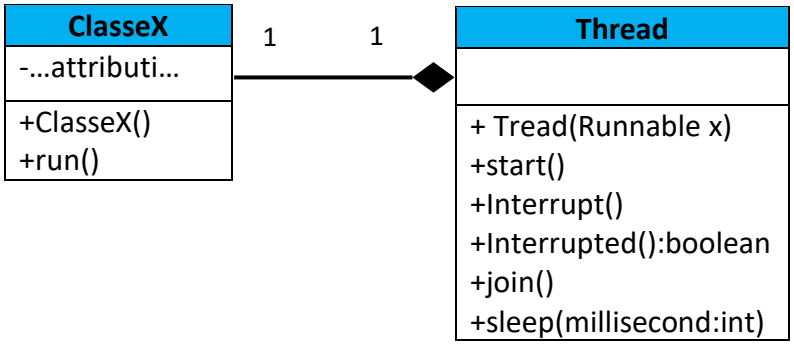
Altri metodi della classe **Thread:**

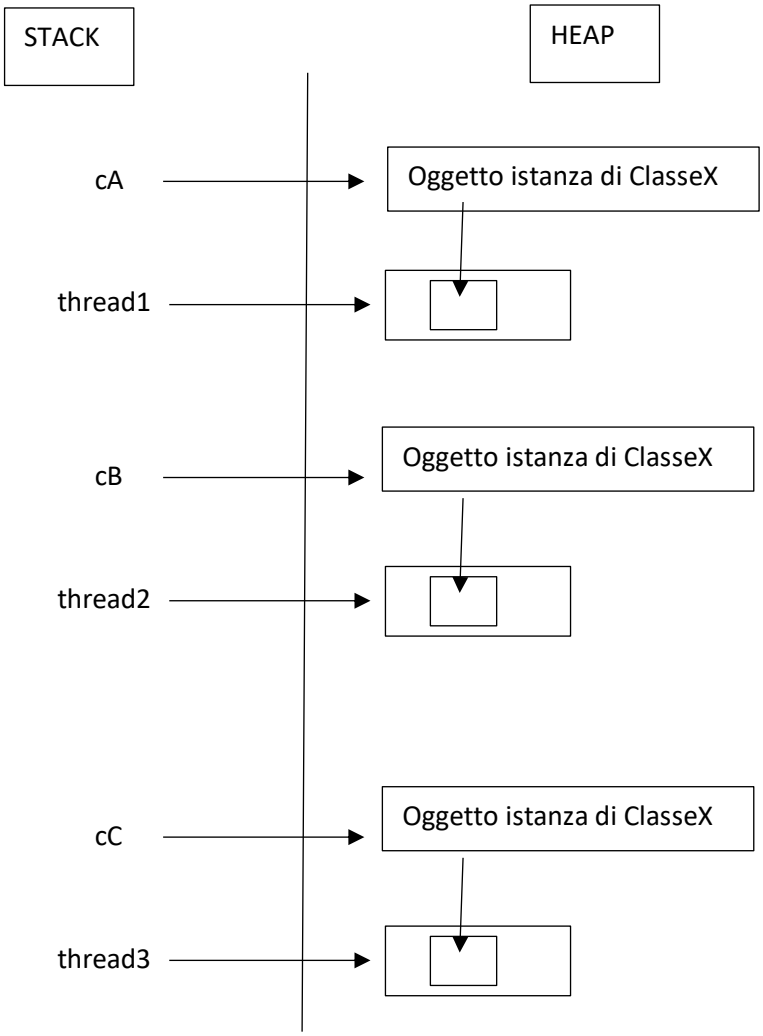
- **Interrupt()**: interrompe un thread
- **Interrupted()**: verifica lo stato di "interrupted" di un thread, restituisce un valore booleano, nel caso tale stato sia true, l'invocazione di questo metodo pone nuovamente lo stato "interrupted" a false.
- **Join()**: quando all'interno del codice di un thread A viene invocato, su un thread B, il metodo B.join(), il thread A **non prosegue** l'esecuzione finchè il thread B è terminato (ossia finchè è concluso il metodo run() di B). Si può dire che A, per proseguire, attende che sia terminata l'esecuzione di B.
- **Sleep** (milliseconds): arresta l'esecuzione del thread per alcuni millisecondi.

La tecnica di esecuzione di un thread è sempre la stessa: nel run della **ClasseX** si realizza un ciclo (for o while) dal quale si esce quando il thread viene interrotto.

```
public void run()
{
        while(!Thread.interrupted())
        {
        ....
        }
}
```

L’interruzione viene svolta nel metodo main() invocando il metodo interrupt() sul thread da interrompere.





Invocando in una MainClass il metodo start() sui 3 thread istanziati, il flusso di esecuzione eseguirà contemporanemante il codice del metodo main della main class e il codice del metodo run() del thread1, del thread2 e del thread3

```
public MainClass
{
        public void main (String[] args)
        {
                ClasseX cA=new ClasseX();     //Istanzio i tre oggetti di classe x
                ClasseX cB=new ClasseX();
                ClasseX c3=new ClasseX();

                Thread1 thread1= new Thread(cA); //Istanzio i tre thread
                Thread2 thread2= new Thread(cB);
                Thread3 thread3= new Thread(cC);

                thread1.start();          //Avvio i tre thread
                thread2.start();
                thread3.start();

        }
}
```

codice eseguito “contemporneamente” dal processore			
metodo main() di App	metodo run() di thread1	metodo run() di thread2	metodo run() di thread3
			

Esempio1 per vedere la differenza fra programmazione multithread e no:

1. Creo la classe ClasseX che scrive per 30 volte una lettera (attributo String di input è la lettera)

```
public class ClasseX
{
    private char lettera;

    public ClasseX(char l)
    {
        this.lettera=l;
    }

    /*
    Il codice con le istruzioni che dovrà eseguire questa classe vengono scritte all'interno di questo metodo run
    Questo sarà il codice eseguito invocando il metodo start su un thread che contiene un'istanza di questa ClasseX
    */
    public void run()
    {
        /*
        Il codice eseguito dal metodo run è sempre un ciclo che si ripete
        per 30 volte scrivendo la lettera
        */
        for(int i=0;i<30;i++)
            System.out.print(c: lettera);

    }
}
```

Nella classe App istanzio I tre oggetti di classe x (**per ora non sono thread**) con parametri “A”,”B”,”C” Ed eseguo Uno dopo l’altro i rispettivi metodi run

```
public class App
{
    public static void main(String[] args)
    {
        /*
        Istanzio tre oggetti di ClasseX
        Ciascuno con una lettera diversa
        */
        ClasseX cA= new ClasseX(1: 'A');
        ClasseX cB= new ClasseX(1: 'B');
        ClasseX cC= new ClasseX(1: 'C');

        /*Come primo esempio eseguo il metodo run di ciascun oggetto
        Classe x senza istanziare Thread (Esecuzione sequenziale )
        */
        cA.run();
        cB.run();
        cC.run();
    }
}
```

Eseguendo il metodo **run()** di ciascuna istanza, vedo l’ output dell’ esecuzione sequenziale

AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCC

2. Ora per provare l'esecuzione multitasking modifico la classe x facendo sì che essa implementi l'interfaccia Serializable.

   Nella classe App eseguo Istanzio tre thread, A ciascuno dei quali passo un'istanza di ClasseX, e li avvio invocando **start() anziché run()**, In questo modo vedo l'ouput dell'esecuzione parallela:

ABCABCABCACBBABCBCAACB ABCABCABCACBBABCBCAACBABCABCABCACB

```
 * @author Gian
 * Questa classe non fa altro che stampare sul monitor una lettera
 * La lettera da stampare è l'unico attributo della classe e viene assegnata
 * a tale attributo nel costruttore.
 * Lo scopo è quello di Eseguire in parallelo con tre thread diversi
 * Tre istanze di questa classe ClasseX
 */
public class ClasseX implements Runnable
{
    private char lettera;

    public ClasseX(char l)
    {
        this.lettera=l;
    }

    /*
    Il codice con le istruzioni che dovrà eseguire questa classe vengono scritte all'interno di questo metodo run
    Questo sarà il codice eseguito invocando il metodo start su un thread che contiene un'istanza di questa ClasseX
    */
    public void run()
    {
        /*
        Il codice eseguito dal metodo run è sempre un ciclo che si ripete
        per 30 volte scrivendo la lettera
        */
        for(int i=0;i<30;i++)
            System.out.print(c: lettera);

    }
}
```

```
public class App
{
    public static void main(String[] args)
    {
        /*
        Istanzio tre oggetti di ClasseX
        Ciascuno con una lettera diversa
        */
        ClasseX cA= new ClasseX(l: 'A');
        ClasseX cB= new ClasseX(l: 'B');
        ClasseX cC= new ClasseX(l: 'C');

        /*Istanzio tre thread a ciascuno dei quali passo come parametro
        La classe che voglio eseguire all'interno di tale thread
        */
        Thread tA=new Thread(task: cA);
        Thread tB=new Thread(task: cB);
        Thread tC=new Thread(task: cC);

        tA.start();
        tB.start();
        tC.start();
    }
}
```

3. Modifico il metodo main della MainClass per mostrare che con il metodo join() l'esecuzione torna sequenziale (per mostrare cosa fa join, attende la fine dell'esecuzione di un thread prima di proseguire).

   tA.start()

   tA.join()

   tB.start()

   tB.join()

   tC.start()

```
11  public class App
12  {
13      public static void main(String[] args) throws InterruptedException
14      {
15          /*
16          Istanzio tre oggetti di ClasseX
17          Ciascuno con una lettera diversa
18          */
19          ClasseX cA= new ClasseX(1: 'A');
20          ClasseX cB= new ClasseX(1: 'B');
21          ClasseX cC= new ClasseX(1: 'C');
22
23          /*Istanzio tre thread a ciascuno dei quali passo come parametro
24          La classe che voglio eseguire all'interno di tale thread
25          */
26          Thread tA=new Thread(task: cA);
27          Thread tB=new Thread(task: cB);
28          Thread tC=new Thread(task: cC);
29
30          tA.start();
            tA.join();
32          tB.start();
33          tB.join();
34          tC.start();
35
36      }
37  }
```

Il thread Tb verrà avviato solamente al termine del thread Ta

Il thread Tc verrà avviato solamente al termine del thread Tb

4. Modifico ClasseX introducendo un ciclo infinito. Modifico il Main() della Main class in modo che quando l'utente inserisce una stringa di input i tre thread vengono interrotti. Per interrompere il thread si invoca il metodo Interrupt() della classe thread.

```java
public class ClasseX implements Runnable
{
    private char lettera;

    public ClasseX(char l)
    {
        this.lettera=l;
    }

    /*
    Il codice con le istruzioni che dovrà eseguire questa classe vengono scritte all'interno di questo metodo run
    Questo sarà il codice eseguito invocando il metodo start su un thread che contiene un'istanza di questa ClasseX
    */
    public void run()
    {
        /*
        Il codice eseguito dal metodo run è
        Un ciclo infinito che viene termina solo quando
        Il thread che contiene l'istanza di ClasseX
        Viene interrotto invocando Interrupt()
        */
        while(!Thread.interrupted())
            System.out.print(c: lettera);

    }
}
```

Source History

```java
public class App
{
    public static void main(String[] args) throws InterruptedException
    {
        /*
        Istanzio tre oggetti di ClasseX
        Ciascuno con una lettera diversa
        */
        ClasseX cA= new ClasseX(l: 'A');
        ClasseX cB= new ClasseX(l: 'B');
        ClasseX cC= new ClasseX(l: 'C');

        /*Istanzio tre thread a ciascuno dei quali passo come parametro
        La classe che voglio eseguire all'interno di tale thread
        */
        Thread tA=new Thread(task: cA);
        Thread tB=new Thread(task: cB);
        Thread tC=new Thread(task: cC);

        tA.start();
        tB.start();
        tC.start();

        Scanner tastiera=new Scanner(source:System.in);
        //Il thread della classe App rimane in attesa
        //dell'Input dell'utente.Quando questo
        //input arriva i tre thread vengono interrotti

        String x=tastiera.nextLine();
        tA.interrupt();
        tB.interrupt();
        tC.interrupt();
        System.out.println(x: "Thread interrotti");
    }
}
```

Eseguendo il codice si vedrà che il processore eseguirà “contemporaneamente” i tre thread assando più volte da un thread all’altro. Si noti che il tempo di esecuzione assegnato dal sistema operativo a ciascuno dei tre thread non è sempre lo stesso. Quando cambia la lettera scritta sul monitor significa che il sistema operativo ha eseguito un contest switching per assegnare al thread la risorsa di output.

ESERCIZIO SUI THREAD (lo scopo dell’esercizio è quello di mostrare l’utilità della programmazione multithreading per svolgere in maniera più veloce un’operazione suddividendola in thread).

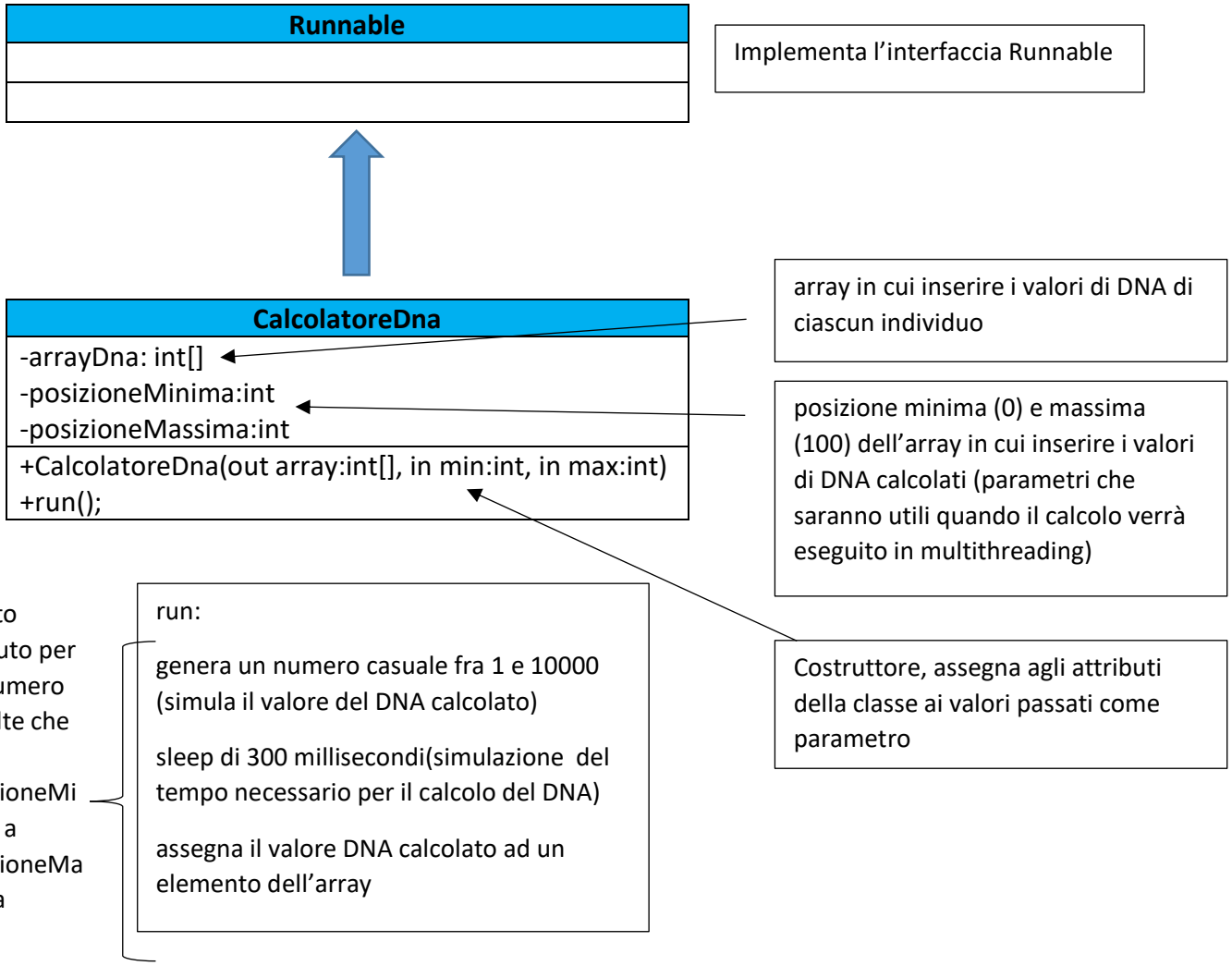
Versione 1

La polizia ha rinvenuto, sul luogo di un efferato omicidio, tracce del DNA del presunto colpevole. Si vuole pertanto determinare il DNA di tutti gli abitanti del villaggio di 100 abitanti per risalire all’identità del colpevole.

Per determinare il DNA di un individuo la procedura è complessa e richiede l’elaborazione dei dati biologici della persona. Per simulare il calcolo del DNA di un individuo si realizzi un metodo che genera un numero casuale compreso fra 1 e 10000 (che rappresenta il DNA) dopo un tempo di attesa di 300 ms (il tempo di attesa serve per “simulare” il tempo di elaborazione dei dati biologici per la determinazione del DNA di una persona).

L’attesa si ottiene con il metodo Thread.sleep(millisecondi) che “arresta” l’esecuzione di un thread per il numero di millisecondi indicato.

Si realizzi la seguente classe CalcolatoreDna che consente di calcolare il DNA di un certo numero di individui e memorizzare il valore in un array.



1. Si crei una MainClass con metodo main che istanzia un calcolatoreDna che calcola il DNA per tutti gli abitanti del villaggio e comunichi i valori calcolati (soluzione monoThread).
2. Si rilevi il tempo impiegato per l'elaborazione (Viene mostrato nella console di output).
3. Si modifichi la classe App istanziando 5 oggetti CalcolatoreDNA anziché uno solo, ciascuno dei quali esegue il calcolo per 20 persone, e iniziando 5 thread uno per ogni calcolatore (soluzione multiThread).
4. Si rilevi il miglioramento in termini di prestazioni rispetto alla soluzione monothread.

Soluzione

```
public class CalcolatoreDNA implements Runnable
{
    private int[] arrayDNA;
    private int posizioneMinima;
    private int posizioneMassima;

    public CalcolatoreDNA(int[] array, int min, int max)
    {
        this.arrayDNA=array;
        this.posizioneMinima=min;
        this.posizioneMassima=max;
    }
    @Override
    public void run()
    {
        Random r=new Random();
        int valoreDNA;
        for(int i=posizioneMinima;i<=posizioneMassima;i++)
        {
            valoreDNA=r.nextInt(bound: 1000);
            try
            {
                Thread.sleep(millis:300);
            }
            catch (InterruptedException ex)
            {
              //niente
            }
            arrayDNA[i]=valoreDNA;
        }
    }
}
```

Soluzione monothred (Dovrebbe impiegare circa 30 secondi)

```
public class App
{
    public static void main(String[] args) throws InterruptedException
    {
       //Prima esecuzione : monothread

        int[] arrayDNA=new int[100];
        int min=0;
        int max=99;
        CalcolatoreDNA cUnico=new CalcolatoreDNA(array: arrayDNA, min, max);
        cUnico.run();
        int i=1;
        for(int valore:arrayDNA)
        {
            System.out.println(i+"-->\t"+valore);
            i++;
        }

    }
}
```

Soluzione multithred (Dovrebbe impiegare circa 8 secondi)

```
public class App
{
    public static void main(String[] args) throws InterruptedException
    {
       //Seconda esecuzione : multithread

        int[] arrayDNA=new int[100];
        int min=0;
        int max=19;
        CalcolatoreDNA c1=new CalcolatoreDNA(array:arrayDNA, min, max);
        min=20;
        max=39;
        CalcolatoreDNA c2=new CalcolatoreDNA(array:arrayDNA, min, max);
        min=40;
        max=59;
        CalcolatoreDNA c3=new CalcolatoreDNA(array:arrayDNA, min, max);
        min=60;
        max=79;
        CalcolatoreDNA c4=new CalcolatoreDNA(array:arrayDNA, min, max);
        min=80;
        max=99;
        CalcolatoreDNA c5=new CalcolatoreDNA(array:arrayDNA, min, max);

        Thread t1 =new Thread(task: c1);
        Thread t2 =new Thread(task: c2);
        Thread t3 =new Thread(task: c3);
        Thread t4 =new Thread(task: c4);
        Thread t5 =new Thread(task: c5);


        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

        t1.join();
        t2.join();
        t3.join();
        t4.join();
        t5.join();

        int i=1;
        for(int valore:arrayDNA)
        {
            System.out.println(i+"-->\t"+valore);
            i++;
        }
    }
}
```

Occhio! ricordati!

I PROBLEMI DELLA PROGRAMMAZIONE CONCORRENTE

La programmazione con più thread viene detta **programmazione concorrente**, perché i thread "sono concorrenti" fra di loro nell'utilizzo delle risorse condivise. In questa tipologia di programmazione sono presenti particolari problemi di sincronizzazione fra i threade e il programmatore deve tener conto e risolvere tali problemi di sincronizzazione. Presentiamo ora i due problemi principali e mostriamo come vanno risolti, tali problemi sono chiamati **problema dell'interferenza fra thread** e **problema del produttore e consumatore**.

PROBLEMA 1. L' interferenza fra due thread (problema della **condivisione** di risorse fra due thread)

Ipotizziamo di avere due thread (thread A e thread B) che condividono una stessa variabile chiamata x con valore iniziale =10 (la risorsa condivisa), ossia entrambi vi possono accedere sia in lettura sia in scrittura.

Supponiamo che il codice del thread A ad un certo punto deve incrementare x di 1 e il codice del thread B deve decrementare x di 1. I due codici sono in esecuzione in parallela (per semplicità consideriamo un solo processore, non cambia niente se ne considerassimo di più) Per evidenziare il problema ipotizziamo che non ci sia nessun meccanismo di sincronizzazione fra i due thread.

Al termine dell'esecuzione di entrambi thread il valore di x dovrebbe essere 10 poiché la variabile è stata incrementata da A e decrementata da B. Vediamo però che può succedere qualcosa di diverso.

Per evidenziare il problema dobbiamo ricordare che ad ogni istruzione ad alto livello corrispondono sempre "più di una" operazione nel linguaggio macchina, quindi le due istruzioni di incremento e decremento potrebbero essere rappresentate in assembly come segue:

Thread A		Thread B	
alto livello	basso livello	alto livello	basso livello
….. x++ ……	LOD x ADD 1 STO X	….. x-- ……	LOD x SUB 1 STO X

Supponiamo che:

- il processore inizi ad eseguire le prime due istruzioni a livello macchina del thread A:

LOD X

Valore nella locazione di memoria X	Valore nel registro del processore
10	10

ADD 1

Valore nella locazione di memoria X	Valore nel registro del processore
10	11

- Prima di eseguire la terza istruzione del thread A avviene il context switching e il processore passa ad eseguire le istruzioni del thread B (eseguendo, questa volta, tutte e tre le istruzioni in linguaggio macchina):

  LOD X

Valore nella locazione di memoria X	Valore nel registro del processore
10	10

  SUB 1

Valore nella locazione di memoria X	Valore nel registro del processore
10	9

  STO X

Valore nella locazione di memoria X	Valore nel registro del processore
9	9

- Dopo aver eseguito le tre istruzioni di B, il processore torna ad eseguire la terza istruzione (a livello macchina) di A.
  Si ricorda che il contxt switching ripristina le condizioni del processore “così come era” quando è stata interrotta l’esecuzione del processo

STO X

Valore nella locazione di memoria X	Valore nel registro del processore
11	11

Il risultato finale è che al termine dell'esecuzione dei due thread la variabile condivisa x vale 11 anziché 10.

Questo è ciò che potrebbe accadere **in assenza di sincronizzazione** fra i thread.

Questo problema, in cui l'esecuzione di un thread causa una alterazione del risultato prodotto da un altro thread è chiamato problema dell'**interferenza** fra thread.

Si immagini che valore potrebbe risultare in "x" se i due thread non si limitassero ad incrementare e decrementare x ma svolgessero operazioni più complesse, ad esempio un ciclo che incrementa di 100 volte "x" per il thread A e un ciclo che decrementa 100 volte "x" per il thread B. Il valore di x potrebbe essere molto diverso da quello atteso al termine dell'esecuzione dei due thread in modalità parallela.

Un caso reale potrebbe essere quello in cui l'applicazione gestisce un conto corrente e il thread A gestisce le operazioni di accredito (thread A accredita 1000 euro) e il thread B le operazioni di pagamento (thread B toglie 1000 euro dal conto). Ci si potrebbe ritrovare alcune volte con 1000 euro in più, alcune volte con 1000 euro in meno, alcune volte con il conto corrente corretto!

Ma a cosa è dovuto il problema dell'interferenza? Il problema è sostanzialmente dovuto alla **non atomicità** delle istruzioni in cui si opera su variabili condivise. Non atomicità significa che le istruzioni in linguaggio ad alto livello sono costituite da **più** istruzioni in linguaggio macchina.

Le parti di codice di un thread in cui **sono coinvolte risorse condivise** con altri thread (ad esempio la variabile x) sono dette **sezioni critiche** del codice.

Il problema della mancata sincronizzazione si può ricondurre quindi alla **non atomicità delle sezioni critiche.**

OSSERVAZIONE: il problema del testing. Poiché le problematiche dovute alla programmazione concorrente non si verificano in ogni esecuzione del software, non è possibile effettuare la loro individuazione attraverso il testing. Infatti è possibile che, con gli stessi dati, in alcuni casi si verifichi il problema e in alcuni casi no. Per questo motivo l'unico modo per evitare i problemi dovuti alla programmazione concorrente

è quello di programmare in maniera particolarmente accurata e attenta implementando i meccanismi della programmazione concorrente.

SOLUZIONE AL PROBLEMA 1

Per risolvere il problema è necessario che le sezioni critiche di un thread non vengano "interrotte" a livello macchina **da sezioni critiche di altri thread che condividono le stesse risorse.**

La programmazione concorrente prevede l'utilizzo di opportuni algoritmi per risolvere questo problema.

Tali algoritmi devono soddisfare i seguenti 3 requisiti:

1. **Mutua esclusione**: una risorsa condivisa deve essere accessibile **in maniera esclusiva** da un solo thread. La stessa cosa si può dire anche così: in un certo istante, solo per uno dei due thread deve essere in esecuzione la sezione critica.

2. **Progresso**: un thread che sta eseguendo istruzioni "lontane" dalla sezione critica non deve ostacolare l'ingresso nella sezione critica da parte di un altro thread. (ossia un thread puo' essere "fermato" prima di accedere alla zona critica ma la decisione di quando "liberare" il suo accesso alla zona critica non deve dipendere da un altro thread lontano dalla zona critica)

3. **Attesa limitata**: un processo può essere "fermato" prima di accedere alla propria sezione critica ma tale attesa non può essere illimitata, prima o poi tutti i processi devono poter accedere alla risorsa. (Altrimenti la soluzione per l'accesso alle risorse condivise sarebbe molto semplice: arresto un processo ed eseguo solamente l'altro…..ma non avrei più il parallelismo).

Vediamo due soluzioni algoritmiche che sembra vadano bene ma in realtà non vanno bene, e poi una terza soluzione che va bene. (Le soluzioni sono presentate per 2 thread ma possono essere estese per un numero generico di thread)

Soluzione 1: semaforo binario

Utilizzo del meccanismo del **semaforo binario**, chiamato anche **mutex** (ideato dall'informatico olandese Edsger Dijkstra nel 1968). Il semaforo binario è una variabile binaria (può valere solo 1 o 0) **condivisa fra due thread**.

Il **protocollo di accesso e uscita alla SC (Sezione Critica)**, ossia le regole per l'accesso alla SC e per l'uscita dalla SC, è il seguente (per facilità diamo alla variabile semaforo binarioil nome "turno"):

- Il thread A può accedere alla propria sezione critica solamente se il turno vale 0
- il thread B può accedere alla propria sezione critica solamente se il turno vale 1
- Ogni thread, dopo aver eseguito la propria sezione critica, cambia il valore al turno (passa il turno al thread concorrente)

Inizialmente turno=0

Thread A	Thread B

….sezione non critica…
 	
while (turno==1)
{
no op (ciclo di attesa)
}
sezione critica
 	
turno = 1
 	
….sezione non critica…
	….sezione non critica…
	
	while (turno==0)
	{
	no op (ciclo di attesa)
	}
	sezione critica
	
	turno = 0
	
	….sezione non critica…

perché la soluzione non va bene?

- Mutua esclusione: ok
- Progresso, **non soddisfatto** perché ad esempio:

  supponiamo che A sia lontano dalla SC (sezione critica)

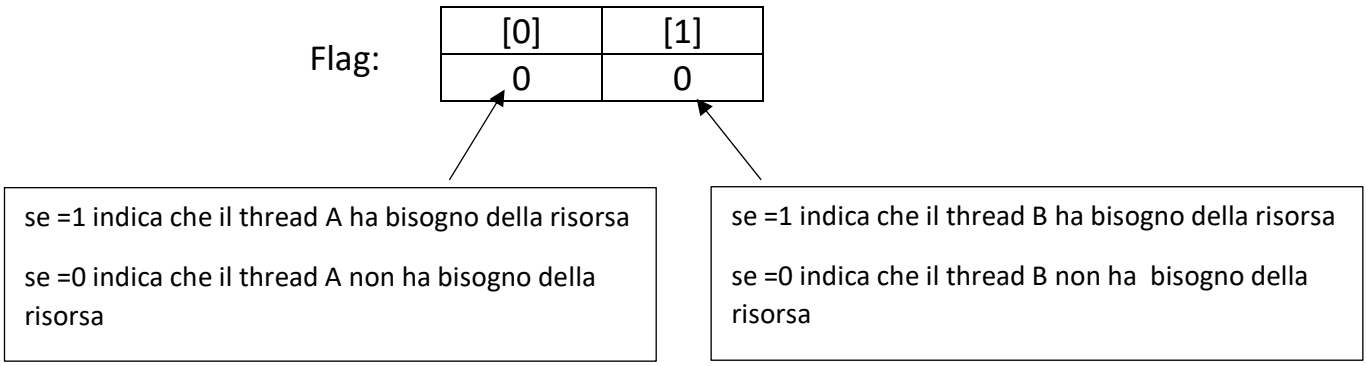
  B esce dalla SC e quindi pone turno =0

B torna alla propria SC (prima che A arrivi alla propria SC), ma non può entrare nella SC perché deve aspettare che vi entri A e chi poi esca ponendo turno=1.

- Attesa limitata: si (prima o poi tutti avranno accesso alla risorsa)

Soluzione 2: flag

Si utilizza un flag, ossia una array di 2 elementi condiviso fra i due Thread che indica la necessità di avere la risorsa (ossia di entrare nella SC)



Inizialmente entrambi i valori del flag sono posti a 0.

Protocollo di accesso e uscita alla SC (vale sia per A che per B):

- il thread pone a 1 il proprio flag (richiesta di accesso alla SC)
- Il thread accede alla SC se il flag del thread concorrente è=0 (quindi ,se il flag del concorrente è =1, il thread attende…..)
- il thread pone a 0 il proprio flag (libera la risorsa) dopo aver eseguito la propria SC

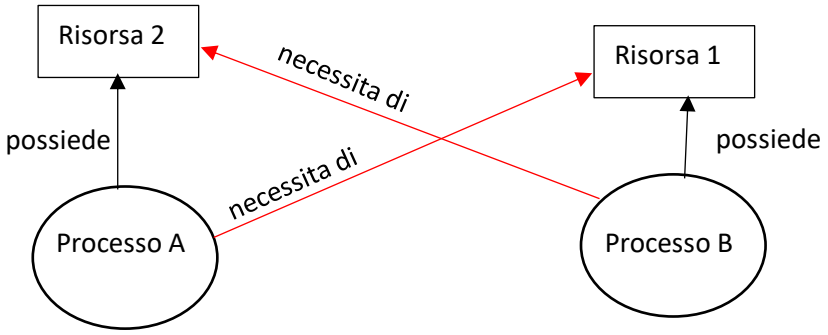
Thread A	Thread B

….sezione non critica…
 	
flag [0]=1
while (flag[1]==1) (richiesta da parte di B)
{
no op (ciclo di attesa)
}
sezione critica
 	
flag [0]=0
 	
….sezione non critica…
	….sezione non critica…
	
	flag [1]=1
	while (flag[0]==1) (richiesta da parte di A)
	{
	no op (ciclo di attesa)
	}
	sezione critica
	
	flag [1]=0
	
	….sezione non critica…

Perché non va bene:

- Mutua esclusione ok
  (quando un thread ha il proprio flag a 1, l'altro thread non può entrare nella SC)
- Progresso: ok
- Attesa limitata no perchè se entrambi pongono a 1 il proprio flag (non essendo ancora nella SC, dopo che A ha posto a 1 il proprio flag, potrebbe avvenire il context switching e B potrebbe porre a 1 il proprio flag) nessuno può entrare nella propria SC.

  Questa situazione è chiamata **Deadlock.**
  In generale la **Deadlock** si verifica quando un processo A necessita, per proseguire l'esecuzione, di una generica risorsa R1 che è posseduta in modo esclusivo da un processo B il quale necessita, per proseguire nell'esecuzione, di una generica risorsa R2 che è posseduta in modo esclusivo dal thread A (o processo A).
  Questa situazione è estremamente pericolosa perché impedisce a ciascun processo di proseguire con l'esecuzione.



Deadlock

(A volte il termine **Deadlock** viene confuso con il termine **Starvation** che invece indica la situazione in cui **un solo** processo non accede mai ad una risorsa condivisa, ad esempio al processore, generalmente perché ha una priorità troppo bassa).

Soluzione 3: Algoritmo di Peterson

Unisce le due soluzioni precedenti, ossia il semaforo binario (turno) e il flag.

Protocollo di accesso e uscita:

- il thread pone a 1 il proprio flag (richiesta di accesso)
- Il thread imposta il turno in favore del concorrente (cede il turno) (thread A imposta il turno a 1, thread B imposta il turno a 0)
- Il thread non accede alla SC fintanto che il flag del concorrente è a 1 e il turno è a favore del concorrente.
- Dopo essere entrato nella SC ed averla eseguita, il thread pone a 0 il proprio flag (libera la risorsa)

aspetto fintanto che l'altro thread ha bisogno della risorsa e ha il turno a favore

Thread A	Thread B

….sezione non critica…
 	
flag [0]=1
turno=1
while (flag[1]==1 and turno==1)
{
no op
}
sezione critica
 	
flag [0]=0
 	
….sezione non critica…
	….sezione non critica…
	
	flag [1]=1
	turno=0
	while (flag[0]==1 and turno==0)
	{
	no op
	}
	sezione critica
	
	flag [1]=0
	
	….sezione non critica…

Esercizio molto utile da fare: provare a considerare le situazioni che rendevano inefficaci le soluzioni 1 e 2 e verificare che esse vengono superate con questo algoritmo di Petersen (La mutua esclusione e il progresso sono sempre garantiti e la deadlock è evitata).

La parola chiave synchronized per gestire la concorrenza in java

Come è stato detto, il problema dell'interferenza fra thread è dato dal fatto che le istruzioni ad alto livello non sono atomiche a basso livello. Java mette a disposizione uno strumento per rende atomico il codice. La parola chiave **synchronized** consente di rendere atomiche le istruzioni di un metodo (oppure anche solo un blocco di codice) quando sono eseguite da un thread.

La parola chiave synchronize può riferirsi ad un oggetto, ad un metodo di un oggetto o semplicemente ad un blocco di codice, questo fa sì che UN SOLO THREAD alla volta potrà eseguire quella parte di codice oppure modificare quell'oggetto. Ciò rende le istruzioni con synchronize atomiche poiché quando eseguite da un thread A, altri thread non possono eseguirle (sullo stesso oggetto) finchè il thread A non ha terminato quel blocco di istruzioni.

Synchronized garantisce sempre la mutua esclusione ma non il progresso e l'attesa limitata. Nei casi più semplici di sincronizzazione fra processi, la parola synchronize è molto utile. In casi più complessi servono ulteriori strumenti a loro volta più complessi che non vedremo (ReentrantLook, Semaphore, CyclicBarrier, Exchanger).

Esempi di sintassi di synchronized:

Metodi sincronizzati: puoi dichiarare un intero metodo come synchronized. In questo caso, il lock (blocco) viene acquisito sull'oggetto stesso, ciò significa che ogni thread che condivide quell'oggetto potrà eseguire quel metodo solo in maniera esclusiva.

```
public synchronized void metodoSincronizzato()
{
    // codice critico
}
```

Attenzione: se in una classe vi sono più metodi synchronized, quando un thread A inizia l'esecuzione di uno di questi metodi, il thread B non potrà eseguire SULLO STESSO OGGETTO alcun metodo synchronized (**anche se il metodo è un metodo diverso da quello eseguito da A**). Invece i due Thred potranno agire contemporaneamente su metodi di oggetti diversi.

Blocchi sincronizzati: puoi utilizzare synchronized per sincronizzare un blocco di codice all'interno di un metodo.

```
public void metodo()
{
synchronized (this)
```

```
        {
        // codice critico
        }
}
```

Oggetti sincronizzati: solo un thread alla volta può accedere all'oggetto:

Esempio:

```
private int[] bufferLock = new int[10];
public void metodo()
{
        synchronized (bufferLock)
        {
                // codice critico
                for(int i=0;i<10;i++)
                    bufferLock[i]=10;          //solo un thread alla volta potrà scrivere in bufferLock
        }
}
```

Esempio di interferenza fra thread in Java.

Si realizzi una classe ContoCorrente che rappresenta un conto corrente bancario. Si realizzi una classe Operazione che rappresenta un'operazione (prelievo o versamento) che verrà svolta su un oggetto contoCorrente in modalità multiThread.

Per rendere evidente la problematica si rallenti uno dei due metodi (ad esempio "prelievo") con uno sleep di 100 ms

ContoCorrente
-numeroConto:String -saldo: double
+ContoCorrente(String numeroConto, double saldoIniziale) +prelievo(double valore) +versamento(double valore) -setSaldo(double saldo) +getSaldo():double +toString(): String

```
public class ContoCorrente
{
   private String numeroConto;
   private double saldo;

   public ContoCorrente(String numeroConto, double saldoIniziale)
   {
      this.numeroConto = numeroConto;
      this.saldo = saldoIniziale;
   }

   public void prelievo(double valore)
   {
      double totale=getSaldo();
     try
      {
         Thread.sleep(100);
      }
    catch (InterruptedException ex)
      {
         System.out.println("Interruzione thread");
      }
      totale=totale-valore;
      setSaldo(totale);
   }

   public void versamento(double valore)
   {
```

Rallento l'operazione di Prelievo per rendere più evidente il problema dell'interferenza

```
    double totale=getSaldo();
    totale=totale+valore;
    setSaldo(totale);
  }

  public void setSaldo(double saldo)
  {
    this.saldo = saldo;
  }



  public double getSaldo() {
    return saldo;
  }

  @Override
  public String toString() {
    return "ContoCorrente{" + "numeroConto=" + numeroConto + ", saldo=" + saldo + '}';
  }

}
```

Si realizzi ora una classe Operazione che consente di svolgere un'operazione (prelievo o versamento) su un oggetto di classe ContoCorrente. La classe Operazione verrà eseguita come thread:

```
public class Operazione implements Runnable
{
  ContoCorrente contoCorrente;
  char tipoOperazione;   //p=prelievo, v=versamento
  double importo;

  public Operazione(ContoCorrente c, char tipo, double importo)
  {
    contoCorrente=c;
    tipoOperazione=tipo;
    this.importo=importo;
  }

  @Override
  public void run()
  {
    if (tipoOperazione=='p' || tipoOperazione=='P')
      contoCorrente.prelievo(importo);
    else
      contoCorrente.versamento(importo);
  }
}
```

La seguente classe APP consente di mostrare che alcune operazioni possono "andare perse" a causa dell'interferenza fra thread.

```
public class App
{
   public static void main(String[] args) throws InterruptedException
   {
      ContoCorrente c1=new ContoCorrente("123", 1000);

      Operazione p;
      Operazione v;
      Thread prelievo = null;
      Thread versamento = null;

      ArrayList<Thread> elencoThread=new ArrayList();

      for(int i=0;i<10;i++)
      {
          p=new Operazione(c1, 'p', 100);
          prelievo=new Thread(p);

         v=new Operazione(c1, 'v', 200);
         versamento=new Thread(v);

         prelievo.start();
         versamento.start();
         elencoThread.add(prelievo);
         elencoThread.add(versamento);
      }

      //attendo che tutti i thread siano terminati
      for(Thread t:elencoThread)
      {
         t.join();
      }
      System.out.println("Conto c1: "+c1.toString());
   }
}
```

istanzio un Conto corrente

Utilizzo un array List in cui "mettere" ogni Thread istanziato. Questo mi servirà quando dovrò "attendere" la fine di tutti i Thread invocando il metodo Join() su ciascun Thread

Con un ciclo for istanzio e avvio 10 thread che "prelevano 100 €" e 10 thread che "versano 10 €"

Ogni Thread viene aggiunto alla lista dei Thread

Attendo la conclusione dei Thread prima di mostrare il saldo finale del conto corrente

Eseguendo il codice potrebbe accadere che il saldo finale si diverso da quello atteso (quello atteso è 2000 €).

Per risolvere la problematica si rendono atomici con synchronized i metodi "prelievo" e "versamento" nella classe ContoCorrente.

```
   public synchronized void prelievo(double valore)
   {
      double totale=getSaldo();
```

```
   try
    {
       Thread.sleep(100);
    }
  catch (InterruptedException ex)
    {
       System.out.println("Interruzione thread");
    }
    totale=totale-valore;
    setSaldo(totale);
}

public synchronized void versamento(double valore)
{
    double totale=getSaldo();
    totale=totale+valore;
    setSaldo(totale);
}
```

SVOLGERE L'ESERCIZIO "PRENOTAZIONI TEATRO"

PROBLEMA 2. La sincronizzazione fra due thread. (Il problema del produttore e consumatore)

Uno dei classici problemi di sincronizzazione fra processi (o thread) che si risolvono mediante le tecniche della programmazione concorrente è il problema del produttore e consumatore.

Situazione:

Un processo chiamato produttore produce **ciclicamente** dei dati e li memorizza in una zona di memoria condivisa chiamata buffer. Un altro processo chiamato processo consumatore legge i dati **ciclicamente** dal buffer condiviso e li elabora. Inizialmente ipotizziamo che il buffer sia costituito da una sola cella di memoria su cui si possa quindi scrivere/leggere un solo dato alla volta.

(Esempi di processi produttore e consumatore sono i seguenti: un router wireless che riceve dati da una rete cablata e li deve inoltrare su una rete wireless. Una applicazione che riceve pacchetti di dati dalla rete e deve elaborarli per visualizzare un filmato. Un wordprocessor che riceve i dati dalla tastiera e li deve mostrare su un monitor….)

Descrizione del problema:

1. Il buffer è una memoria condivisa fra i due processi pertanto i due processi devono accedervi in regime di **mutua esclusione** ossia non contemporaneamente (altrimenti leggendo i dati mentre il produttore scrive, i dati letti possono essere inconsistenti, non corretti)
2. Le operazioni di lettura e di scrittura vanno inoltre sincronizzate perché:
    a. dopo aver scritto un dato il produttore non deve scrivere finchè il consumatore ha letto il dato prodotto (per evitare di sovrascriverlo)
    b. dopo aver letto un dato il consumatore non deve leggere prima che il produttore abbia scritto un nuovo dato (per evitare di leggere due volte lo stesso dato).

Per risolvere il problema si utilizzano due semafori binari S1 e S2 che però ridefiniamo più precisamente rispetto alla definizione data precedentemente.

Un generico semaforo binario (S), detto anche **mutex**, nella definizione di Dijkstra non è semplicemente una variabile binaria. In realtà è un contatore su cui si possono svolgere due operazioni in maniera **atomica** (vedremo che in Java, per rendere atomiche delle istruzione si usa la parola chiave **syncronized**). Le operazioni sono le seguenti:

S.wait(): Il semaforo viene decrementato se il suo valore è >0, altrimenti l'esecuzione del processo in cui è presente il semaforo, viene sospeso finchè S diventa >0.

S.signal(): Il semaforo viene incrementato e il processo contenente il semaforo S, se "stava aspettando" che il semaforo diventasse >0, riprende l'esecuzione.

SOLUZIONE AL PROBLEMA 2

Utilizzando due semafori binari, inizializzati con S1=1 e S2=0, la sincronizzazione fra i due thread produttore e consumatore avviene implementando al loro interno i seguenti algoritmi

thread Produttore	thread Consumatore

while(true)
{
dato=produciDato();
 	
S1.wait();
 	
buffer=dato;
 	
S2.signal();
}
	while(true)
	{
	S2.wait();
	
	dato=buffer;
	
	S1.signal();
	
	elabora(dato);
	}

In questo modo il Produttore produce il primo dato e lo carica nel buffer, dopodichè (essendo S1=0) non scriverà più nel buffer finchè il consumatore avrà letto il dato dal buffer e, successivamente incrementato a S1=1 "riabilitando" il produttore a scrivere.

Allo stesso modo, finchè non è stato scritto il primo dato dal produttore il consumatore non legge dal buffer (essendo S2=0). Quando il Produttore avrà scritto nel buffer, S2 verrà incrementato a 1 e il Consumatore potrà leggerlo. A questo punto il Consumatore incrementa S1=1 abilitando nuovamente la scrittura da parte del produttore, e non potrà più leggere finchè il produttore non avrà scritto un nuovo dato.

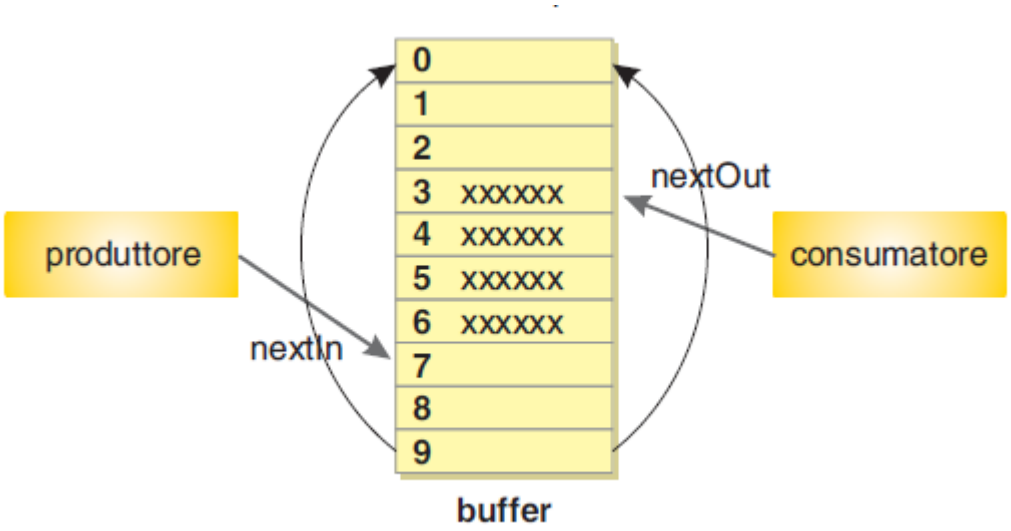
Produttore Consumatore con dimensione del buffer maggiore.

La soluzione precedente risolve il problema della sincronizzazione fra i thread produttore e consumatore, però nel caso in cui il produttore fosse molto veloce a produrre dati rispetto alla velocità del consumatore a leggerli, il produttore dovrebbe passare molto tempo inattivo perché non può scrivere nel buffer il nuovo dato che ha prodotto finchè il produttore non ha letto il dato precedente.

Il problema si risolve utilizzando un **buffer circolare** in grado di contenere più dati.

Un buffer circolare è un buffer dove le operazioni di scrittura da parte del Produttore e le operazioni di lettura da parte del Consumatore avvengono circolarmente, ossia la scrittura/lettura parte dalla posizione zero fino ad arrivare all'ultima posizione del buffer, dopodiché riparte dalla posizione zero.

Un esempio di un buffer circolare di dimensione 10 è il seguente:



Possiamo rappresentare il buffer come un array.

NextIn è l'indice della posizione nella quale verrà scritto il prossimo dato da parte del Produttore.

NextOut è l'indice della posizione del buffer dalla quale verrà letto il prossimo dato da parte del Consumatore.

Il Produttore inizierà a scrivere dalla posizione 0 del buffer e proseguirà fino alla posizione 9, poi ripartirà a scrivere dalla posizione 0 ma scrivendo solo nelle posizioni che sono già state lette dal Consumatore.

I codici che consentono la sincronizzazione dei due thread sono i seguenti. In questo caso il semaforo binario S1 viene inizializzato con un valore pari alla dimensione del buffer, e S2 a 0. Quindi nel nostro esempio

S1=10 e S2=0

In questo modo il Produttore potrà scrivere fino a 10 dati senza che il Consumatore legga alcun dato, prima che S1 arrivi a 0 "impedendo" la scrittura di altri dati.

Il codici eseguiti ciclicamente dai due thread sono i seguenti:

Si osservi che per "far ripartire" da 0 gli indicatori delle posizioni di scrittura si usa l'operatore % (mod) che restituisce il resto della divisione intera.

thread Produttore	thread Consumatore

nextInt=0;
 	
while(true)
{
dato=produciDato();
 	
S1.wait(); — Decrementa S1 finchè S1=0, a quel punto interrompe l'esecuzione del thread finchè S1 viene incrementato.
 	
buffer[nextIn]=dato;
 	
nextInt=(nextInt+1)%10;
 	
S2.signal(); — Incrementa S2
}
	nextOut=0
	
	while(true)
	{
	S2.wait(); — Decrementa S2 finchè S2=0, a quel punto interrompe l'esecuzione del thread finchè S2 viene incrementato.
	
	dato=buffer[nextOut]
	
	nextOut=(nextOut+1)%10
	
	S1.signal(); — Incrementa S2
	
	elabora(dato);
	}

Come abbiamo detto, i metodi signal() e wait() del semaforo devono essere resi atomici. Tale operazione è possibile premettendo la parola chiave **synchronized**() alla firma dei metodi che agiscono sulla memoria condivisa.

Ad esempio sui seguenti due metodi della classe Elenco:

```
synchronized public boolean metodoA( float parametro1, String parametro2)
synchronized public boolean metodoB (float parametro1, String parametro2)
```

Quando un thread1 invoca un metodo “synchronized” di una classe, un thread2 non può iniziare l’esecuzione di un metodo “synchronized” (sia che si tratti dello stesso metodo che di altri) fino a che il thread1 non ha terminato l’esecuzione del metodo. Syinchronized consente quindi di rendere atomica l’esecuzione di un metodo da parte di un thread.

Esercizio produttore consumatore con una cella di memoria condivisa:

Scriviamo il codice per realizzare i due thread Produttore e Consumatore che condividono una sola cella di memoria.

Il dato prodotto dal Produttore è un numero intero casuale compreso fra 1 e 1000, la zona di memoria condivisa è un array di un solo elemento (utilizzare una variabile int o Integer non va bene, come spiegato nel codice).

Simuliamo il consumo del dato da parte del Consumatore nel seguente modo:

1. un ritardo di 2 secondi
2. la lettura del dato
3. l’output del dato sulla console

SOLUZIONE:

Innanzitutto creiamo la classe Semaforo che rappresenta il semaforo di Dijkstra. Il metodo wait del semaforo è stato chiamato P(), il metodo Signal è stato chiamato V(), le due lettere utilizzate sono state utilizzate da Dijkstra e corrispondono alle iniziali delle due parole olandesi proberen (tentare, verificare) e verhogen(incrementare).

```
public class Semaforo
{
   private int valore;  //contatore che vale 1 o 0

   public Semaforo(int v)
   {
      valore =v;
   }

   //metodo atomico wait si chiama P()
   //dalla parola olandese proberen (tentativo)
   public synchronized void P() throws InterruptedException
   {
      while(valore==0)
      {
          wait();                          //metodo (della classe Object, quindi ereditato
                                            // da ogni classe) che ferma il thread
                                           //il thread verrà risvegliato dal metodo Java
                                           //notify() o notifyAll()
```

```
    }
    valore--;
  }

  //metodo atomico signal si chiama V()
  //dalla parola olandese verhogen (incrementare)
  public synchronized void V()
  {
    valore++;
    notify();              //metodo (ereditato della classe Object) che risveglia il thread
  }
}
```

Codice della classe Produttore, la produzione del dato consiste semplicemente nella generazione di un numero casuale fra 1 e 1000

```
public class Produttore implements  Runnable
{
    private int[] buffer;    //devo usare un array come memoria condivisa
                             // un int non va bene perchè il passaggio di parametri
                             //avviene sempre per valore in java
                             //un Integer non va bene poichè le classi wrapper sono imutabili
                             //quindi se ne modifichi il valore crea un'altra variabile (che non è più
                             //quella condivisa
    private Semaforo s1;
    private Semaforo s2;

//Nel costruttore passo come parametro il buffer condiviso e i due semafori

    public Produttore(int[] buffer,Semaforo s1,Semaforo s2 )
    {
      this.buffer=buffer;
      this.s1=s1;
      this.s2=s2;
    }

    public void run()
    {
      while(!Thread.interrupted())
      {
        try
        {
          s1.P();            //se c'è un dato non ancora letto dal consumatore...wait
        }
        catch (InterruptedException ex) //se il thread viene interrotto durante wait
        {                                //si verifica questa eccezione ma il thread non si
                                         //interrompe. Va interrotto con un break

          System.out.println("Produttore interrotto");
          break;
        }
        buffer[0]=produciDato();
        System.out.println("Dato prodotto -> "+buffer[0]);
        s2.V();              //attiva il semaforo s2 per "avvisare" Consumatore che
                             //c'è un dato pronto da leggere
      }
```

```
    }

    /*
    Genera un numero casuale
    */
    private int produciDato()
    {
       Random r=new Random();
       return 1+(r.nextInt(10000));
    }
}
```

Codice della classe Consumatore, la consumazione del dato consiste semplicemente nel raddoppiare il dato letto e mostrarlo dopo un'attesa di 2 secondi

```
public class Consumatore implements Runnable
{
   private int[] buffer;            //devo usare un array come memoria condivisa
                                    // un int non va bene perchè il passaggio di parametri
                                    //avviene sempre per valore in java
                                   //un Integer non va bene poichè le classi wrapper
                                   // sono imutabili
                                   //quindi se ne modifichi il valore crea
                                   //un'altra variabile (che non è più quella condivisa
  private Semaforo s1;
  private Semaforo s2;

   //Nel costruttore passo come parametro il buffer condiviso e i due semafori
  public Consumatore(int[] buffer,Semaforo s1,Semaforo s2 )
  {
     this.buffer=buffer;
     this.s1=s1;
     this.s2=s2;
  }

  public void run()
  {
     while(!Thread.interrupted())
     {
        //se non ci sono dati non ancora letti...wait
        s2.P();
        try
        {
           Thread.sleep(2000);
        }
        catch (InterruptedException ex) //se il thread viene interrotto durante sleep
                                        //si verifica questa eccezione ma il thread non si
                                        //interrompe. Va interrotto con un break
        {
           System.out.println("Consumatore interrotto ");
           break;
        }
        System.out.println("Dato elaborato -> "+2*buffer[0]); //lettura ed elaborazione
                                                              //del dato
        s1.V();                          //signal
     }
  }
}
```

Codice della classe App

```
public class App
{
   public static void main(String[] args)
   {
      int[] buffer=new int[1];                    //devo usare un array come memoria condivisa
                                                  // un int non va bene perchè il passaggio di parametri
                                                  //avviene sempre per valore in java
                                                 //un Integer non va bene poichè le classi wrapper
                                                 // sono imutabili
                                                 //quindi se ne modifichi il valore crea
                                                //un'altra variabile (che non è più quella condivisa

      Semaforo s1=new Semaforo (1);
      Semaforo s2=new Semaforo(0);

      Produttore p=new Produttore(buffer, s1,s2);
      Consumatore c=new Consumatore(buffer, s1,s2);

       Thread produttore=new Thread(p);
       Thread consumatore=new Thread(c);

      Scanner tastiera=new Scanner(System.in);
      System.out.println("Premi invio per interrompere");
      produttore.start();
      consumatore.start();

      tastiera.nextLine();
      produttore.interrupt();
      consumatore.interrupt();
   }
}
```

OUTPUT

```
Dato prodotto -> 6218
Dato elaborato -> 12436
Dato prodotto -> 3741
Dato elaborato -> 7482
Dato prodotto -> 1842
Dato elaborato -> 3684
Dato prodotto -> 7311
Dato elaborato -> 14622
Dato prodotto -> 653
Dato elaborato -> 1306
Dato prodotto -> 5495
```

Esercizio produttore consumatore con buffer circolare di dimensione 10:

La classe Semaforo è la stessa dell'esercizio precedente.

Il Produttore e il consumatore condividono un buffer circolare di dimensione 10.

Il produttore produce periodicamente un dato (numero casuale) che scriverà nel buffer circolare. Il tempo che passa fra la produzione di un dato e l'altro è un valore casuale fra 0 e 1000 millisecondi.

Il consumatore aspetta un tempo casuale (fra 0 e 1000 ms) prima di leggere dal buffer circolare. Dopo aver letto il dato, per "mostrare" che il dato è stato letto, il consumatore scrive un valore 0 nel buffer.

Ogni volta che un dato viene scritto o letto, viene mostrato nella console l'intero buffer circolare.

Codice della classe Produttore

```
public class Produttore implements  Runnable
{
    private int[] buffer;
    private Semaforo s1;
    private Semaforo s2;

    public Produttore(int[] buffer,Semaforo s1,Semaforo s2 )
    {
      this.buffer=buffer;
      this.s1=s1;
      this.s2=s2;
    }

    public void run()
    {
      int nextIn=0;
      while(!Thread.interrupted())
      {
        s1.P();      //se c'è un dato non ancora letto dal consumatore...wait
        try
        {
          Thread.sleep(numeroCasuale()); //attendo tempo casuale che simula
                                         // l' attesa per la produzione del dato
        }
        catch (InterruptedException ex)
        {
          System.out.println("Errore sleep "+ex.getMessage());
        }
        buffer[nextIn]=numeroCasuale();   //produce il dato
        mostraBuffer();                   //mostra il buffer
        nextIn=(nextIn+1)%buffer.length;
        s2.V();      //attiva il semaforo s2 per "avvisare" Consumatore che
                     //c'è un dato pronto da leggere
```

```
        }
    }

    /*
    Genera un numero casuale
    */
    private int numeroCasuale()
    {
       Random r=new Random();
       return (1+r.nextInt(1000));
    }

    private void mostraBuffer()
    {
       System.out.println();
       for (int valore:buffer)
       {
          System.out.print(valore+"\t");
       }
       System.out.println(" scritto");
    }
}
```

Codice della classe Consumatore

```
public class Consumatore implements Runnable
{
   private int[] buffer;
   private Semaforo s1;
   private Semaforo s2;

   public Consumatore(int[] buffer,Semaforo s1,Semaforo s2 )
   {
      this.buffer=buffer;
      this.s1=s1;
      this.s2=s2;
   }

   public void run()
   {
      int nextOut=0;
      while(!Thread.interrupted())
      {

         //se non ci sono dati non ancora letti...wait
         s2.P();
         try
         {
            Thread.sleep(numeroCasuale());  //attendo tempo casuale che simula l' attesa per
                                            //la consumazione del dato
         }
         catch (InterruptedException ex)
         {
            System.out.println("Sleep error: "+ex.getMessage());
         }
```

```
            buffer[nextOut]=0;                  //"simulo" la consumazione sostituendo il valore 0
                                                // nel buffer dato letto
            mostraBuffer();
            nextOut=(nextOut+1)%buffer.length;
            s1.V();
        }
    }

    private void mostraBuffer()
    {
        System.out.println();
        for (int valore:buffer)
        {
            System.out.print(valore+"\t");
        }
       System.out.println(" letto");
    }

    private int numeroCasuale()
    {
        Random r=new Random();
        return r.nextInt(1000);
    }
}
```

Codice della classe App

```
public class App
{

    public static void main(String[] args)
    {
        int[] buffer=new int[10]; //Il buffer è da 10 eleemnti

        Semaforo s1=new Semaforo (buffer.length);      //S1 è inizializzato alla stessa
                                                       //lunghezza del buffer
        Semaforo s2=new Semaforo(0);

        Produttore p=new Produttore(buffer, s1,s2);
        Consumatore c=new Consumatore(buffer, s1,s2);

        Thread tProd=new Thread(p);
        Thread tCons=new Thread(c);

        tProd.start();
        tCons.start();
//Aggiungere l'arresto dei thread

    }
}
```

OUTPUT:

253	0	0	0	0	0	0	0	0	0	scritto
253	765	0	0	0	0	0	0	0	0	scritto
0	765	0	0	0	0	0	0	0	0	letto
0	765	98	0	0	0	0	0	0	0	scritto
0	765	98	41	0	0	0	0	0	0	scritto
0	0	98	41	0	0	0	0	0	0	letto
0	0	98	41	455	0	0	0	0	0	scritto
0	0	0	41	455	0	0	0	0	0	letto
0	0	0	41	455	27	0	0	0	0	scritto
0	0	0	41	455	27	40	0	0	0	scritto
0	0	0	0	455	27	40	0	0	0	letto
0	0	0	0	0	27	40	0	0	0	letto
0	0	0	0	0	27	40	634	0	0	scritto
0	0	0	0	0	27	40	634	523	0	scritto
0	0	0	0	0	0	40	634	523	0	letto
0	0	0	0	0	0	40	634	523	83	scritto
0	0	0	0	0	0	0	634	523	83	letto
0	0	0	0	0	0	0	0	523	83	letto
538	0	0	0	0	0	0	0	523	83	scritto
538	0	0	0	0	0	0	0	0	83	letto
538	0	0	0	0	0	0	0	0	0	letto
538	83	0	0	0	0	0	0	0	0	scritto
538	83	795	0	0	0	0	0	0	0	scritto
538	83	795	129	0	0	0	0	0	0	scritto
0	83	795	129	0	0	0	0	0	0	letto
0	0	795	129	0	0	0	0	0	0	letto
0	0	795	129	567	0	0	0	0	0	scritto

Esercizio da fare:
Modificare le classi Produttore e consumatore in modo che il dato prodotto sia inserito manualmente dall'utente. Si consiglia di rallentare la lettura da parte del consumatore per apprezzare meglio la sincronizzazione fra i due processi