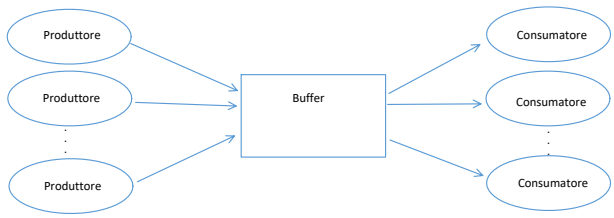
Produttori-Consumatori



Produttore:
- ciclicamente immette un nuovo valore nel buffer
- se il buffer è pieno attende che si liberi spazio

Consumatore:
- ciclicamente preleva un nuovo valore dal buffer
- se il buffer è vuoto attende che sia immesso un nuovo valore

Buffer:
- oggetto condiviso dai produttori e dai consumatori
- è in grado di contenere un numero limitato di valori

```
public class Buffer {

  private static final int DEFAULT_SIZE = 4;
  private String[] vett;
  private int testa, coda;
  private int quanti;

  public Buffer(){
    this(DEFAULT_SIZE);
  }

  public Buffer(int n) {
    vett = new String[n];
  }

  public synchronized void inserisci(String s) throws InterruptedException {
    while(pieno()) {
      System.out.println(Thread.currentThread().getName() + ": mi blocco, buffer pieno");
      wait();
    }
    vett[coda] = s;
    coda = (coda + 1) % vett.length;
    quanti++;
    notifyAll();
  }

  public synchronized String estrai() throws InterruptedException {
    while(vuoto()) {
      System.out.println(Thread.currentThread().getName() + ": mi blocco, buffer vuoto");
      wait();
    }
    String t = vett[testa];
    testa = (testa + 1) % vett.length;
    quanti--;
    notifyAll();
    return t;
  }

  public synchronized boolean pieno() {
    return quanti == vett.length;
  }

  public synchronized boolean vuoto() {
    return quanti == 0;
  }
 }

public class Produttore extends Thread {

 private Buffer b;

 public Produttore(String n, Buffer b) {
  super(n);
  this.b = b;
 }

 public void run(){
  int c = 0;
  try {
   while(true) {
    sleep((long)(Math.random()*1000));
    b.inserisci(getName() + ": " + c++);
    System.out.println(getName() + ": ho inserito");
   }
  } catch(InterruptedException ie) {
   return;
  }
 }
}


public class Consumatore extends Thread {

 private Buffer b;

 public Consumatore(String n, Buffer b) {
  super(n);
  this.b = b;
 }

 public void run(){
  try {
   while(true) {
    sleep((long)(Math.random()*1000));
```

```
      String s = b.estrai();
      System.out.println(getName() + ": ho estratto \"" + s + "\"");
    }
   } catch(InterruptedException ie) {
     return;
   }
  }
}

import java.util.Scanner;

public class Prova {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Quanti produttori? ");
    int p = sc.nextInt();
    System.out.println("Quanti consumatori? ");
    int c = sc.nextInt();
    System.out.println("Dimensione del buffer? ");
    int b = sc.nextInt();

    Buffer buf = new Buffer(b);
    for(int i=0; i<p; i++)
      new Produttore("Produttore " + i, buf).start();
    for(int i=0; i<c; i++)
      new Consumatore("Consumatore " + i, buf).start();
  }
}
```

Supponiamo di sostituire le *notifyAll()* con delle *notify()*.

Esempio di sequenza di operazioni che porta a blocco (deadlock): 2 produttori, 1 consumatore, buffer di dimensione 1.

Coda pronti: P0, P1, C0
Bloccati: nessuno
- P0 inserisce

Coda pronti: P0, P1, C0
Bloccati: nessuno
- P1 prova a inserire e si blocca per buffer pieno

Coda pronti: P0, C0
Bloccati: P1
- P0 prova a inserire e si blocca per buffer pieno

Coda pronti: C0
Bloccati: P0, P1
- C0 estrae e risveglia P0 (a caso)

Coda pronti: C0, P0
Bloccati: P1
- C0 prova a estrarre e si blocca per buffer vuoto

*Coda pronti: P0
*Bloccati: C0, P1
*- P0 inserisce e risveglia P1 (a caso)

Coda pronti: P0, P1
Bloccati: C0
- P1 prova a inserire e si blocca per buffer pieno

Coda pronti: P0
Bloccati: C0, P1
- P0 prova a inserire e si blocca per buffer pieno

Coda pronti:
Bloccati: C0, P0, P1

BLOCCO

Se avessi avuto notifyAll(): nellelemento indicato dagli asterischi avrei riportato tra i pronti anche il consumatore impedendo il blocco.

Di *wait()* ne esistono più versioni:

public final void wait(long timeout) throws InterruptedException

- Il thread corrente si blocca fino a quando non si verifica una delle seguenti cose:
  - *notify()* è invocata su questo oggetto e il thread è selezionato
  - *notifyAll()* è invocata su questo oggetto
  - scade il timeout specificato (in millisecondi)
  - viene invocato il metodo *interrupt()* sul thread
- Se timeout è zero il timer non scatta mai
- Durante la *wait()* il lock sull'oggetto viene rilasciato ed è automaticamente riacquisito prima che la *wait()* sia completata (indipendentemente dal motivo per cui si esce dalla *wait()*)
- *InterruptedException* viene lanciata se il thread è bloccato su una *wait()* e viene interrotto.

public final void wait(long timeout, int nanos) throws InterruptedException

- Come sopra, ma il timeout ha anche la componente nanosecondi

public final void wait() throws InterruptedException

- Equivalente a wait(0)

Il timeout può essere usato come meccanismo di programmazione difensiva, per esempio per tollerare il fallimento del thread che esegue notify().

La wait() deve sempre essere invocata testando la condizione di blocco in un ciclo e non in un if:

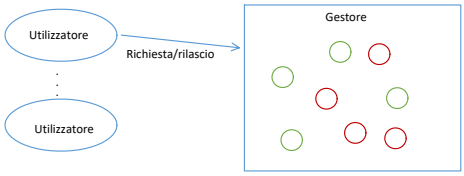
```
while(condizione_di_blocco)          OK
  wait()

if(condizione_di_blocco)             NO
  wait()
```

Se c'è un solo thread che viene notificato e la condizione per proseguire è adesso vera può sembrare logico usare un if. Il problema è che le specifiche della JVM prevedono la possibilità di "spurious wakeup": il thread viene svegliato senza che ci siano *notify()* (è un evento raro).

Gestore di risorse equivalenti

Un gestore dispone di *n* risporse equivalenti.

Gli utilizzatori chiedono una risporsa al gestore quando ne hanno bisogno e la restituiscono quando hanno finito.



Il comportamento del gestore può essere specificato dai seguenti due metodi:

- *int richiesta()*: restituisce al chiamante l'indice della risorsa assegnata; se non ci sono risorse disponibili il chiamante si blocca.
- *void rilascio(int k)*: rilascia la risorsa indicata da *k*.

```
public class Gestore {

  private boolean[] risOccupate;
  private int numOccupate;

  public Gestore(int n) {
    risOccupate = new boolean[n];
  }

  public synchronized int richiesta() throws InterruptedException {
    while(numOccupate == risOccupate.length)
      wait();
    int i=0;
    for(; i<risOccupate.length && risOccupate[i]; i++);
    risOccupate[i] = true;
    numOccupate++;
    return i;
  }

  public synchronized void rilascio(int k) {
    risOccupate[k] = false;
    numOccupate--;
    notify();
  }
}
```

```
public class Utilizzatore extends Thread {

  private Gestore g;
  private int numRichieste;
  private static final int DEFAULT_NUM_RICHIESTE = 100;

  public Utilizzatore(String n, Gestore g, int r) {
    super(n);
    this.g = g;
    this.numRichieste = r;
  }

  public Utilizzatore(String n, Gestore g) {
    this(n, g, DEFAULT_NUM_RICHIESTE);
  }

  public void run(){
    try {
      for(int i=0; i<numRichieste; i++) {
        int k = g.richiesta();
        System.out.println(getName() + ": ho ottenuto la risorsa " + k);
        sleep((long)(Math.random()*1000));
        g.rilascio(k);
        System.out.println(getName() + ": ho rilasciato la risorsa " + k);
      }
    } catch(InterruptedException ie) {
      return;
    }
  }
}
```

```
import java.util.Scanner;
public class Prova {
```

```
public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  System.out.println("Quanti utilizzatori?");
  int u = sc.nextInt();
  System.out.println("Quante risorse?");
  int r = sc.nextInt();
  Gestore g = new Gestore(r);
  for(int i=0; i<u; i++)
    new Utilizzatore("U"+i, g).start();
 }
}
```

Interrupt

- E' possibile inviare un interrupt a un thread, in genere per chiedergli di interrompere le sue operazioni.
- Per inviare un interrupt a un thread si usa il metodo

  public void interrupt()

  sull'oggetto thread da interrompere.
- Ad ogni thread è associato un interrupt status.
- Quando inviamo un interrupt a un thread:
  - se il thread è bloccato su
    - wait
    - sleep
    - join

    l'interrupt status non viene settato e il thread esce da wait, sleep, join con una *InterruptedException*
  - altrimenti viene settato l'interrupt status.
- Un thread può controllare il suo interrupt status con i seguenti metodi (ereditati dalla classe *Thread*):
  - *public static boolean interrupted()*: restituisce *true* se l'interrupt status è settato, *false* se non è settato. In ogni caso resetta l'interrupt status (se lo chiamo due volte di fila la seconda restituisce sicuramente *false*).
  - *public boolean isInterrupted()*: restituisce *true* se l'interrupt status è settato, *false* altrimenti. Non resetta lo status.

Esempio:

```
public class Creatore {

  public static void main(String[] args){
    Esecutore e = new Esecutore();
    e.start();
    try{
      Thread.sleep(2000);
    } catch(InterruptedException ie) {
      // do nothing
    }
    e.interrupt();
  }
}
```

```
public class Esecutore extends Thread {
  public void run(){
    while(!interrupted()) {
      fai();
    }
  }

  private void fai(){
    System.out.print(".");
  }
}
```

```
public class Creatore2 {

  public static void main(String[] args){
    Esecutore2 e = new Esecutore2();
    e.start();
    try{
      Thread.sleep(2000);
    } catch(InterruptedException ie) {
      // do nothing
    }
    e.interrupt();
  }
}
```

```
public class Esecutore2 extends Thread {
  public void run(){
    try {
      while(!interrupted()) {
        fai();
        Thread.sleep(100);
      }
    } catch (InterruptedException ie) {
        //Exit
        System.out.println("Sono stato interrotto");
    }
  }

  private void fai(){
    System.out.print(".");
  }
}
```

potrebbe essere un'altra operazione bloccante esce con eccezione

Dopo due secondi *Esecutore* viene interrotto e termina il proprio lavoro.

NOTA: il thread che viene interrotto deve cooperare controllando se ha ricevuto interruzioni e nel caso terminare le proprie operazioni.

Join

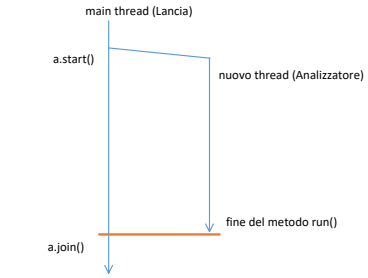
Un thread può attendere la terminazione di un altro thread mediante il metodo *join()*.

Esempio:

```
public class Lancia {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    Analizzatore a = new Analizzatore(n);
    a.start();
    try {
```

```
      a.join();
      boolean r = a.getRisultato();
      System.out.println(r ? "primo" : "non primo");
    } catch (InterruptedException ie) {
      //nothing
    }
  }
}
```

```
public class Analizzatore extends Thread {

  private int n;
  private boolean risultato;

  public Analizzatore(int n) {
    this.n = n;
  }

  public void run(){
    risultato = primo(n);
  }

  public boolean getRisultato(){
    return risultato;
  }

  private boolean primo(int n){
    for(int i=2; i<n; i++) {
      if(n%i == 0)
        return false;
    }
    return true;
  }
}
```



Ci sono tre versioni di join:

public final void join(long millis) throws InterruptedException
attende la terminazione del thread su cui il metodo è invocato o che siano trascorsi millis millisecondi (il primo dei due eventi). Se millis vale zero il timeout non scatta mai. Se il thread bloccato sulla join viene interrotto viene lanciata InterruptedException.

public final void join(long millis, int nanos) throws InterruptedException
come sopra ma con la possibilità di specificare anche i nanosecondi.

public final void join() throws InterruptedException
equivale a join(0)

Thread e applicazione

Fino adesso abbiamo detto che ogni applicazione Java parte con un thread che esegue il metodo main
- Se non vengono creati altri thread l'applicazione termina quando termina l'esecuzione del main.

In effetti in Java ci sono due tipi di thread:
- *user* thread
- *daemon* thread

Una applicazione termina quando termina l'ultimo user thread.
- i daemon thread non mantengono in vita l'applicazione
- quando termina l'ultimo user thread se ci sono dei daemon thread ancora attivi questi vengono terminati

I thread creati da uno user thread sono user thread, quelli creati da un daemon thread sono daemon thread.

Il primo thread, quello del main, è uno user thread.

Per creare un daemon thread usare
- *setDaemon(true)*
- dopo creazione, ma prima di *start*

E' possibile forzare la terminazione di tutti i thread e dell'applicazione mediante il metodo

System.exit(int status)

ESERCIZIO

Lino ha appena trovato un nuovo lavoro (vedi video
https://www.youtube.com/watch?v=cFCLln7HGpU).
Realizzare le classi Lino, DottTomas, Comando, Console e Prova secondo
le seguenti specifiche.

Lino:
In modo ciclico
- attende un comando
- esegue il comando appena ricevuto.
Per ottenere un nuovo comando usa il metodo prendiComando() della classe Console.
L'esecuzione dei comandi consiste di una semplice stampa a video.

DottTomas:
In modo ciclico
- impartisce un comando a caso
- si addormenta per un tempo casuale compreso tra 0 e 1 secondo.
Per impartire un comando usa il metodo inserisciComando() della classe Console.

Comando:
I comandi possibili sono solo due: CICALINO, TELEFONO

Console:
Una Console e' in grado di contenere numero massimo di comandi (quantita ' da
definire a piacere). I comandi sono gestiti secondo una disciplina FIFO.
La classe e' dotata di due metodi:
void inserisciComando(Comando c) inserisce il comando c nella coda.
Comando prendiComando() prende il comando da piu' tempo nella coda.

Prova:
Crea un oggetto Console condiviso tra Lino e DottTomas. Attiva i thread.

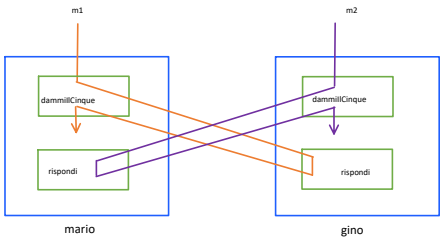
Politiche di funzionamento:
Se un comando rimane nella Console (coda) per piu' di 100 ms allora Lino viene
licenziato. Il controllo sui tempi di permanenza dei comandi nella coda avviene
solo in occasione di nuovi inserimenti o di prelievi.
Se il DottTomas tenta di inserire un nuovo comando nella coda e la coda e' piena
allora Lino viene licenziato.
Il licenziamento viene rappresentato con LicenziatoException (da definire).

Suggerimenti:
Per ottenere il tempo corrente usare System.currentTimeMillis()
https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis()

Esempio di esecuzione:
...
Lino: rosso!
Lino: il dottor Tomas non è in sede.
Lino: il dottor Tomas non è in sede.
Lino: verde!
Lino: spento!
Lino: rosso!
Lino: il dottor Tomas non è in sede.
Lino: verde!
Lino: il dottor Tomas non è in sede.
Lino: il dottor Tomas non è in sede.
Lino: spento!
Lino: il dottor Tomas non è in sede.
Tomas: Coda piena!
Lino: rosso!
Lino: Sono stato licenziato!

```
public class MioThread extends Thread {
  private Fratello f1;
  private Fratello f2;
  public MioThread(Fratello f1, Fratello f2) {
    this.f1 = f1;
    this.f2 = f2;
  }
  public void run(){
    while(true) {
      f1.dammillCinque(f2);
    }
  }
}
```

```
public class Lancia {
  public static void main(String[] args) throws Exception {
    Fratello mario = new Fratello("Mario");
    Fratello gino = new Fratello("Gino");

    MioThread m1 = new MioThread(mario, gino);
    m1.start();
    Thread.sleep(100);
    MioThread m2 = new MioThread(gino, mario);
    m2.start();
  }
}
```



1. *m1* invoca il metodo sincronizzato *dammillCinque()* su *mario* -> *m1* ha il lock su *mario*
2. *m2* invoca il metodo sincronizzato *dammillCinque()* su *gino* -> *m2* ha il lock su *gino*
3. *mario.dammillCinque()*, eseguito da *m1*, chiama *gino.rispondi()*. *m1* è adesso bloccato in attesa del lock su *gino* (posseduto da *m2*)
4. *gino.dammillCinque()*, eseguito da *m2*, chiama *mario.rispondi()*. *m2* è adesso bloccato in attesa del lock su *mario* (posseduto da *m1*)

Possibile soluzione:

```
public class FratelloND {

  private String nome;

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

  public String getNome(){
    return nome;
  }

  public void dammillCinque(FratelloND f) {
    FratelloND primo, secondo;
    if(getNome().compareTo(f.getNome()) < 0) {
```

```
    primo = this;
    secondo = f;
   } else {
    primo = f;
    secondo = this;
   }
   synchronized(primo) {
    synchronized(secondo) {
     System.out.println("dammi il cinque: " + getNome() + ", " +f.getNome());
     f.rispondi(this);
    }
   }
  }

  public void rispondi(FratelloND f) {
   System.out.println("rispondi: " + getNome() + ", " +f.getNome());
  }
}
```