Informatica 3

Part I: Programming Languages
Programmazione Concorrente

Laurea in Ingegneria Informatica
Politecnico di Milano
Polo di Milano Leonardo

Programmazione concorrente

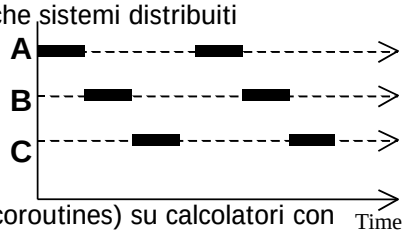
- Gli oggetti, nei linguaggi OO, permettono di suddividere un programma in unità di codice indipendenti
- A volte, è necessario o conveniente utilizzare **flussi di esecuzione** indipendenti
- **Programmazione concorrente**: programmi strutturati come insiemi di computazioni concorrenti (**tasks**),
  che procedono in parallelo (**flussi di controllo separati**) e occasionalmente interagiscono

Processi vs. Thread

- **Multitasking**: processi concorrenti a livello di sistema operativo
  - **Processo**: esecuzione di un intero programma
  - Spazi di indirizzamento separati: nessuna condivisione di dati
  - Gestione dispendiosa dei processi concorrenti, soprattutto per la comunicazione fra processi
- I thread (o lightwight processes) risolvono questi problemi
  - Sistemi operativi multithreaded
  - Flussi di controllo separati all'interno di uno stesso spazio di indirizzamento → eventuale condivisione di memoria e quindi comunicazione

Parallelismo fisico e logico

- Parallelismo fisico
  - calcolatori con piu' CPU o anche sistemi distribuiti
- Parallelismo logico:



  - Esecuzione di unità diverse (coroutines) su calcolatori con singolo processore
  - Il parallelismo è solo apparente, ma tuttavia fornisce vantaggi significativi
    - Esempio: un thread gestisce l'I/O (per es. il download di un file da internet), mentre altri thread gestiscono altre attività

Modelli di concorrenza

- **Cooperazione**: i processi/thread usano e poi rilasciano esplicitamente la CPU secondo la loro logica di esecuzione
  - Gestione semplice a runtime, ma...
  - ... la gestione della concorrenza è lasciata al programmatore: errori di cooperazione possono bloccare il sistema
- **Preemption**: l'esecuzione di un processo/thread può essere interrotta dal runtime per lasciare la CPU ad altre attività
  - Time slicing: ogni processo/thread può usare la CPU per un quanto di tempo, poi il controllo passa ad un'altra unità

Java

- **Processi**: si appoggia a meccanismi del Sistema Operativo
- **Thread**: supportati dal linguaggio e dall'interprete
  - Ogni thread è associato ad un' instanza della classe **`Thread`**
- Modello di concorrenza preemptive
  - Non sempre basato su time slicing (dipende dall'implementazione)
  - Se il time-slicing è disponibile, Java garantisce che i thread con uguale priorità vengano eseguiti in modalità round-robin
  - La priorità è gestita diversamente a seconda della specifica implementazione

Creazione di un thread in Java - 1

- Si crea una sottoclasse di *java.lang.Thread*
  - Ha i metodi necessari per creare ed eseguire un thread

```
public class MyThread extends Thread {
   private String message;
   public MyThread(String m) {message = m;}
   public void run() {
        for (int r=0; r<20; r++).
        System.out.println(message);
   }
}
public class ProvaThread {
   public static void main(String[] args) {
        MyThread t1,t2;
        t1=new MyThread("primo thread");
        t2=new MyThread("secondo thread");
        t1.start();
        t2.start();
   }
}
```

Un oggetto thread è definito a partire da una sottoclasse di **`Thread`**

Si ridefinisce il metodo **`run()`**

Creazione di un oggetto thread (allocazione in memoria)

Attivazione del thread

Problemi

- Per creare un nuovo thread bisogna creare nuove sottoclassi di **`Thread`**
- Non è possibile riutilizzare classi che già ereditano da un'altra classe diversa da **`Thread`**
  - Java non ammette ereditarietà multipla

Creazione di un thread in Java - 2

- Si implementa l'interfaccia *java.lang.Runnable*

```
public class MyThread implements Runnable {
   private String message;
   public MyThread(String m) {message = m;}
   public void run() {
        for (int r=0; r<20; r++)
           System.out.println(message);
   }
}
public class ProvaThread {
   public static void main(String[] args) {
        Thread t1,t2;
        MyThread r1,r2;
        r1=new MyThread("primo thread");
        r2=new MyThread("secondo thread");
        t1=new Thread(r1);
        t2=new Thread(r2);
        t1.start();
        t2.start();
   }
}
```

Si definisce una classe che implementa l'interfaccia **Runnable**

Si ridefinisce il metodo **run**

Si crea un oggetto **Runnable**

Si crea un oggetto thread mediante una new a cui e' passato un oggetto **Runnable**

Si attiva il thread tramite **start()**

Attivazione e terminazione

- **start()**
  - Ordina alla JVM la creazione di un nuvo thread
  - Richiama il metodo **run()** per il nuovo thread (sia quello della classe **Thread**, sia quello fornito da un **Runnable**)
- Si può pensare che **run()** sia simile ad un **main()**
  - Come **main()**, **run()** costituisce un punto di partenza per la JVM
- Quando il metodo **run()** termina (con successo o a causa di una eccezione), il thread termina

Non determinismo

- I thread sono eseguiti in maniera **non deterministica**
- L'esecuzione dei thread procede in un ordine non definito
- Su macchine diverse, lo stesso codice potrebbe generare un output diverso
  - Politiche di scheduling, caratteristiche del processore, …

Non determinismo

```
public class Test implements Runnable
{
  private int number;

  public Test(int number)
  {
    this.number = number;
  }

  public void run()
  {
    for (int i = 0; i < 2; i++)
    {
      System.out.print
       ("I am thread number " + number);
      System.out.println();
    }
  }
```

```
  public static void
  main(String[] argv)
  {
    for (int i = 0;
            i <
  Integer.parseInt(argv[0]);
  i++)
    {
      Thread thread =
        new Thread(new
  Test(i));
      thread.start();
    }
  }
}
```

12

Non determinismo – Output 1

- Una possibilità

  I am thread number 0
  I am thread number 1
  I am thread number 2
  I am thread number 3
  I am thread number 4
  I am thread number 0
  I am thread number 1
  I am thread number 2
  I am thread number 3
  I am thread number 4

- Un'altra possibilità

  I am thread number 0
  I am thread number 0
  I am thread number 2
  I am thread number 4
  I am thread number 4
  I am thread number 3
  I am thread number 1
  I am thread number 2
  I am thread number 3
  I am thread number 1

13

Non determinismo – Output 2

- Ancora un'altra possibilità

  I am thread number 0I am thread number 1I am thread number 2

  I am thread number 3
  I am thread number 4I am thread number 3

  I am thread number 2
  I am thread number 1
  I am thread number 4
  I am thread number 0

14

Non determinismo e risorse condivise

- x= 500, y= 400, z= 100 tre variabili di un programma concorrente
- La somma deve rimanere costante e uguale a 1000

Esecuzione 1

Thread 1	Thread 2
`read (x)`	
	`read (y)`
	`y = y-100`
	`read (z)`
	`z = z+100`
	`write y`
	`write z`
`read (y)`	
`read (z)`	
`s=x+y+z`	

Esecuzione corretta

s = 1000

Esecuzione 2

Thread 1	Thread 2
`read (x)`	
	`read (y)`
`read (y)`	
	`y = y-100`
	`read (z)`
	`z = z+100`
	`write y`
`read (z)`	`write z`
`s=x+y+z`	

Esecuzione scorretta

s = 1100

Controllo della concorrenza: Monitor

- Ad ogni oggetto Java è associato un **lock**
  - Una variabile che indica se l'oggetto è libero o già assegnato ad un altro thread in modo esclusivo
- È possibile controllare l'accesso concorrente di più thread a uno stesso oggetto, qualificando uno o più metodi dell'oggetto come **`synchronized`**
- Si dice anche che un oggetto contenente metodi synchronized è associato a un **monitor** (Hoare) che controlla il lock

Accesso al monitor - 1

- Sia **`t`** un thread che tenta di accedere ad un oggetto **`o`** invocando il metodo synchronized **`o.m`**
- Se nessun thread sta eseguendo **`o.m`**, allora **`t`** acquisisce il monitor su **`o`** e comincia ad eseguire **`m`**
  - Altri thread che tentano di accedere ad **`o`** attraverso altri metodi synchronized vengono sospesi fino a quando **`t`** non rilascia il monitor
  - Quando **`t`** termina, gli altri thread sospesi su **`o`** vengono risvegliati e competono per acquisire il monitor. Uno solo di essi lo acquisisce (scelta non deteministica)
- Se invece un altro thread **`t'`** detiene il lock, allora **`t`** si blocca fino a quando **`t'`** non rilascia il lock.

Accesso al monitor - 2

- Se un thread che detiene il lock per un oggetto richiama un altro metodo sincronizzato per lo stesso oggetto il thread non si sospende (*reentrant locking)*
- L'accesso in mutua esclusione vale solo per i metodi synchronized: l'accesso attraverso gli altri metodi può avvenire anche mentre un thread detiene il lock
- Anche i metodi **`static`** possono essere dichiarati synchronized (monitor associato alla classe, distinto da quello degli oggetti istanziati)

Accesso al monitor - 3

- La parola chiave **`synchronized`** non fa parte della signature del metodo
- Quando si sovrascrive un metodo synchronized in una sotto classe, il nuovo metodo non è automaticamente synchronized – è necessario specificarlo esplicitamente
  - Il metodo della super classe rimane synchronized; quindi, se si invoca **super**.foo(), la sincronizzazione è gestita così come atteso
- I metodi delle interfacce non possono essere dichiarati **synchronized**

19

Blocco synchronized

- Talvolta è necessario controllare l'accesso concorrente a porzioni limitate di codice

```
synchronized (obj) {
... codice critico ... }
```

  - N.B.: Maggiore è la porzione di codice controllata, minore è il parallelismo
- I metodi synchronized possono essere rappresentati come blocchi synchronized

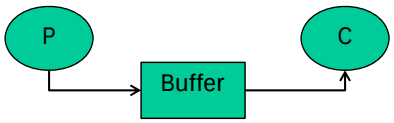
```
void synchronized m () {
... codice critico ...
}
```

≡

```
void m() {
synchronized (this) {
... codice critico ...
}
}
```

Sincronizzazione

- In alcuni casi oltre all'accesso mutuamente esclusivo sullo stato di un oggetto è necessario gestire il coordinamento delle operazioni dei thread
- Es.: **Produttore-Consumatore**



- Se P e C non sono sincronizzati:
  - P è più veloce di C: un valore prodotto da P potrebbe sovrascrivere un altro valore non ancora letto da C
  - C è più veloce di P: uno stesso valore potrebbe essere letto più volte

Primitive di sincronizzazione

- Associate a ogni oggetto (definite in **`Object`**)
- Sospendono un thread all'interno di un monitor (**`wait`**) o risvegliano un thread (**`notify`**) o tutti i thread sospesi (**`notifyall`**)
- Operano sul monitor associato a un oggetto:
  - Possono essere invocate dal thread solo quando questo ha acquisito il monitor (solo durante l'esecuzione di metodi **`synchronized`**)
  - Se si invocano su un oggetto per cui il thread non ha acquisito il monitor, viene generata un'eccezione **`IllegalMonitorStateException`**

Sospendere un thread

- **`wait()`**: se il thread invoca il metodo **`wait()`** su un oggetto per cui detiene il lock, il thread si sospende e rilascia il monitor per quell'oggetto
- I thread in attesa di accedere al blocco synchronized vengono risvegliati
  - Per poter riprendere l'esecuzione i thread risvegliati devono acquisire il monitor sull'oggetto
  - Non è detto che un thread riprenda immediatamente la sua esecuzione dopo una **`notify()`**
- Nel produttore-consumatore:
  - Se P verifica che C non ha ancora letto il valore di Buffer, P si sospende, cedendo il controllo a C
  - Se C verifica che P non ha ancora prodotto un nuovo valore, C può sospendersi cedendo il controllo a P

Risvegliare un thread

- I thread sospesi da una **`wait()`** possono essere risvegliati tramite:
  - **`notify()`**: risveglia un thread tra quelli sospesi (scelta non deterministica)
  - **`notifyall()`**: risveglia tutti i thread sospesi
    - È più costosa ma più sicura: **`notify()`** potrebbe risvegliare un thread la cui esecuzione non può procedere (deadlock)
- **`wait(int millisecondi)`**: sospende il thread per il tempo specificato
  - Allo scadere del tempo il thread è automaticamente risvegliato, anche senza una **`notify()`**
- Es. Produttore-Consumatore:
  - P in attesa di produrre un nuovo valore, viene risvegliato da C non appena C legge il valore corrente

Produttore-consumatore in Java

```
class Producer extends Thread {
  private Buffer buf;
  private int number;
  public Producer(Buffer b)
    { buf = b; }
  public void run() {
    for (int i = 0; i < 10; i++) {
      buf.put(i);
      System.out.println(
        "Producer put: " + i);
    }
  }
}

class Consumer extends Thread {
  private Buffer buf;
  public Consumer(Buffer b)
    { buf = b; }
  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = buf.get();
      System.out.println(
        "Consumer got: " + value);
    }
  }
}
```

```
class Buffer {
 private int seq;
 private boolean available = false;
 public synchronized int get() {
  while (!available) {wait();}
  available = false;
  notifyAll();
  return seq;
 }
 public synchronized void put(int value){
  while (available) {wait();}
  seq = value;
  available = true;
  notifyAll();
 }
}

public class ProducerConsumerTest {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    p.start();
    c.start();
  }
}
```

Il codice necessario a realizzare la sincronizzazione è evidenziato in rosso
Inoltre, sono state omesse le eccezioni `InterruptedException` associate a `wait`

Produttore-consumatore in Java

```
Producer put: 0
Producer put: 1
Producer put: 2
Producer put: 3
Producer put: 4
Producer put: 5
Producer put: 6
Producer put: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Producer put: 8
Producer put: 9
```

Codice senza sincronizzazione: il consumatore perde quasi tutti i dati

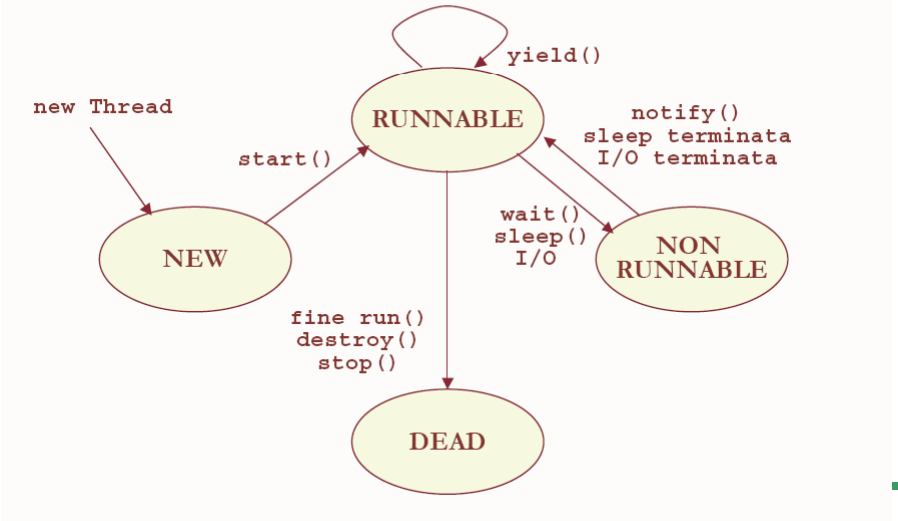
Codice con sincronizzazione: nessun dato perso
(Si noti come la stampa dei messaggi sia fuori ordine: essa infatti non è all'interno della sezione sincronizzata)

```
Producer put: 0
Producer put: 1
Consumer got: 0
Producer put: 2
Consumer got: 1
Producer put: 3
Consumer got: 2
Producer put: 4
Consumer got: 3
Producer put: 5
Consumer got: 4
Producer put: 6
Consumer got: 5
Producer put: 7
Consumer got: 6
Producer put: 8
Consumer got: 7
Producer put: 9
Consumer got: 8
Consumer got: 9
```

Altre primitive

- **`yeld()`**: cede esplicitamente l'uso del processore a un altro thread
  - Il thread rimane runnable: se nessun altro thread e' in attesa del processore, il thread può quindi proseguire la sua esecuzione
  - Rilascia i monitor posseduti dal thread
  - Necessaria nelle versioni della JVM non dotate di pre-emptive time slicing
- **`sleep(int millisecondi)`**: addormenta il thread corrente per il tempo specificato
  - Non rilascia il monitor
  - Sconsigliata all'interno di metodi/blocchi synchronized

Ciclo di vita di un thread



Spin Lock

```
while (!avaialble) {yeld();}
```
→spin lock

vs

```
while (!avaialble) {wait();}
```

- **Spin lock** (o *spin loop* o *busy wait*): con **`yeld()`** il thread non viene mai sospeso
  - **`wait()`** controlla invece la variabile solo quando una **`notify()`** notifica un cambiamento di stato
- Spreco del tempo di CPU per un tempo indefinito
- Potrebbe impedire l'accesso di altri thread all'oggetto condiviso
  - Se il thread che esegue **`yeld()`** ha priorità più alta degli altri rimarrà in esecuzione
  - **`wait()`** rimuove invece il thread dalla coda dei runnable.
- Uno spin lock è efficiente solo se i thread devono bloccarsi per un tempo breve, poichè evitano l'overhead per il cambio di contesto

Satefy e liveness

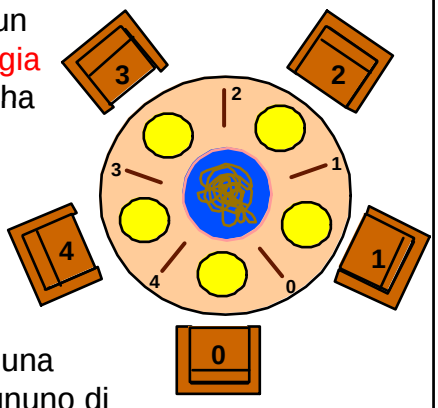
- **Safety**: il sistema non entra mai in stati non desiderati
  - available→buffer pieno; !available→buffer vuoto
- **Liveness**: il sistema prima o poi entra negli stati desiderati
  - Es.: un valore prodotto da P prima o poi è letto da C
- A seconda degli scenari, una proprietà può essere più rilevante dell'altra
  - Es.: impianti nucleari o interfacce utente

Problemi legati alla liveness

- **Deadlock**: ciascuna delle attività concorrenti è in attesa che altre attività rilascino alcune delle risorse condivise
  - Esempio: I task **T1** e **T2** utilizzano le risorse **A** e **B**. **T1** è bloccato perchè in attesa di accedere a una risorsa **A**. **T2** possiede il lock su **A**, ma è a sua volta bloccato in attesa della risorsa **B**. Il lock di **B** è posseduto da **T1.**
  - Evitare il deadlock è a carico del programmatore
- **Starvation**: un'attività concorrente, anche se abilitata ad essere eseguita, non riesce ad accedere alle risorse, che sono ottenute solo dalle altre attività

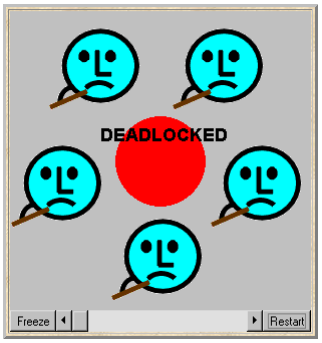
Il problema dei cinque filosofi

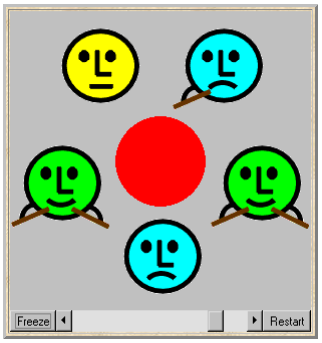
Cinque filosofi siedono attorno a un tavolo rotondo. Ogni filosofo mangia spaghetti o pensa. Ogni fiolosofo ha bisogno di due forchette per mangiare.



Tra ogni coppia di filosofi è posta una forchetta; essi concordano che ognuno di loro userà le forchette immediatamente a destra o a sinistra

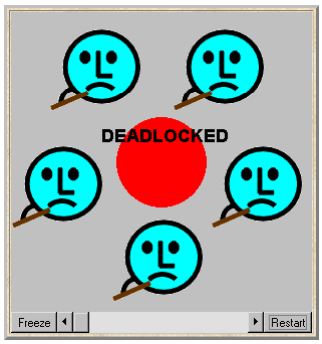
Il problema dei cinque filosofi





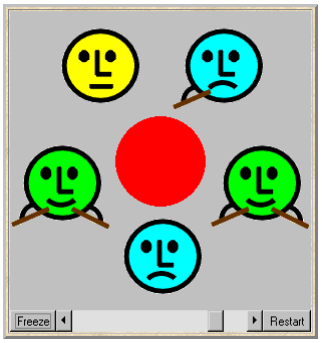
I cinque filosofi: deadlock

- Appena hanno fame, i fiolosofi prendono le forchette che sono disponibili
- Non rilasciano la forchetta fino a quando non hanno mangiato
- Esempio di deadlock: tutti i fiolosofi decidono di prendere la forchetta alla loro destra, poi tentano di prendere la forchetta alla loro sinistra
- Se nessuno rilascia la forchetta, si crea una dipendenza circolare sulla condizione di attesa di una risorsa (un deadlock)



I cinque fisolofi: soluzione deadlock-free

- La presa della forchetta è posta in una “sezione critica”
- I filosofi cercano di prendere per prima la forchetta a sinistra
- Se possiedono la forchetta a sinistra allora possono prendere anche quella a destra
- Se la destra non è disponibile, allora devono metter giù la forchetta a sinistra
- Le forchette devono essere usate solo mentre un filosofo mangia
- Due filosofi possono mangiare contemporaneamente; una forchetta rimarrà libera



Soluzione

```
class Philosopher extends Thread {
  private static Random rand = new Random();
  private static int counter = 0;
  private int number = counter++;
  private Chopstick leftChopstick;
  private Chopstick rightChopstick;
  static int ponder = 0; // Package access
  public Philosopher(Chopstick left, Chopstick right) {
    leftChopstick = left;
    rightChopstick = right;
    start();
  }
```

Soluzione

```
public void think() {
   System.out.println(this + " thinking");
   if(ponder > 0)
     try {
       sleep(rand.nextInt(ponder));
     } catch(InterruptedException e) {
       throw new RuntimeException(e);
     }}
 public void eat() {
   synchronized(leftChopstick) {
     System.out.println(this + " has "
       + this.leftChopstick + " Waiting for "
       + this.rightChopstick);
     synchronized(rightChopstick) {
       System.out.println(this + " eating");
     }
   }}
```

Soluzione

```
 public String toString() {
    return "Philosopher " + number;
  }
  public void run() {
    while(true) {
      think();
      eat();
    }
  }}
class Chopstick {
  private static int counter = 0;
  private int number = counter++;
  public String toString() {
    return "Chopstick " + number;
  }
}
```

Soluzione

```
public class DiningPhilosophers {
  public static void main(String[] args) {
    Philosopher[] philosopher = new
    Philosopher[Integer.parseInt(args[0])];
    Philosopher.ponder = Integer.parseInt(args[1]);
    Chopstick left = new Chopstick(),
      right = new Chopstick(),
      first = left;
    int i = 0;
    while(i < philosopher.length - 1) {
      philosopher[i++] = new Philosopher(left, right);
      left = right;
      right = new Chopstick();
    }
    if(args[2].equals("deadlock"))
      philosopher[i] = new Philosopher(left, first);
    else // Swapping values prevents deadlock:
      philosopher[i] = new Philosopher(first, left);
 }}
```

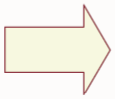
Terminazione di un thread

- Fine dell'esecuzione di **`run()`**
- **`stop()`**: metodo deprecato perchè insicuro
  - il thread rilascia i monitor precedentemente acquisiti (**`ThreadDeath`** exception propagation).
  - Se alcuni oggetti protetti dal lock sono in uno stato incoerente, altri thread sono comunque abilitati ad accedere a questi oggetti
  - Per motivi analoghi, **`suspend()`** e **`resume()`** sono deprecati
- Alternativa: codice che modifica il valore di alcune variabili per indicare che un thread deve essere terminato

Terminare un thread in maniera sicura

- Per evitare i problemi causati da **`stop()`**, si può strutturare il codice in maniera opportuna, ad esempio:

```
...
private Thread t;
public void start() {
  t = new Thread(this);
  t.start();
}
public void stop() {
  t.stop(); // NON SICURO!
}
public void run() {
  Thread thisThread =
    Thread.currentThread();
  while (true) {
    try { thisThread.sleep(1000); }
    catch(InterruptedException e){ ... }
    faiQualcosa ();
  }
}
```



```
...
private Thread t;
public void start() {
  t = new Thread(this);
  t.start();
}
public void stop() {
  t = null;
}
public void run() {
  Thread thisThread =
    Thread.currentThread();
  while (t == thisThread) {
    try { thisThread.sleep(1000); }
    catch(InterruptedException e){ ... }
    faiQualcosa ();
  }
}
```

Interrompere un thread

- La tecnica precedente non funziona quando il thread e' sospeso (**`wait()`**, **`sleep()`**, **`join()`**)
- **`interrupt()`**: metodo sicuro per terminare un thread sospeso
  - Solleva l'eccezione **`InterruptedException`**; il thread può quindi gestire l'interruzione a seconda del suo stato attuale
- Nel caso precedente:

```
public void stop() {
Thread t2 = t;
t = null;
t2.interrupt;
}
```

Se il thread è in esecuzione, si setta il flag di terminazione.

Se il thread è bloccato a causa della **`sleep()`**, si propaga una **`InterruptedException`**

Altre primitive per la gestione dei thread

- Il metodo **`join`** consente al thread chiamante di sospendersi in attesa della terminazione del thread corrispondente all'oggetto su cui il metodo è invocato

```
Thread t = new Thread(aRunnable);
t.start();
try {
  t.join();
} catch(InterruptedException e) { … }
```

- Il metodo (di classe) **`currentThread`** ritorna un riferimento al thread attualmente in esecuzione (cioè al thread chiamante, che ha effettuato l'invocazione)
  - **`Thread myself = Thread.currentThread`**
  - Si noti che se il thread è stato creato usando un **`Runnable`**, il thread così ritornato è diverso dall'oggetto **`Runnable`**

Gestire le priorità

- Un thread ha per default la priorità del thread che l'ha creato
- **`setpriority (int priority)`** permette di modificare la priorità
- Utile per thread che restano a lungo bloccati o che al contrario fanno uso intenso della CPU
- Lo scheduler dell'interprete Java selezionerà per primi i thread runnable con priorità più alta

Gruppi di thread

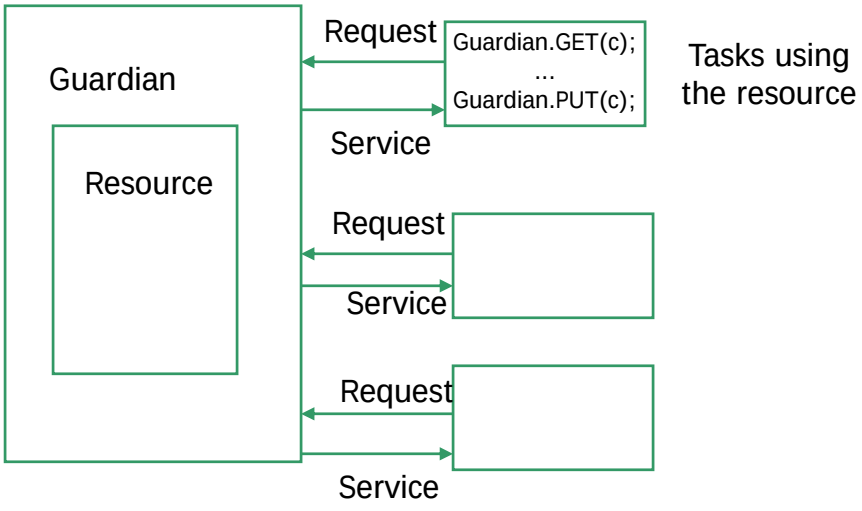
- Utili per manipolare insiemi di thread
  - Es.: cambiare la priorità per tutti i thread nel gruppo
- I thread creati appartengono a un gruppo di default
- Per creare un thread in un altro gruppo si usa il costruttore
  `Thread t = new Thread(group, runnable)`
- La classe **`ThreadGroup`** contiene i metodi per:
  - Gestire le collezioni di thread
  - Cambiare la priorità al gruppo
  - Terminare l'esecuzione dei thread nel gruppo (con gli stessi problemi gia' discussi per i singoli thread).

Considerazioni finali

- I thread hanno un costo
  - È sempre necessario definire un thread? Può essere sufficiente creare un nuovo oggetto in uno stesso flusso di controllo?
- La sincronizzazione ha un costo
  - Invocare un metodo synchronized può costare 4 volte un metodo normale
  - È necessario limitare il più possibile le sezioni critiche per massimizzare il parallelismo del sistema

Task synchronization in Ada

- Guardians and rendez-vous
- The Ada style of designing concurrent systems
- In Ada a shared object is active (whereas a monitor is passive)
  - it is managed by a guardian process which can accept rendez-vous requests from tasks willing to access the object



A Guardian Task

Accettazione non deterministica delle richieste di rendez-vous

```
loop
        select
                when  NOT_FULL
                        accept  PUT (C: in CHAR);
                        This is the body of PUT; the client  calls it as if it
                        were a normal procedure
                        end ;
                or
                when  NOT_EMPTY
                        accept  GET (C: out CHAR);
                        This is the body of GET; the client  calls it as if it
                        were a normal procedure
                end ;
        end select ;
end loop  ;
```

Put e Get sono le entry del task Guardian

Semantica dei rendez-vous

1) chiamata ad ENTRY (analoghe a chiamata di procedura)
2) il task chiamato deve fare una ACCEPT -> rendezvous

Di solito task body usa costrutto SELECT per decidere se e quando accettare.

```
SELECT
 WHEN C1 => PRG1
OR
 WHEN C2 => PRG2
OR
 PRG3 -- sempre aperta
ELSE
 PRG4
END SELECT;
```

- C1 e C2 sono alternative APERTE <=> C1, C2 = true
- PRGk puo` contenere ACCEPT o DELAY

1) esegui se esiste PRGk con Ck aperta e contenente una ACCEPT

2) se non esiste, allora esegui PRGk' con Ck' aperta e DELAY (il piu` piccolo DELAY aperto)

3) se tutto chiuso, esegui ramo ELSE. Se non c'e` ELSE, sospendi in attesa di apertura.