v 1.1-20181102

PROGRAMMAZIONE CONCORRENTE IN JAVA
- PRIMA PARTE -

Alessandro Ricci
a.ricci@unibo.it

SOMMARIO DEL MODULO

- Introduzione
  - thread e multi-threading
    - richiami (da Sistemi Operativi)
  - programmazione multi-threaded e paradigmi di programmazione
    - software and the concurrency revolution
- Programmazione multi-threaded in Java
  - API di base: classe Thread, interfaccia Runnable
  - Esempi: parallelismo & multi-core programming
  - Terminazione di un thread
- Multi-threading e GUI
  - come realizzare interfacce grafiche Swing reattive
- Bibliografia

PROGRAMMAZIONE MULTI-THREADED

- Oggigiorno la maggior parte delle applicazioni e sistemi software, sia standalone, sia di rete, è **multithreaded**.
  - un'applicazione è realizzata come un processo con più thread, che rappresentano attività di tipo diverso in esecuzione *concorrentemente*
    - ovvero: la loro esecuzione si sovrappone nel tempo
- Esempi:
  - Web Browser: un thread per visualizzare immagini / testo, uno per recuperare dati via connessioni di rete
  - Word Processor: un thread per visualizzare testo e grafica, uno per raccogliere input (da tastiera..) dell'utente, un altro ancora per controllare errori grammaticali del documento
  - Web Server: un thread per ogni client che sta interagendo con il sito
  - Video-giochi- ...

PERVASIVITA’

- Con il programma top è possibile osservare quanti thread sono creati nell’ambito di un processo UNIX

```
  PID COMMAND      %CPU   TIME    #TH  #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
14942 top          12.6%  1:16.91   1    16    25   416K   400K   816K  27.1M
14918 Grab          0.0%  0:06.38   3   231   217  4.36M  13.4M  26.5M   125M
14906 bash          0.0%  0:00.10   1    12    16   168K   832K   776K  18.2M
14905 login         0.0%  0:00.02   1    13    38   128K   372K   508K  26.9M
14904 Terminal     34.4%  0:27.80   3    61   166  2.79M+ 11.4M  7.91M+  111M+
14804 java          0.0%  0:17.85  10   193   116  4.98M  23.9M  10.1M   263M
14737 lookupd       0.0%  0:00.37   2    34    56   272K   680K   928K  28.5M
14598 Acrobat 5.    0.3%  3:03.87   4    77   486  1.77M  8.73M  3.39M   154M
14515 JavaApplic    0.3% 32:26.28  18   506   367  14.7M  37.5M  21.4M   378M
14485 java_swt      1.1% 27:51.19  25   >>>   458  75.2M  42.1M  90.1M   495M
14484 eclipse       0.0%  0:00.10   1    11    21     0K   516K   408K  27.2M
14279 TextEdit      0.0%  0:24.75   2    88   151  1.30M  9.00M  14.4M   111M
14135 Acrobat       1.9% 52:11.85   4    90   486  6.32M  9.25M  6.25M   210M
14090 DiskManage    0.0%  0:00.49   2    54    43     0K   760K   552K  35.7M
13714 Safari        0.0%  4:05.00   7   127   390  17.3M  17.2M  21.5M   156M
13146 Microsoft     0.0%  1:38.36   1    68   137   272K  5.31M   956K   100M
13145 Microsoft    25.3%  6:32:30   3   261  1212  85.6M  35.0M  41.3M   311M
...
```

- Come è possibile verificare, la maggior parte dei processi / applicazioni è multithreaded

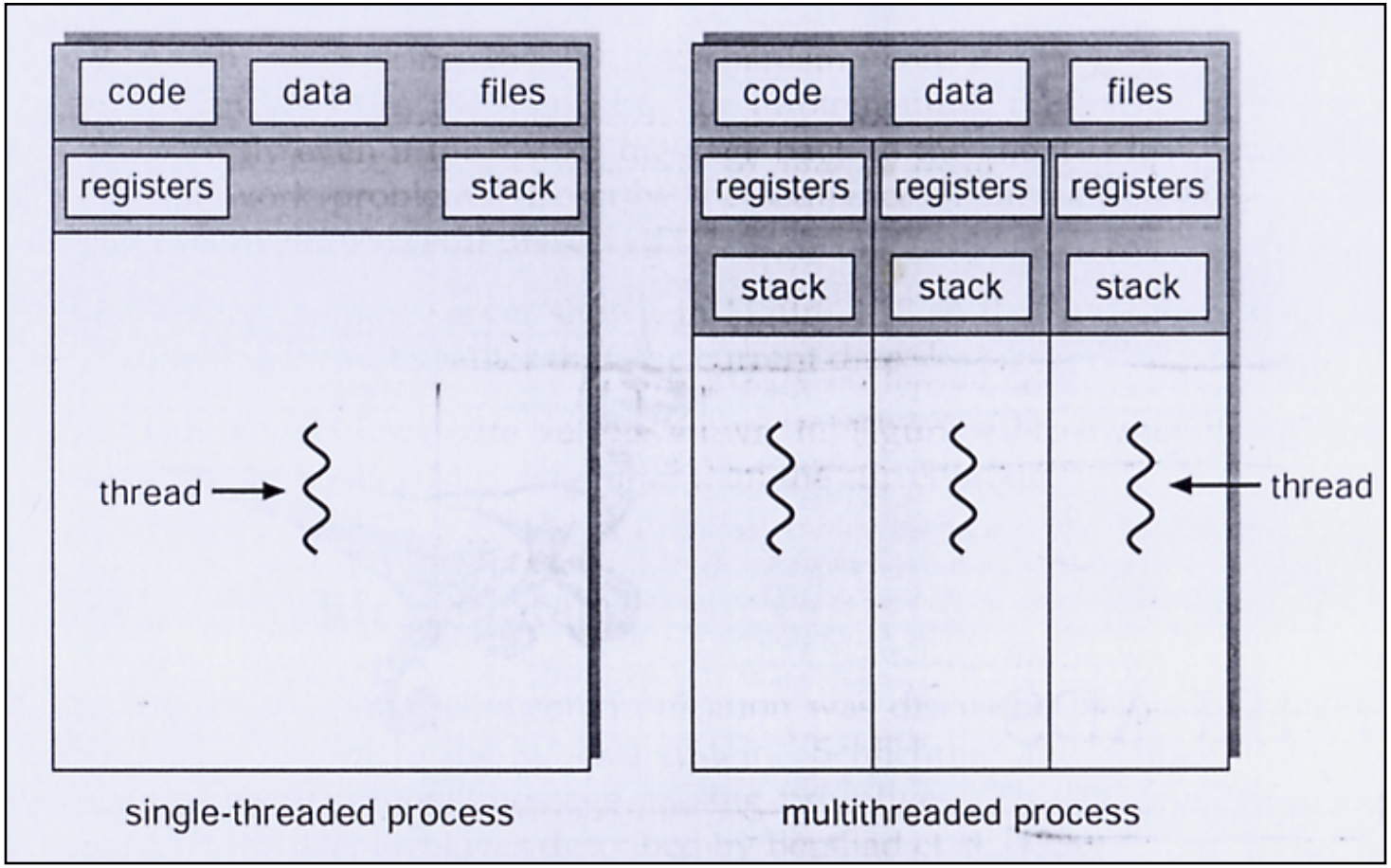
CONCORRENZA VS. PARALLELISMO

- **Esecuzione Concorrente**
  - l'esecuzione di due attività si sovrappone nel tempo
    - una inizia prima che l'altra sia terminata
  - non implica necessariamente l'esecuzione su processori (fisici o logici) distinti
    - può essere lo stesso processore (=> scheduling)
- **Esecuzione Parallela**
  - l'esecuzione si sovrappone nel tempo, *su processori distinti*

THREAD NEI S.O.: RICHIAMI

- Nei SO che supportano i thread, un thread diviene l'entità di base con cui si rappresenta un'attività in esecuzione su una CPU, un flusso di controllo
- Un thread nei SO moderni è caratterizzato da:
  - un thread ID
  - program counter
  - set di registri
  - un proprio stack
- In un medesimo processo possono essere creati e mandati in esecuzione più thread:
  - tutti i thread di un processo ne condividono il codice, dati e tipicamente altre risorse del sistema operativo (es: file aperti, segnali,..).
  - quindi a differenza di un processo (pesante) con un solo flusso di esecuzione, processi con più thread possono eseguire più attività simultaneamente.
- Il context switch fra thread è molto più leggero di un context switch a livello di processi

THREAD NEI S.O.: RICHIAMI



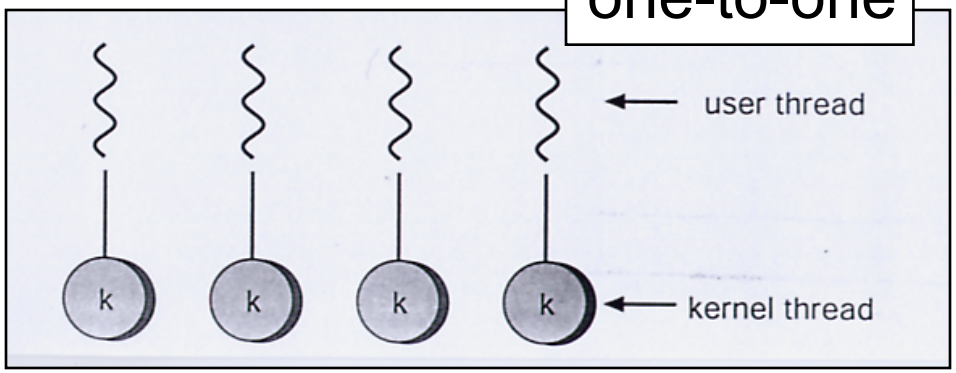
- Si parla di **multithreaded programming** per indicare la programmazione di sistemi mediante l'uso di thread.

KERNEL THREADS E USER THREADS

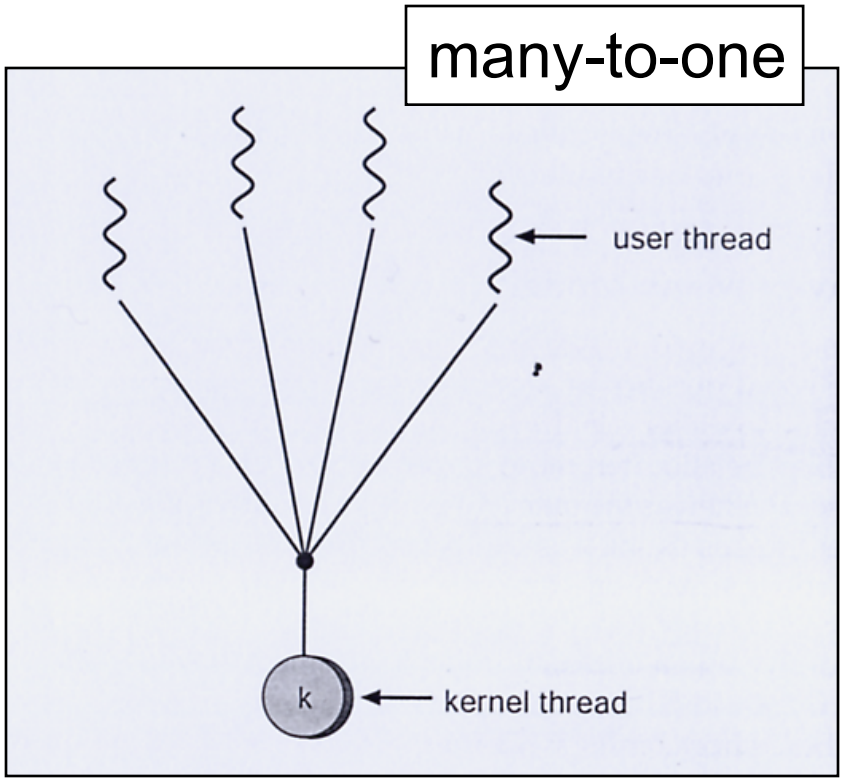
- Il supporto multithread può essere a due livelli
  - **user thread**
    - thread forniti allo user level, "simulato" mediante apposite librerie, senza il supporto diretto del kernel
  - **kernel thread**
    - thread forniti direttamente dal kernel del SO
      - è il caso di tutti i sistemi operativi moderni
- Vari approcci sono utilizzati al fine di mappare thread allo user level in thread al kernel level. I principali sono:
  - **Many-to-one**
    - più user thread vegono mappati su un kernel-thread. Gestione efficiente (user level), ma tutti i thread sono bloccati se si esegue una chiamata di sistema
  - **One-to-One**
    - uno user thread è mappato esattamente su un kernel thread.
  - **Many-to-Many**
    - più user thread vengono mappati su un insieme più piccolo o uguale

KERNEL THREADS E USER THREADS

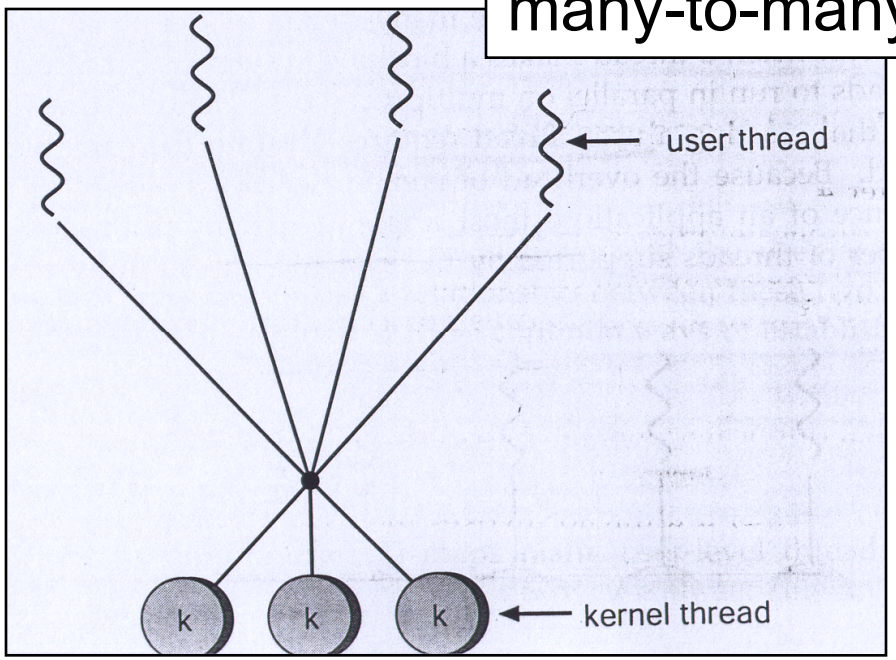
one-to-one



many-to-one



many-to-many



BENEFICI DEL MULTI-THREADING

- I benefici dall'utilizzo di thread vengono in genere classificati in quattro categorie principali:
  - **reattività (responsiveness)**
    - la possibilità di creare thread dedicati all'interazione con l'utente, concorrentemente all'esecuzione di operazioni a lungo termine, è fondamentale per creare interfacce utente (UI) adeguate
  - **condivisione di risorse (resource sharing)**
    - thread di uno stesso processo condividono dati e codice, che non devono quindi essere replicati
  - **performance (economy)**
    - la creazione e lo switch di contesto a livello di thread richiede molte meno risorse temporali e spaziali che non la creazione e switch fra processi.
  - **sfruttamento di architetture multiprocessore/multicore**
    - i benefici dell'uso di thread multipli si sentono ancor più su architetture multiprocessore, dove thread distinti possono essere messi in esecuzione concorrente su CPU diverse, aumentando notevolmente la concorrenza.

IMPORTANZA DELLA PROGRAMMAZIONE CONCORRENTE OGGI

- ***Think concurrent!***
  - programmazione concorrente non solo come insieme di meccanismi per aumentare le performance e/o reattività delle applicazioni..
  - ma come strumento concettuale e insieme di principi utili nella progettazione di applicazioni, sistemi
- Impatto a livello di paradigma
  - **problem solving**
    - come affrontare e pensare alla soluzione di problemi
  - **design e sviluppo di programmi d sistemi**
    - modularità, incapsulamento, riusabilità, estendibilità...
- “Software & concurrency revolution” by H. Sutter and J. Larus (bibliografia)

TERMINOLOGIA

- **Programmazione *parallela* (parallel programming)**
  - l'esecuzione di programmi (o parti di programma) si sovrappone nel tempo, andando in esecuzione su processori fisici separati
- **Programmazione *concorrente* (concurrent programming)**
  - l'esecuzione di programmi (o parti di programma) si sovrappone nel tempo, senza necessariamente andare in esecuzione su processori fisici separati
- **Programmazione *asincrona* (asynchronous programming)**
  - esecuzione di computazioni in modo asincrono, non bloccante
- **Programmazione *distribuita* (distributed programming)**
  - quando i processori sono distribuiti in rete e non c'è condivisione fisica della memoria
- **Programmazione multi-threaded**
  - quando le parti in esecuzione concorrente sono rappresentate da thread

DESIGN TASK-ORIENTED

- Organizzazione *task-oriented* dei programmi
  - un **task** rappresenta a livello logico un compito ben definito che può essere svolto in modo parzialmente o totalmente concorrente ad altri compiti
  - nel caso di prog. multithreaded => eseguito da thread
- Individuazione e gestione delle dipendenze fra i task
  - esempi:
    - risorse condivise e utilizzate in compiti diversi, informazioni che servono per svolgere un certo task e prodotte da un altro task
    - dipendenze temporali: un certo task deve essere svolto dopo che altri task sono stati completati...
  - nel caso di prog. multithreaded => porta all'uso di meccanismi per la sincronizzazione, coordinazione fra thread
- Decomposizione di task in sotto-task
  - division of labor

PROGRAMMAZIONE CONCORRENTE E OOP

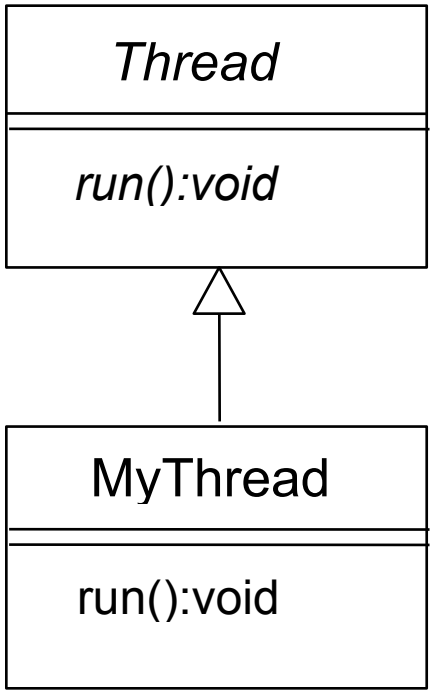
- Visione a livello di paradigma
  - **come integrare programmazione concorrente e OOP ?**
    - storicamente gli oggetti in OOP erano stati pensati per esser logicamente autonomi, concorrenti…
    - con comunicazione basata su scambio di messaggi
      - http://worrydream.com/EarlyHistoryOfSmalltalk/
  - come integrare programmazione multi-threaded e OOP?
- OOP + Multithreading
  - linguaggi OOP puri + librerie per multi-threading
    - esempi: linguaggi C/C++ e libreria PThread su sistemi POSIX, oppure libreria Win32 per sistemi Windows
  - linguaggi OOP estesi con astraz. di prima classe
    - modelli ad **attori**, oggetti concorrenti, agenti...
  - ibrido: linguaggio con meccanismi abilitanti + supporto a livello di libreria
    - Java, Objective C, C# (e CLR lang)

PROGRAMMAZIONE MULTI-THREADED IN JAVA

- Java è uno dei pochi linguaggi che fornisce supporto per i thread direttamente a livello di linguaggio, cercando di modellare tale nozione in termini di oggetto (classe).
  - la JVM si occupa quindi della creazione e gestione dei thread: come vengono mappati sui kernel thread dipende dal sistema (tipicamente one-to-one)
  - supporto esteso nella versione JDK5.0 con l'introduzione di una nuova libreria (**`java.util.concurrent`**)
- Un thread è rappresentato dalla classe astratta **`Thread`**, caratterizzata dal metodo astratto **run**, che definisce il comportamento (attività) del thread.
  - un thread concreto si definisce estendendo la classe Thread, ed specificando il comportamento del metodo run.
  - a tempo di esecuzione, un thread viene creato come un normale oggetto Java, e mandato in esecuzione invocando il metodo **start**.

LA CLASSE `Thread`

- la classe Thread è fornita direttamente nel package java.lang.




```
public class MyThread extends Thread  {
  public void run(){
    <corpo del thread>
    ...
  }
}
```

- Il codice eseguito dallo thread è specificato nel metodo **`run`**
  - il codice effettivo eseguito dipende dall'implementazione specifica descritta nel metodo run della classe derivata

ESEMPIO: UN CLOCK

- Thread `Clock` che visualizza in standard output la data e l'ora completi, ogni step millisecondi, con step specificato in fase di costruzione

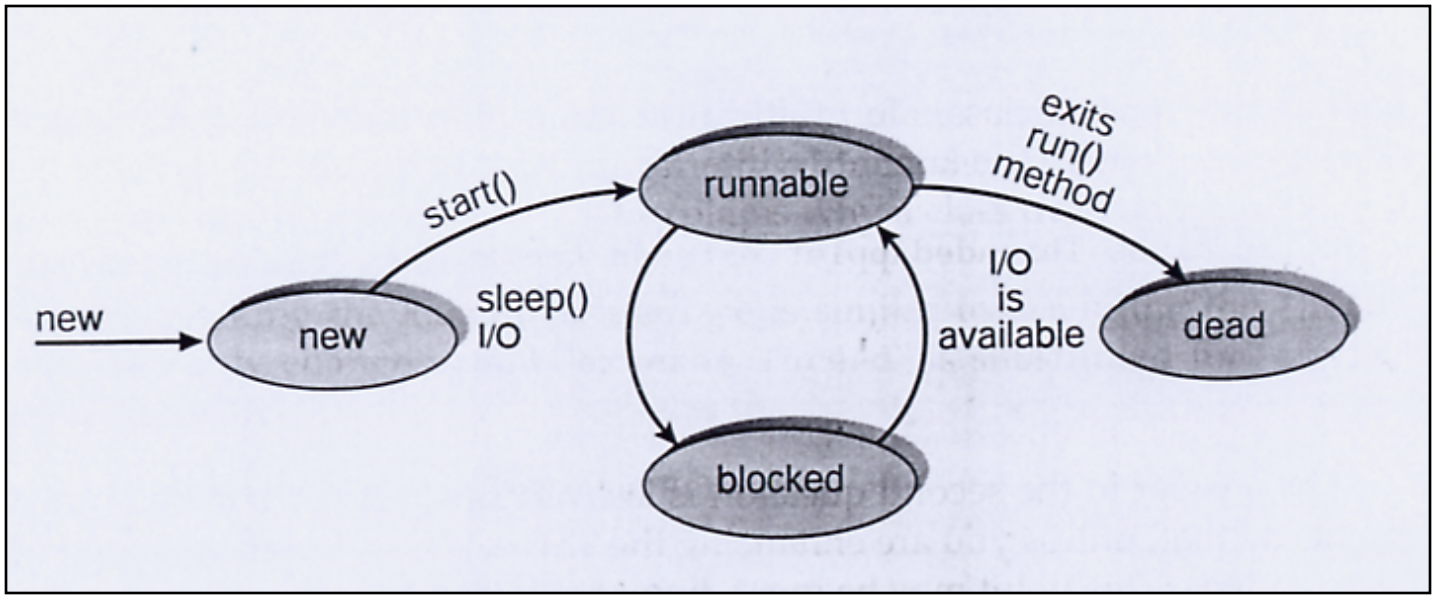
```
package oop.concur.part1;
import java.util.*;

public class Clock extends Thread {
  private int step;
  public Clock(int step){
    this.step=step;
  }
  public void run(){
    while (true) {
      System.out.println(new Date());
      try {
        sleep(step);
      } catch (Exception ex){
      }
   }
  }
}
```

INTERFACCIA DELLA CLASSE `Thread`

- Costruttori/metodi significativi della classe Thread sono:
  - **Thread**(String name)
    - costruisce il thread di nome name
  - void Thread.**sleep**(long ms)
    - metodo statico per addormentare il thread corrente di ms millisecondi
  - void destroy()
    - distrugge il thread
  - void setPriority(int priority)
    - cambia la priorità di esecuzione del thread
  - String getName()
    - ottiene il nome del thread
  - boolean isAlive()
    - verifica se il thread è 'vivo'
  - void **interrupt**()
    - interrompe l'attesa del thread (nel caso fosse in sleep o wait)
  - Thread Thread.**currentThread**()
    - metodo statico per recuperare il riferimento al thread corrente
- Metodi deprecati
  - stop, suspend, resume

STATI DI UN THREAD

- Un thread in Java può trovarsi in uno dei seguenti stati:
  - **NEW**: appena creato (con new)
  - **RUNNABLE:** elegibile di essere eseguito dalla JVM oppure direttamente in esecuzione. L'invocazione del metodo start() alloca memoria per il nuovo thread nella JVM, quindi viene invocato il metodo run(), che provoca il cambiamento dello stato del thread da NEW a RUNNABLE.
  - **BLOCKED**: stato in cui si trova il thread se esegue una operazione bloccante o sospensiva, come una operazione di I/O, oppure operazioni quali sleep()
  - **DEAD**: stato in cui si trova il thread quando termina il corpo del metodo run()



LANCIO (“SPAWNING”) DI UN THREAD

- Esecuzione del metodo **start** sull’oggetto thread
  - viene creato un nuovo flusso di controllo che manda in esecuzione il metodo run
- Esempio dell’orologio

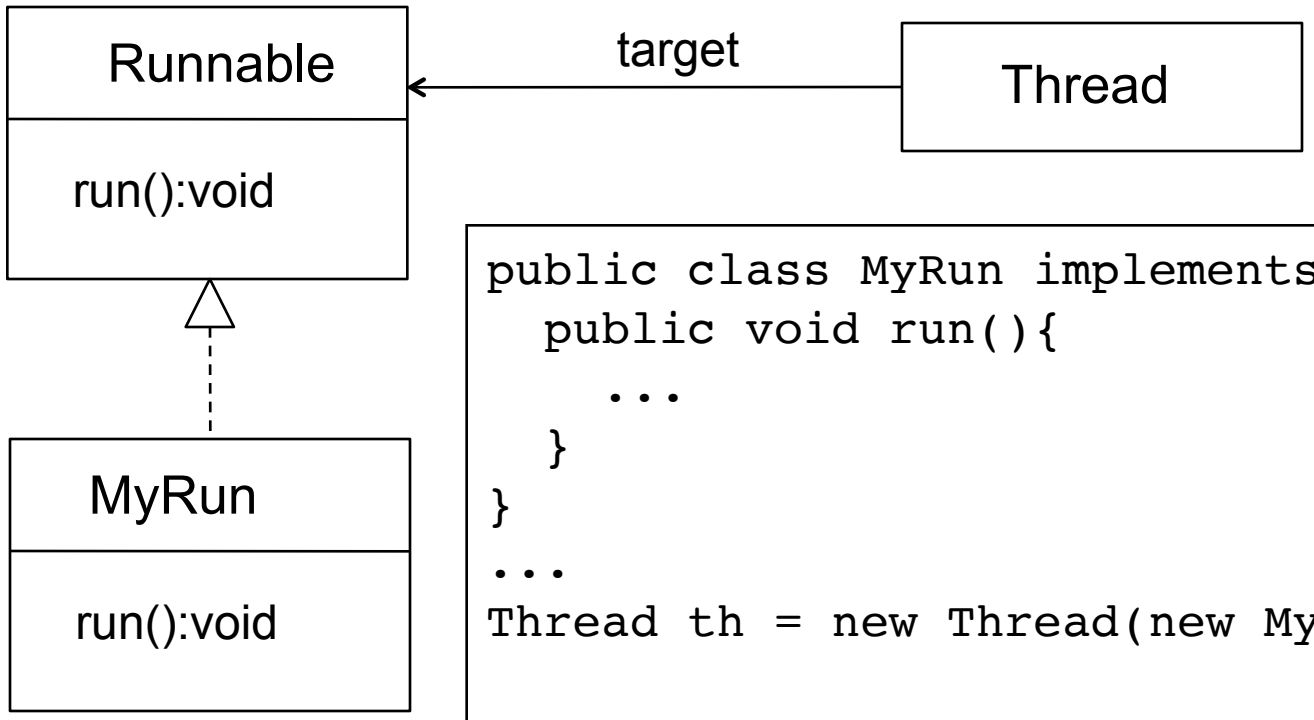
```
package oop.concur.part1;
import java.io.*;
public class TestClock {
  static public void main(String[] args) throws Exception {
    Clock clock = new Clock(1000);
    clock.start();

    BufferedReader reader = new BufferedReader(
                          new InputStreamReader(System.in));
    String input = null;
    do {
      input = reader.readLine();
      System.out.println("eco: "+input);
    } while (!input.equals("exit"));
    System.exit(0);
  }
}
```

NOTE

- Thread e main app
  - un'applicazione Java ha sempre almeno un flusso di controllo in esecuzione
    - è il main thread, che esegue il metodo statico main
  - in realtà ha più di un thread
    - garbage collector
    - profiler / debugger listeners, RMI listeners, etc
- start() vs. run()
  - cosa succede se invochiamo il metodo run() anziché il metodo start() per eseguire un thread?

INTERFACCIA Runnable

- Esiste anche un secondo modo per definire un thread, basato su interfacce, utile quando la classe che funge da thread è già parte di una gerarchia di ereditarietà e non può derivare da Thread




```
public class MyRun implements Runnable {
  public void run(){
     ...
  }
}
...
Thread th = new Thread(new MyRun());

// con lambda (Java vers. >= 1.8)
Thread th1 = new Thread(() -> { … });
```

CLOCK WITH RUNNABLE

- Esempio Clock usando un'espressione lambda per definire il task:

```
new Thread(() -> {
    while (true) {
        System.out.println(new Date());
        try {
            Thread.sleep(2000);
        } catch (Exception ex){
            ex.printStackTrace();
        }
    }
}).start();
```

- Approccio utile in particolare quando il thread rappresenta un compito di durata finita e limitata, da svolgere in modo asincrono rispetto al thread che lo lancia

MONITORING THREADS: JConsole TOOL

- Java Monitoring and Management Console, a tool grafico fornito con il JDK
  - usa il supporto nativo della JVM per monitoraggio performance e uso risorse
    - Java Management Extension (JMX) technology
  - http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html
- Tra le varie funzionalità, permette di monitorare i thread lanciati

INFINITE LOOPS

- Esempio che genera un numero N di thread che eseguono loop infiniti
  - check uso CPU con valori in input 1, 2, 4, 8...

```
public class InfiniteLoops {
    public static void main(String[] args){
        for (int i = 0; i < Integer.parseInt(args[0]); i++){
            System.out.println("Spawning #"+i+"...");
            new Thread(() -> { while (true){} } ).start();
        }
    }
}
```

<DEMO BOUNCING BALLS>

THINK CONCURRENT

- Fra i principali benefici della programmazione concorrente => aumentare le performance sfruttando al meglio le risorse HW parallele
- Nel caso della programmazione multi-threaded questo significa ripensare alla soluzione di problemi e la progettazione di algoritmi/ strutture dati / sistemi in termini di insiemi (eventualmente dinamici) di thread ognuno dei quali svolge una parte del compito complessivo e cooperano opportunamente al fine di ottenere l'obiettivo finale
- Primo esempio/problema: concur prime
  - stampare tutti i numeri primi da 1 a N, non necessariamente in oridine, con N fornito in input
    - per semplicità si consideri l'algoritmo più semplice per verificare se un numero è primo o meno...

ESEMPIO #1: CONCUR PRIME

- Programma in Java che stampa in standard output tutti i numeri primi da 1 a N non necessariamente in ordine, utilizzando P thread
  - N e P parametri dell'applicazione

```
package oop.concur.part1;

public class ConcurrentPrimeMain {
  public static void main(String[] args){
    if (args.length!=2){
      System.err.println("Args: <N primes> <N threads>");
      System.exit(1);
    }
    try {
      int n = Integer.parseInt(args[0]);
      int t = Integer.parseInt(args[1]);
      int nNumbers = n / t;
      int base = 0;
      for (int i=0; i < t-1; i++){
        new PrimeWorker(base,nNumbers).start();
        base+=nNumbers;
      }
      new PrimeWorker(base,n-base).start();
    } catch(Exception ex){
      System.err.println("Wrong arguments.");
      System.exit(2);
    }
  }
}
```

PRIME WORKER THREAD

```
package oop.concur.part1;

class PrimeWorker extends Thread {

  private int base;
  private int nNumbers;

  public PrimeWorker(int base, int nNumbers){
    this.base = base;
    this.nNumbers = nNumbers;
  }

  public void run(){
    for (int i = base+1; i <= base+nNumbers; i++){
      if (isPrime(i)){
        System.out.println(""+i);
      }
    }
  }

  private boolean isPrime(int num){
    int sq = (int)Math.sqrt(num);
    for (int i=2; i <= sq; i++){
      if ((num % i)==0){
        return false;
      }
    }
    return true;
  }
}
```

NOTE

- La suddivisione del lavoro è veramente equa (fair)?
- E se volessimo stamparli in ordine?
- C'è interazione fra i thread?
  - sono sempre corrette le stampe in standard output?
- Come variano le performance se si evita la stampa in output?

ESEMPIO #2: CONCURRENT SORTING

- Consideriamo come secondo esempio il problema dell'ordinamento degli elementi di un vettore
  - algoritmi: QuickSort, MergeSort, BubbleSort, HeapSort...
  - tutti sono sequenziali
- Soluzione multi-threaded? Quale design?...
- Soluzione con 2, 4, N processori
  - suddivisione del lavoro
  - aggregazione del risultato finale e necessità di sincronizzazione

UN PRIMO MECCANISMO DI SINCRONIZZAZIONE: JOIN

- Un thread può attendere la terminazione di un altro thread invocandone il metodo join

```
...
MyThread thread = new MyThread();
thread.start();
System.out.println("Waiting for thread termination...");
thread.join();
System.out.println("Thread completed.");
```

- Forme più articolate di sincronizzazione verranno discusse nella seconda parte

ESEMPIO #2: CONCURRENT SORTING CON JOIN

- Esempio con 2 thread (esempio completo nel materiale)

```
...
int[] v = Arrays.copyOf(data, data.length);
int middle = v.length / 2;
SortingWorker w1 = new SortingWorker(0,middle,v);
SortingWorker w2 = new SortingWorker(middle,v.length,v);
w1.start();
w2.start();
try {
  w1.join();
  w2.join();
  merge(v,data,0,middle,middle,v.length);
} catch (InterruptedException ex){}
...
```

```
class SortingWorker extends Thread {
  private int minIndex, maxIndex;
  private int[] data;
  public SortingWorker(int minIndex, int maxIndex, int[] data){
    this.minIndex = minIndex; this.maxIndex = maxIndex; this.data = data;
  }
  public void run(){
    Arrays.sort(data,minIndex,maxIndex);
  }
}
```

NOTE

- Come generalizzare la strategia a 4, 8, N thread/processi?
- Peso della parte sequenziale della strategia (merge...)
  - quale strategia per migliorare questo aspetto?

ULTERIORI ASPETTI DI BASE

- Terminazione di un thread
- Ottenere il numero di processori a disposizione

TERMINAZIONE DI UN THREAD

- La terminazione di un thread (thread cancellation) consiste nella terminazione della sua attività prima del suo completamento. Il thread da terminare prende in genere il nome di target thread.
  - Ad esempio si pensi ad un insieme di thread con il medesimo compito di ricerca di informazioni in un archivio: non appena uno trova le informazioni, gli altri possono essere fermati.
  - Oppure al thread relegato al caricamento di una pagina in un Web Browser, al momento in cui si preme il pulsante di stop.
- In genere si considerano due tipi di terminazioni:
  - asynchronous cancellation
    - un thread ne termina immediatamente il target thread
  - **deferred cancellation**
    - il target thread controlla periodicamente se deve terminare
- I problemi relativi alla terminazione di un thread sono per lo più legati alle risorse che tale thread può aver bloccato o comunque averne possesso: in particolare ciò è problematico nel caso di asynchronous cancellation.

TERMINAZIONE DI UN THREAD IN JAVA

- La terminazione di un thread può essere sia di tipo asincrono, sia deferred.
  - nel primo caso - *deprecato* - la terminazione avviene invocando il metodo **stop**()
  - nel secondo caso la terminazione avviene controllando nel metodo run() stesso che non si siano verificate le condizioni per la terminazione.
- Un esempio consiste nell'utilizzo del metodo `isInterrupted` che valuta se è stata richiesta l'interruzione del thread (mediante metodo `interrupt`): in tal caso si fa in modo di terminare il metodo run.

DEFERRED CANCELLATION: ESEMPIO

```
package oop.concur.part1;
import java.util.*;

public class StoppableClock extends Thread {
  private int step;
  private volatile boolean stopped;

  public Clock(int step){
    this.step = step;
    this.stopped = false;
  }

  public void run(){
    while (!stopped) {
      System.out.println(new Date());
      try {
         sleep(step);
      } catch (Exception ex){
        System.out.println("Interrupted!");
      }
   }
  }

  public void forceStop(){
    stopped = true;
    interrupt();
  }
}
```

ATTRIBUTO VOLATILE

- L'attributo volatile specificato per un campo implica che ogni thread che voglia accedere al campo in lettura, legga effettivamente il valore corrente in memoria e non una eventuale versione che tiene in cache/nello stack
  - questo potrebbe capitare in seguito ad ottimizzazioni operate dal compilatore e comporta problemi in caso di accessi concorrenti..
- In particolare:
  - l'accesso in lettura e scrittura di campi di tipo boolean è garantito essere atomico a livello di JVM
    - questo vale anche per int, non vale invece per double e long
  - per cui - nell'esempio - non ci sono corse critiche nell'accesso concorrente da parte del thread stesso nel metodo run (...! stopped...) e di un altro thread che invoca il metodo forceStop
  - ciò che invece può capitare - se non si specifica volatile - è che il valore di stopped letto in run non sia quello effettivamente in memoria, ma una versione in cache
    - eventualmente errata, se il campo è statp aggiornato da forceStop..

INTERRUPT

- Il metodo interrupt ha effetto sul thread chiamato nel caso esso sia bloccato su metodi come sleep, wait (prossimo modulo) che vengono quindi sbloccati generando un'eccezione di tipo InterruptedException
  - che è parte della signature del metodo
- Nell'esempio, nel caso in cui il thread StoppableClock sia bloccato sulla sleep, questo viene immediatamente sbloccato e quindi può uscire dal ciclo while

OTTENERE NUMERO PROCESSORI

- Funzionalità fornita dalla classe Runtime presente in java.lang
  - metodo **availableProcessors**()
    - restituisce il numero di processori HW logici presenti nel sistema
      - logici => incluso hyper-threading
  - da invocare sull'unico oggetto Runtime presente nel sistema (pattern singleton)

```
public class PrintAvailableProcs {
  public static void main(String[] args) {
    int nprocs = Runtime.getRuntime().availableProcessors();
    System.out.println("Available processors: "+nprocs);
  }
}
```

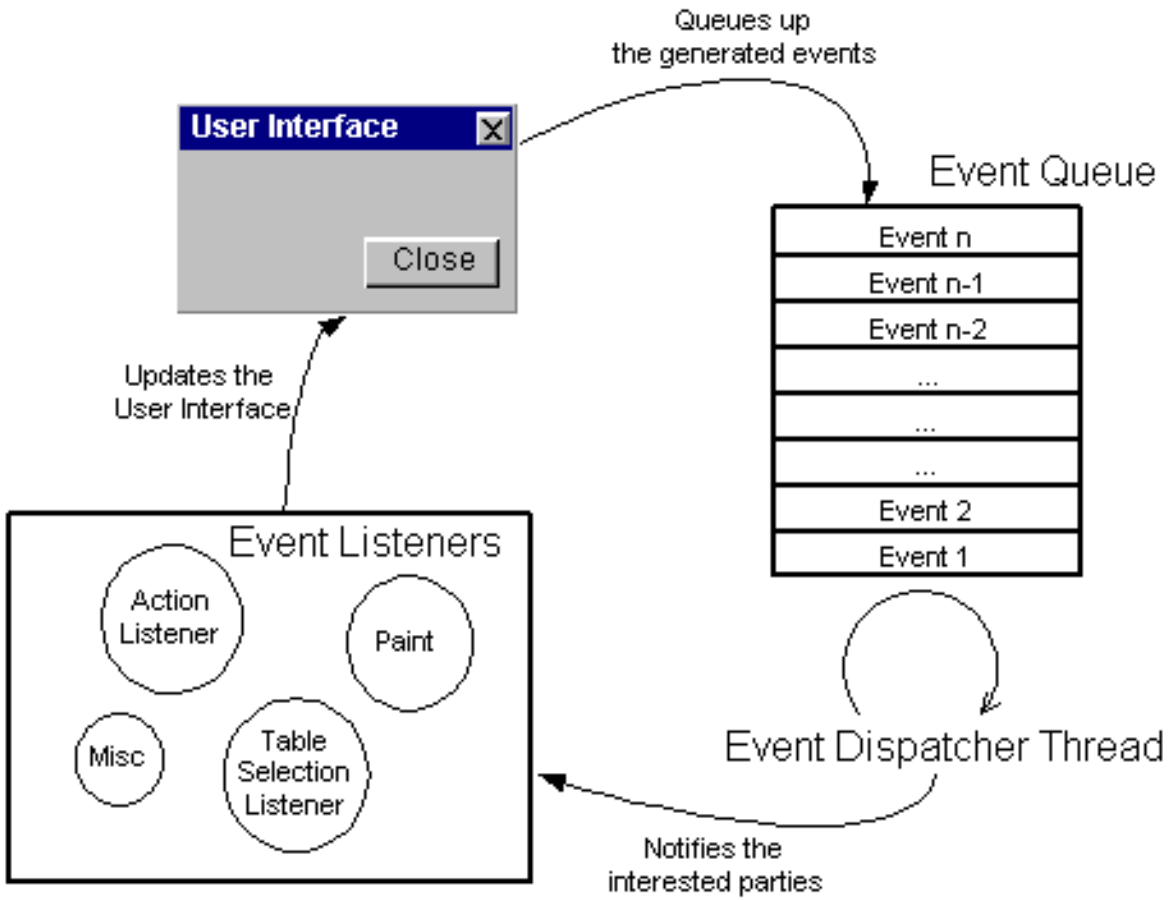
MULTI-THREADING E INTERFACCE GRAFICHE

- I sottosistemi grafici tipicamente usano un unico thread per processare gli eventi che concernono i componenti della GUI
  - tipicamente gestiti con una singola coda di eventi
- Un esempio è dato dalla libreria Java **Swing**
  - EDT (Event Dispatcher Thread)
  - creato inizializzato alla prima creazione di un componente Swing
  - elabora i vari eventi di qualsiasi frame o più in generale componente, chiamando opportunamente i listener agganciati
- Stessa architettura anche per **JavaFX**
- Conseguenze importanti
  - se l'esecuzione da parte del thread di un metodo del listener è particolarmente pesante dal punto di vista computazionale, l'interfaccia grafica "perde reattività"
    - caso estremo: se l'esecuzione da parte del thread di un metodo di un listener entra in loop infinito, tutta l'interfaccia grafica rimane bloccata

ESEMPIO: MANDIAMO IN STALLO Swing...

```
class MyFrame extends JFrame {

  public MyFrame(){
    super("Test Swing thread");
    setSize(120,60);
    setVisible(true);
    JButton button = new JButton("test");
    button.addActionListener((ActionEvent ev) -> {
      while (true){}
    });
    getContentPane().add(button);
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent ev){
        System.exit(-1);
      }
    });
  }
}

public class TestSwingThread {
  static public void main(String[] args){
    new MyFrame();
    new MyFrame();
  }
}
```

EVENT LOOPS



[Figura tratta da https://www.javaworld.com/article/2073477/swing-gui-programming/customize-swingworker-to-improve-swing-guis.html]

ESEMPIO: MANDIAMO IN STALLO JavaFX…

```
public class TestJavaFXThread extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Test javaFX thread");
        Button btn = new Button();
        btn.setText("test");
        btn.setOnAction((ev) -> {
             while (true){}
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

PROGRAMMAZIONE MULTI-THREADED PER APPLICAZIONI INTERATTIVE

- L'utilizzo di thread diviene dunque indispensabile per la realizzazione di applicazioni interattive e reattive, in cui l'interfaccia grafica deve rispondere con una certa prontezza all'input dell'utente
- Per realizzare interfacce reattive e corrette, valgono le seguenti regole:
  - non utilizzare mai il thread che gestisce gli eventi dell'interfaccia grafica per svolgere compiti pesanti
  - usare quindi thread separati per l'esecuzione di attività che possono comportare un certo impegno di risorse spazio temporali si devono
    - i thread possono essere creati on-the-fly nei listener, oppure si utilizzano pool di thread pre-esistenti e schemi produttore-consumatore
  - qualora thread diversi dall'EDT debbano interagire con oggetti/ componenti dell'interfaccia grafica, non si devono invocare direttamente i metodi di tali oggetti ma si devono utilizzare i metodi invokeLater and invokeAndWait in SwingUtilities

ESEMPIO: COSTRUIAMO UN CRONOMETRO...

- Nell'esempio che segue un conteggio viene pilotato mediante una interfaccia grafica dotata di tre pulsanti
  - alla pressione di start, il conteggio parte e viene incrementato ogni decimo di secondo, alla pressione di stop il conteggio si ferma e reset lo riporta a zero
  - viene usato un Counter generatore di eventi relativi al cambiamento del proprio valore
  - l'attività di incremento del conteggio viene eseguita da un thread separato, indipendente, creato allo start e fermato con lo stop
    - ciò permette di avere l'interfaccia sempre reattiva, nonostante l'attività di conteggio in corso

Counter

```
public class Counter {
  private ArrayList<CounterEventListener> listeners;
  private int cont, base;

  public Counter(int base){
    this.cont = base; this.base = base;
    listeners = new ArrayList<CounterEventListener>();
  }

  public void inc(){
    cont++;
    notifyEvent(new CounterEvent(cont));
  }

  public void reset(){
    cont = base;
    notifyEvent(new CounterEvent(cont));
  }

  public int getValue(){ return cont; }

  public void addListener(CounterEventListener l){ listeners.add(l); }

  private void notifyEvent(CounterEvent ev){
    for (CounterEventListener l: listeners){
      l.counterChanged(ev);
    }
  }
}
```

```
public class CounterEvent {
  private int value;
  public CounterEvent(int v){
    value = v;
  }
  public int getValue(){
    return value;
  }
}
```

```
public interface CounterEventListener {
  void counterChanged(CounterEvent ev);
}
```

CounterGUI

```
public class CounterGUI
  extends JFrame
  implements ActionListener,

CounterEventListener {

  private JButton start;
  private JButton stop;
  private JButton reset;
  private JTextField display;

  private Counter counter;
  private CounterAgent agent;

  public CounterGUI(Counter c){
    …
    counter = c;
    …
    start = new JButton("start");
    stop  = new JButton("stop");
    reset = new JButton("reset");
    stop.setEnabled(false);
    start.addActionListener(this);
    stop.addActionListener(this);
    reset.addActionListener(this);
    counter.addListener(this);
}
```

```
  ...
  public void actionPerformed(ActionEvent ev){
    Object src = ev.getSource();
    if (src==start){
      agent = new CounterAgent(counter);
      agent.start();
      start.setEnabled(false);
      stop.setEnabled(true);
      reset.setEnabled(false);
    } else if (src == stop){
      agent.interrupt();
      start.setEnabled(true);
      stop.setEnabled(false);
      reset.setEnabled(true);
    } else if (src == reset){
      counter.reset();
    }
  }

  public void counterChanged(final CounterEvent ev){
    SwingUtilities.invokeLater(() -> {
          display.setText(""+ ev.getValue());
   });
  }
}
```

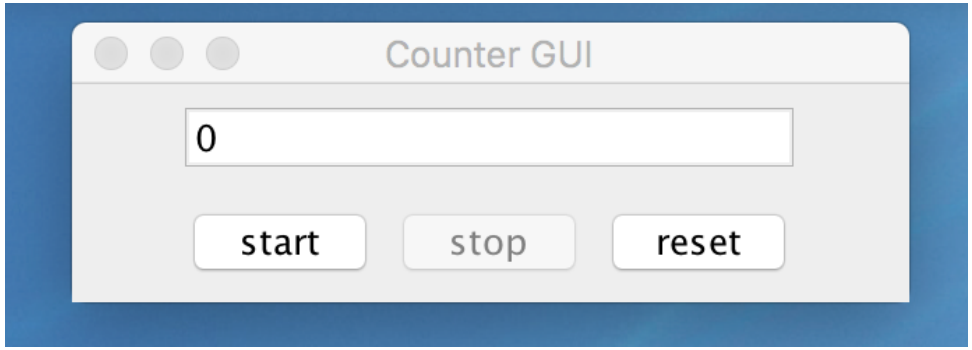
CounterAgent

```
public class CounterAgent extends Thread{
  private Counter counter;
  private volatile boolean stopped;

  public CounterAgent(Counter c){
    counter = c;
  }

  public void run(){
    stopped = false;
    while (!stopped){
      counter.inc();
      try {
        Thread.sleep(10);
      } catch(Exception ex){
      }
    }
  }

  public void interrupt(){
    super.interrupt();
    stopped = true;
  }
}
```

TestCounter

- TestCounter è l'applicazione vera e propria.

```
public class TestCounter {
  public static void main(String[] args) {
    Counter c = new Counter(0);
    javax.swing.SwingUtilities.invokeLater(() -> {
      new CounterGUI(c).setVisible(true);
    });
  }
}
```



THREAD-SAFETY IN SWING

- In Swing - come nella maggior parte dei GUI toolkit - le classi che rappresentano componenti dell'interfaccia non sono thread-safe
  - questo perché si presuppone che l'unico thread che vi acceda sia l'EDT
- Questo implica che se altri thread devono interagire con tali componenti, non lo possono fare direttamente, ma devono interagire con l'EDT
  - i metodi invokeXXX in SwingUtilities lo fanno inserendo il compito specificato (istanza di un una classe che implementa Runnable) in una coda che acceduta dall'EDT
    - analogo all'invio di un messaggio
  - **invokeLater** => accoda e non attende lo svolgimento del compito
  - **invokeAndWait** => accoda e attende che l'EDT abbia svolto il compito

BIBLIOGRAFIA PER APPROFONDIMENTI

- Concurrent Programming in Java: Design Principles and Pattern, 2/E - Doug Lea - Addison-Wesley Professional
  - testo di riferimento per la programmazione concorrente in Java
- Software and the Concurrency Revolution - by Herb Sutter, James Larus - ACM Queue - http://queue.acm.org/detail.cfm?id=1095421
- Multithreaded toolkits: A failed dream? - Graham Hamilton - http://news.jchk.net/article/a2c3e767100ce327f46c1ad9d0e89e7db19ab227