versione 1.0.0

Seminario Sistemi Distribuiti LA

II Facoltà di Ingegneria - Cesena

a.a 2004/2005

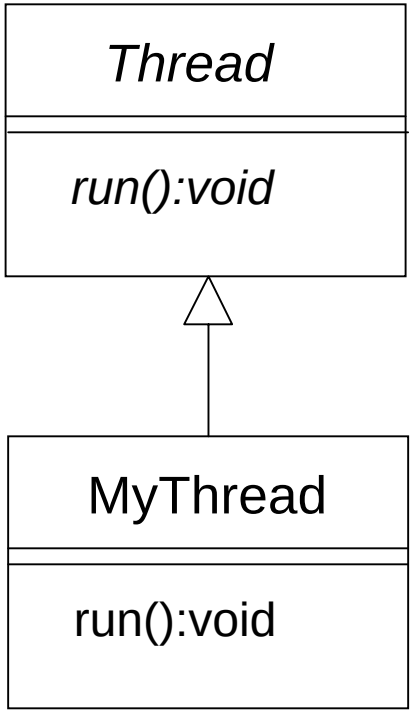
THREAD e SINCRONIZZAZIONE FRA THREAD IN JAVA

Threads 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).
- Uno 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 runtime 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 MyThreade extends Thread  {
  public void run(){
    ...
  }
}
```

- 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.

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**

Esempio: un orologio

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

```
package modulo2a;
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){
      }
   }
  }
}
```

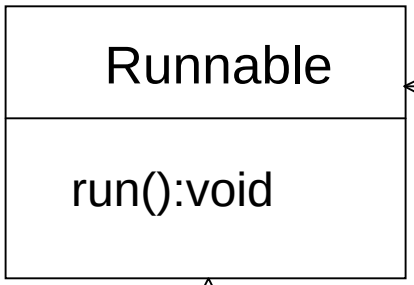
Esempio: Test dell'orologio

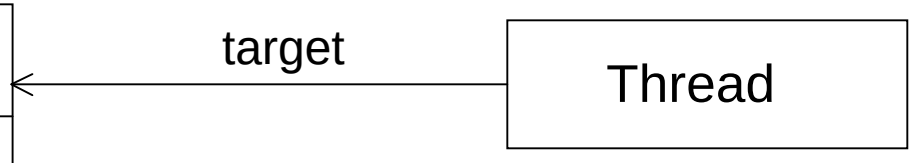
- Test dell'orologio. Per verificare che è in esecuzione concorrente rispetto al flusso di controllo principale, mentre l'orologio è in esecuzione si leggono informazioni da stadin e le si visualizzano in stdout

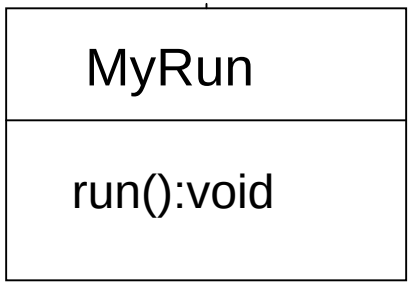
```
package modulo2a;
import java.io.*;
public class TestClock {
  static public void main(String[] args) throws Exception {
    Clock clock = new Clock(1000);
    clock.start();

    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);
  }
}
```

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`



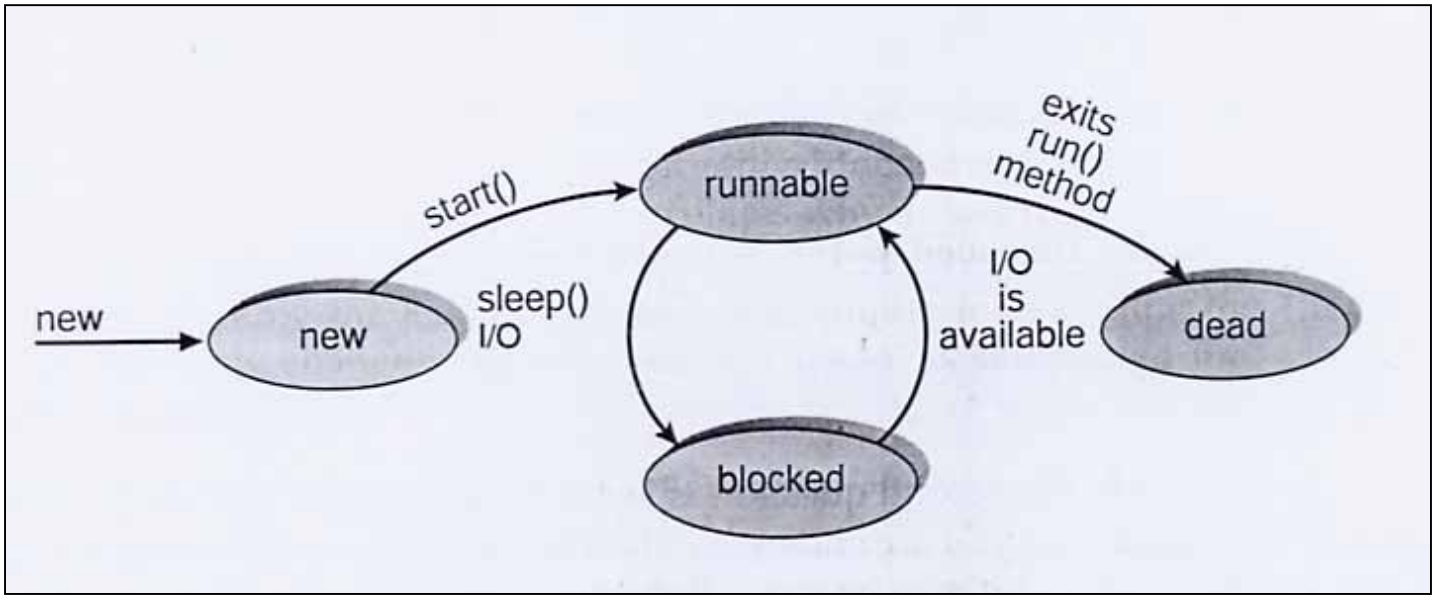





```
public class MyRun implements Runnable {
  public void run(){
    ...
  }
}
...
Thread th = new Thread(new MyRun());
```

Stati di un thread

- Uno 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()



Terminazione di 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 uno 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.

Thread e libreria grafica Swing

- Per gestire *tutti* gli eventi associati ai componenti attivi dell'interfaccia grafica, la libreria Swing utilizza un *unico* thread, creato inizializzato alla prima creazione di finestra che si esegue
- E' facile verificare che tutta l'attività concernente tutti i componenti Swing (anche di finestre distinte) è a carico del medesimo thread: è sufficiente creare due finestre, con due pulsanti una ciascuno, e registrare come ascoltatore di una un listener dal comportamento opportunamente errato, che ad esempio esegua un ciclo infinito oppure addormenti lo thread corrente per un tempo molto lungo. Si potrà verificare che non è più possibile interagire con nessuno dei componenti, né della medesima finestra, né di finestre distinte: Nessun evento è più generato / ascoltato
- E' solamente possibile muovere le finestre (ma non chiuderle ...), in quanto tale comportamento è gestito direttamente del sistema operativo sottostante

Esempio per mandare in stallo Swing

```
package modulo2a;
import javax.swing.*;
import java.awt.event.*;
class MyFrame extends JFrame implements ActionListener {
  public MyFrame(){
    super("Test Swing thread");
    setSize(100,50);
    JButton button = new JButton("test");
    button.addActionListener(this);
    getContentPane().add(button);
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent ev){
         System.exit(-1);
      }});
    }
    public void actionPerformed(ActionEvent ev){
      try { Thread.sleep(100000000);
      } catch (Exception ex){}
    }
}
public class TestSwingThread {
  static public void main(String[] args){
    new MyFrame().show();
    new MyFrame().show();
  }
}
```

Thread per applicazioni interattive

- L'utilizzo di thread è indispensabile per la realizzazione di applicazioni interattive, in cui l'interfaccia grafica deve rispondere con una certa prontezza all'input dell'utente
- Come pattern generale, si deve fare in modo di *non* utilizzare mai il thread di Swing per svolgere compiti pesanti, pena lo stallo di tutta l'interfaccia
  - per l'esecuzione di attività che possono comportare un certo impegno di risorse spazio temporali si devono usare thread separati
- 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 (1/2)

```
package modulo2a;

import java.util.*;

public class Counter {

  private ArrayList listeners;
  private int cont;
  private int base;

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

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

...
```

```
package modulo2a;

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

Counter (2/2)

```
package modulo2a;

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

```
    ....

    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){
      Iterator it = listeners.iterator();
      while (it.hasNext()){
        ((CounterEventListener)it.next()).counterChanged(ev);
      }
   }
}
```

CounterGUI (1/3)

```
package modulo2a;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


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;
  ...
```

CounterGUI (2/3)

```
public CounterGUI(Counter c){
  setTitle("Counter GUI");
  setSize(300,100);

  counter = c;

  display = new JTextField(5);
  ...

  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);
  counter.reset();
}
```

CounterGUI (3/3)

```
  ...
  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(CounterEvent ev){
    display.setText(""+ ev.getValue());
  }
}
```

CounterAgent

```
package modulo2a;

public class CounterAgent extends Thread{
  private Counter counter;
  private 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.

```
package modulo2a;

public class TestCounter {
  public static void main(String[] args) {
    Counter c = new Counter(0);
    new CounterGUI(c).show();
  }
}
```

Metodi synchronized

- Dichiarando un metodo **synchronized** si vincola l'esecuzione del metodo ad un solo thread per volta.
- I thread che ne richiedono l'esecuzione mentre già uno sta eseguendo vengono automaticamente sospesi dalla JVM, in attesa che il thread in esecuzione esca dal metodo
- Dichiarando più metodi synchronized il vincolo viene esteso a tutti i metodi in questione: se un thread sta eseguendo un metodo synchronized, ogni thread che richiede l'esecuzione di un qualsiasi altro metodo synchronized viene sospeso e messo in attesa.
  - I metodi synchronized sono dunque mutuamente esclusivi
- Da notare che tale vincolo non vale nei confronti dei metodi *non synchronized:* il fatto che un thread sta eseguendo un metodo synchronized, non vieta ad altri thread di eseguire concorrentemente eventuali metodi *non* synchronized dell'oggetto stesso

Esempio

- Nell’esempio che segue 2 thread (di classe UserA) accedono concorrentemente ad un oggetto condiviso di classe A, invocando il metodo synchonized m
- Dalla trace (traccia) d’esecuzione si vede come il secondo thread “entra” nel metodo solo quando il primo è uscito

Classe A synchronized e thread UserA

```
package modulo2f;
public class A {
  public synchronized void m(){
    System.out.println("Thread "+Thread.currentThread()+" entered.");
    try {
      Thread.sleep(5000);
    } catch (Exception ex){
    }
    System.out.println("Thread "+Thread.currentThread()+" exited.");
  }
}
```

```
package modulo2f;

public class UserA extends Thread {
    private A obj;
    public UserA(A obj){
      this.obj = obj;
    }
    public void run(){
      log("before invoking m");
      obj.m();
      log("after invoking m");
    }
    private void log(String msg){
      System.out.println("["+Thread.currentThread()+"] "+msg);
    }
}
```

Test

- Eseguendo il test è possibile verificare come che userB riesce ad entrare nel metodo m solo quando userA è uscito

```
package modulo2f;
public class TestUserA {
    public static void main(String[] args) {
        A obj = new A();
        UserA userA = new UserA(obj);
        UserA userB = new UserA(obj);
        userA.start();
        try {
            Thread.sleep(500);
        } catch (Exception ex){
        }
        userB.start();
    }
}
```

```
DEIS-Alessandro-Notebook:~/SOLA/Workspace/bin aricci$ java modulo2f.TestUserA
[Thread[Thread-0,5,main]] before invoking m
Thread Thread[Thread-0,5,main] entered.
[Thread[Thread-1,5,main]] before invoking m
Thread Thread[Thread-0,5,main] exited.
[Thread[Thread-0,5,main]] after invoking m
Thread Thread[Thread-1,5,main] entered.
Thread Thread[Thread-1,5,main] exited.
[Thread[Thread-1,5,main]] after invoking m
```

Mutua esclusione

- Un utilizzo immediato di synchronized è la mutua esclusione rispetto all'esecuzione di un metodo da parte di N thread, o più in generale di tutti i metodi che possono interferire fra loro
- Come esempio banale riprendiamo il contatore descritto nel modulo precedente: si evitano le interferenze viste nel caso in cui molteplici thread ne usino una istanza concorrentemente defindendo i metodi del contatore che possono interferire come synchronized
  - si evita l'uso di semafori

Counter Synchronized

```
package modulo2f;

public class CounterSynchronized implements ICounter {
  private int cont;

  public SynchronizedCounter(){
    cont = 0;
  }

  public synchronized void inc(){
    cont++;
  }

  public synchronized void dec(){
    cont--;
  }

  public synchronized int getValue(){
    return cont;
  }
}
```

```
package modulo2f;

public interface ICounter {
  void inc();
  void dec();
  int getValue();
}
```

Utilizzatore A del counter (immutato)

```
package modulo2f;

public class CounterUserA extends Thread {
    private int ntimes;
    private ICounter counter;
    public CounterUserA(ICounter c, int n){
        ntimes=n;
        counter = c;
    }
    public void run(){
        log("starting - counter value is "+counter.getValue());
        for (int i=0; i<ntimes; i++){
            counter.inc();
        }
        log("completed - counter value is "+counter.getValue());
    }
    private void log(String msg){
        System.out.println("[COUNTER USER A] "+msg);
    }

}
```

Utilizzatore B del counter (immutato)

```
package modulo2f;

public class CounterUserB extends Thread {
    private int ntimes;
    private ICounter counter;
    public CounterUserB(ICounter c, int n){
        ntimes=n;
        counter = c;
    }
    public void run(){
        log("starting - counter value is "+counter.getValue());
        for (int i=0; i<ntimes; i++){
            counter.dec();
        }
        log("completed - counter value is "+counter.getValue());
    }
    private void log(String msg){
        System.out.println("[COUNTER USER B] "+msg);
    }

}
```

Test

- Il test è analogo a quanto visto nel modulo precedente

```
package modulo2f;
public class TestSynchronizedCounter {
    public static void main(String[] args){
        ICounter c = new CounterSynchronized ();
        int ntimes = Integer.parseInt(args[0]);
        CounterUserA agentA = new CounterUserA(c,ntimes);
        CounterUserB agentB = new CounterUserB(c,ntimes);
        agentA.start();
        agentB.start();
    }
}
```

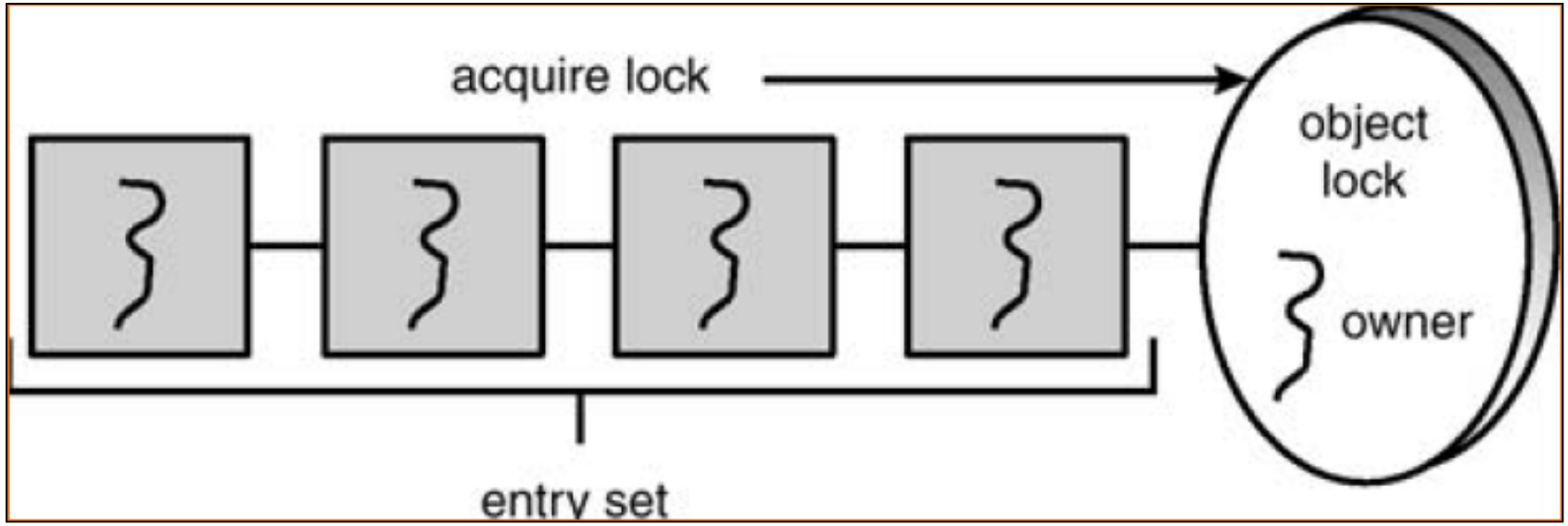
- L'utilizzo di synchronized fa in modo che non ci siano interferenze e il conteggio risultante alla fine è sempre 0

```
$ java modulo2f.TestSynchronizedCounter 10000000
[COUNTER USER A] starting - counter value is 0
[COUNTER USER B] starting - counter value is 723829
[COUNTER USER A] completed - counter value is 210926
[COUNTER USER B] completed - counter value is 0
```

Metodi synchronized: funzionamento

- A livello di implementazione il meccanismo di coordinazione synchronized è realizzato mediante un lock associato nativamente ad ogni oggetto.
- L'esecuzione di un metodo synchronized comporta prima l'acquisizione del lock: se il lock è già posseduto da un altro thread, il richiedente è bloccato / sospeso e inserito nella lista dei thread in attesa, chiamata **entry set** del lock. Se il lock è disponibile invece, il thread diviene proprietario del lock dell'oggetto ed esegue il metodo.
- Quando il proprietario rilascia il lock - o perché ha terminato l'esecuzione del metodo o perché deve esser sospeso, in seguito all'esecuzione di una wait - se l'entry set non è vuota, viene rimosso uno thread e ad esso viene assegnato il lock, ripristinandone l'esecuzione.
- La JVM non specifica alcuna politica di gestione dell'entry set: tipicamente viene utilizzata una politica FIFO (code).

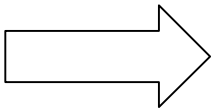
[Figura] Lock ed entry set



Monitor in Java

- Il meccanismo synchronized permette di realizzare agilmente monitor in Java: è sufficiente definire synchronized tutti i metodi che fungono da ‘entry’ nel monitor

```
monitor XXX {

  entry op0(...){
    ...
  }

  entry op1(...){
    ...
  }
  ...

}
```



```
class XXX {

  public synchronized op0(...){
    ...
  }

  public synchronized op1(...){
    ...
  }
  ...

}
```

Blocchi synchronized

- E' possibile "proteggere" con synchronized non solo metodi interi, ma solo porzioni di metodi, definendo dei blocchi synchronized:

```
...
synchronized(Obj) {
        < synchronized block >
        ...
}
...
```

- Il codice all'interno del blocco synchronized viene eseguito dal thread corrente solo dopo aver ottenuto il lock dell'oggetto *Obj* specificato come argomento
- Un metodo dichiarato synchronized non è altro che un metodo il cui corpo è racchiuso da un blocco synchronized:

```
synchronized(this) {
        <corpo del metodo>
}
```

Sezioni critiche con blocchi synchronized

- I blocchi synchronized permettono di realizzare sezioni critiche all'interno di metodi, senza necessariamente rendere tutto il metodo synchronized

```
Object mutex = new Object();
...

public void someMethod(){
  nonCriticalSection();

  synchronized (mutex){
    criticalSection();
  }

  nonCriticalSection();
}
```

mutex in questo caso è un campo privato della classe

criticalSection() indica qualsiasi parte (metodo) della classe che deve essere eseguita come sezione critica

Note su synchronized

- **synchronized** non fa parte della signature del metodo in quanto tale, ma è una sorta di attributo con cui viene annotato il metodo, come aspetto non funzionale
- Questo implica in particolare che:
  - non viene ereditato da classi che estendono la classe base in cui compare
  - non ha senso specificarlo nelle interfacce
- Relazioni fra invocazioni di metodi del medesimo oggetto:
  - Un metodo synchronized può invocare un altro metodo synchronized sul medesimo oggetto, senza “attese circolari”
  - Se un metodo synchronized chiama un altro metodo non synchronized sul medesimo l’oggetto, il lock comunque rimane
  - se un metodo non synchronized invoca un altro metodo synchronized dell’oggetto, viene acquisito il lock normalmente

Wait e notify

- L'altro meccanismo fondamentale di sincronizzazione fornito in Java è dato dalle operazioni **wait, notify** e **notifyAll**, fornite come metodi pubblici della classe Object, classe base di qualsiasi nuova classe definita

```
void wait() throws InterruptedException;
void wait(lont timeout) throws InterruptedException;
void notify();
void notifyAll;
```

- Tali operazioni hanno esattamente la semantica vista per le **condition variable**: ogni thread che esegue una wait su un oggetto O, viene sospeso fin quando un qualsiasi altro thread non esegua una notify o notifyAll sul medesimo oggetto O
  - Di fatto in questo modo ogni oggetto Java può fungere anche da condition variable

Condition variable in Java

- Il medium di coordinazione condition variable è realizzabile in modo immediato usando synchronized + wait / notify

```
package modulo2f;

public class Condition {

    public synchronized void wait_condition(){
        try {
            wait();
        } catch (InterruptedException ex){
        }
    }

    public synchronized void signal_condition(){
        notify();
    }

}
```

Wait / notify: funzionamento

- Il meccanismo di wait/notify è implementato mediante un **wait set**, insieme di thread che hanno eseguito una wait sull'oggetto e attendono di esser notificati.
- Alla costruizione dell'oggetto il wait set è inizialmente vuoto: quando un thread esegue una wait sull'oggetto, esso viene sospeso e inserito nel wait set.
- L'esecuzione di una notify invece comporta la selezione (arbitraria) di un qualsiasi thread in attesa nel wait set, che viene rimosso dal set e quindi risvegliato.

Relazione fra i meccanismi

- Un aspetto importante e sottile che lega di per sé i due meccanismi precedenti, concerne il fatto che per poter eseguire una wait su un oggetto, un thread **deve prima averne ottenuto il lock.**
- Questo implica che i metodi wait / signal su un oggetto X possono essere invocati solamente

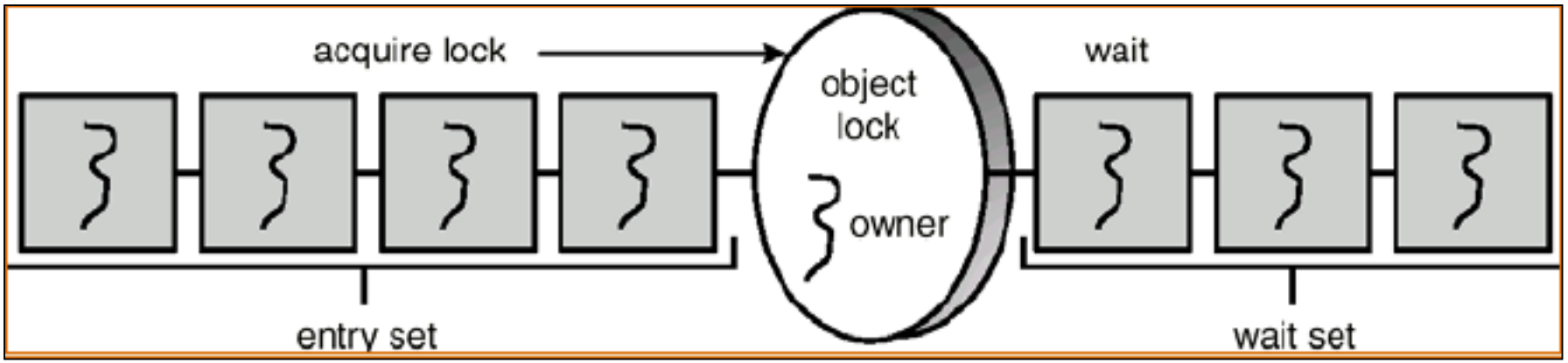
  1) o all'interno di un metodo synchronized dell'oggetto X stesso

  2) o all'interno di un blocco synchronized:

```
synchronized(Obj) {
        < synchronized block >
        Obj.wait(); // oppure Obj.notify();
        ...
}
```

  Affinché un thread possa eseguire il blocco <synchronized block> è necessario che prima ne acquisisca il lock. Se un metodo wait o signal viene eseguito da un thread su un oggetto senza possederne il lock, viene generata una eccezione dalla JVM.

[Figura] Wait Set ed Entry Set



Esempio: sincronizzazione di attività

- Nell'esempio che segue si sincronizza l'attività di due thread mediante l'utilizzo condiviso di un oggetto normale nel ruolo di condition variable
- I due thread sono rappresentate dalle classi Notifier e Worker
  - Notifier in realtà non è un thread direttamente, ma indirettamente tramite Swing
- Worker è in attesa di notifiche sull'oggetto usato come trigger, mediante una wait
- Notifier crea una GUI con un pulsante, premendo il quale viene invocata

Notifier

```
package modulo2f;
import javax.swing.*;
import java.awt.event.*;

public class Notifier extends JFrame implements ActionListener{
  private JButton button;
  private Object  trigger;

  public Notifier(Object synch){
    trigger = synch;
    setTitle("Trigger");
    setSize(150,60);
    setResizable(false);
    addWindowListener(new WindowAdapter(){
             public void windowClosing(WindowEvent ev){ System.exit(-1);}
             public void windowClosed(WindowEvent ev){System.exit(-1);}});
    button = new JButton("notify");
    JPanel p = new JPanel();
    p.add(button);
    getContentPane().add(p);
    button.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ev){
    synchronized (trigger){
      trigger.notify();
    }
  }
}
```

Worker ‘triggerato’

```
package modulo2f;

public class Worker extends Thread  {
  public Object trigger;
  public Worker(Object synch){
    trigger = synch;
  }
  public void run(){
    log(”waiting notification.");
    try {
      synchronized(trigger){
        trigger.wait();
      }
      log("notified.");
      } catch (Exception ex){
      }
    }

  private void log(String msg){
    System.out.println("[WORKER "+this+"] "+msg);
  }
}
```

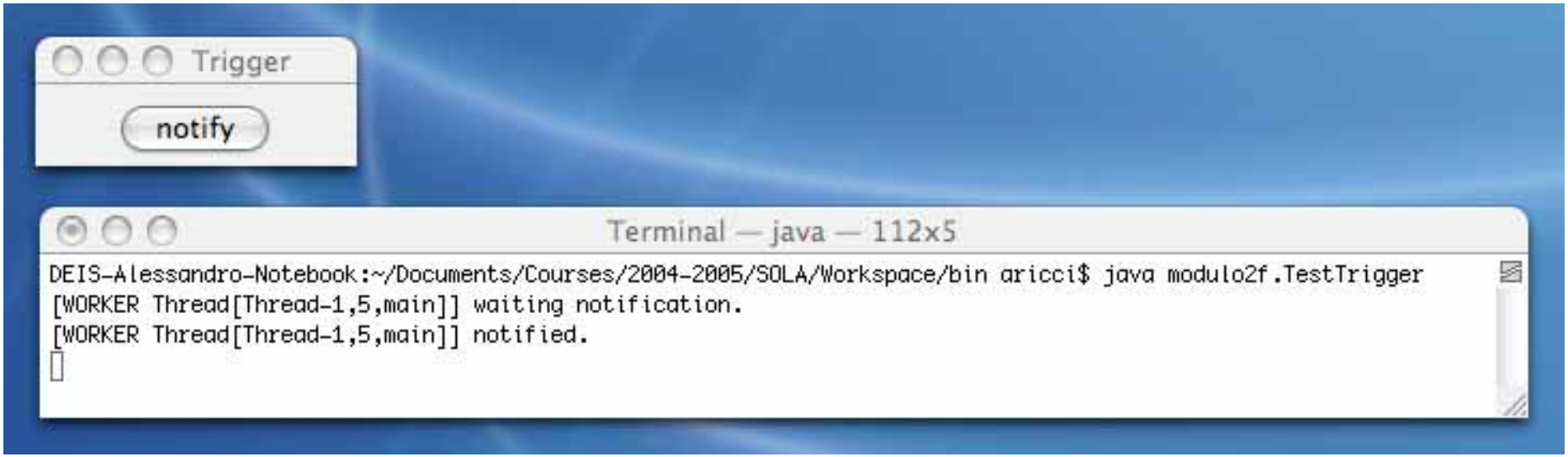
Test #1: Un notifier e un worker

- Nel primo test si considera un notifier e un worker

```
package modulo2f;

public class TestTrigger {
    public static void main(String[] args) {
        Object trigger = new Object();
        Notifier notifier = new Notifier(trigger);
        Worker  worker  = new Worker(trigger);
        notifier.show();
        worker.start();
    }
}
```

Test #1: risultato

- Lanciando l'applicazione, alla pressione del pulsante trigger viene visualizzato in standard output il messaggio dal Worker
- Da notare che la cosa non succede ripremendo il pulsante: questo perché il worker alla notifica esce.



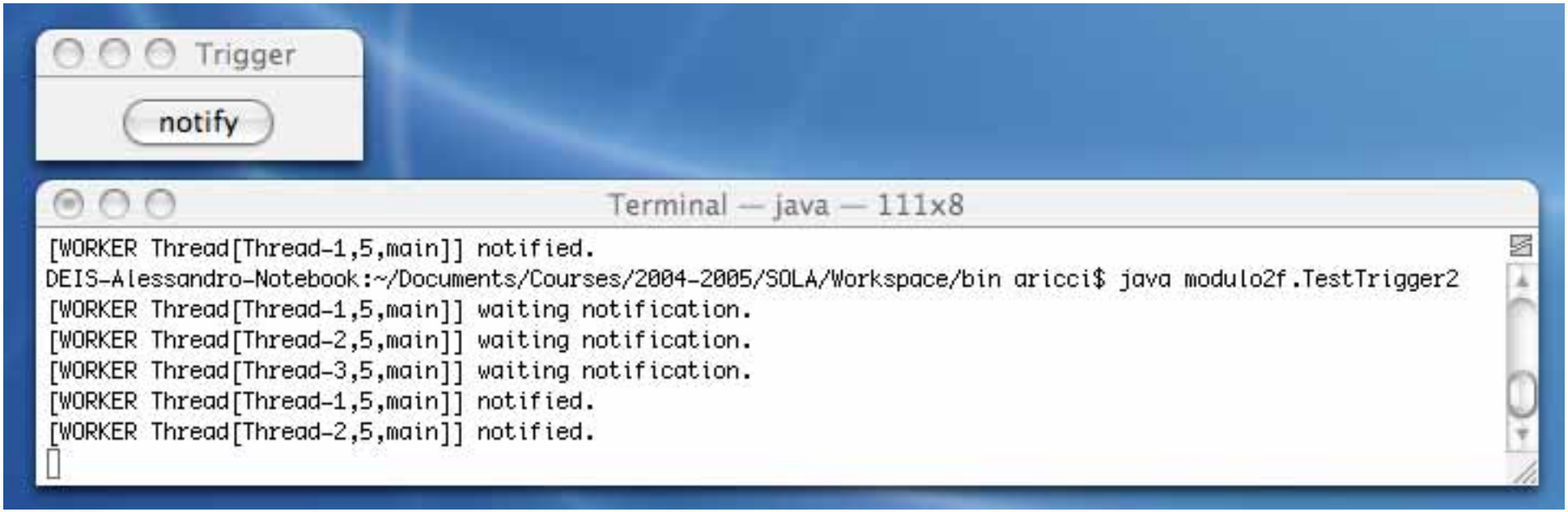
Test #2: Un notifier e N worker

- Nel secondo test si considera un notifier e tre worker

```
package modulo2f;

public class TestTrigger2 {
    public static void main(String[] args) {
        Object trigger = new Object();
        Notifier notifier = new Notifier(trigger);
       Worker  workerA  = new Worker(trigger);
       Worker  workerB  = new Worker(trigger);
       Worker  workerC  = new Worker(trigger);
        notifier.show();
         workerA.start();
         workerB.start();
         workerC.start();
    }
}
```

Test #2: risultato

- Lanciando l'applicazione, alla pressione del pulsante trigger viene visualizzato in standard output il messaggio da UNO dei worker, uno qualsiasi appartenente al wait set dell'oggetto trigger
- Premendo tre volte vengono prelevati dal wait set tutti e tre i worker



Variante con notifiche multiple

- Creiamo una variante del notifier che semplicemente faccia una notifyAll sul trigger invece di una semplice notify: in questo modo *tutti* i thread in attesa nei wait set vengono svegliati

NotifierAll

```
package modulo2f;
import javax.swing.*;
import java.awt.event.*;

public class Notifier extends JFrame implements ActionListener{
  private JButton button;
  private Object  trigger;

  public Notifier(Object synch){
    trigger = synch;
    setTitle("Trigger");
    setSize(150,60);
    setResizable(false);
    addWindowListener(new WindowAdapter(){
             public void windowClosing(WindowEvent ev){ System.exit(-1);}
             public void windowClosed(WindowEvent ev){System.exit(-1);}});
    button = new JButton("notify");
    JPanel p = new JPanel();
    p.add(button);
    getContentPane().add(p);
    button.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ev){
    synchronized (trigger){
      trigger.notifyAll();
    }
  }
}
```

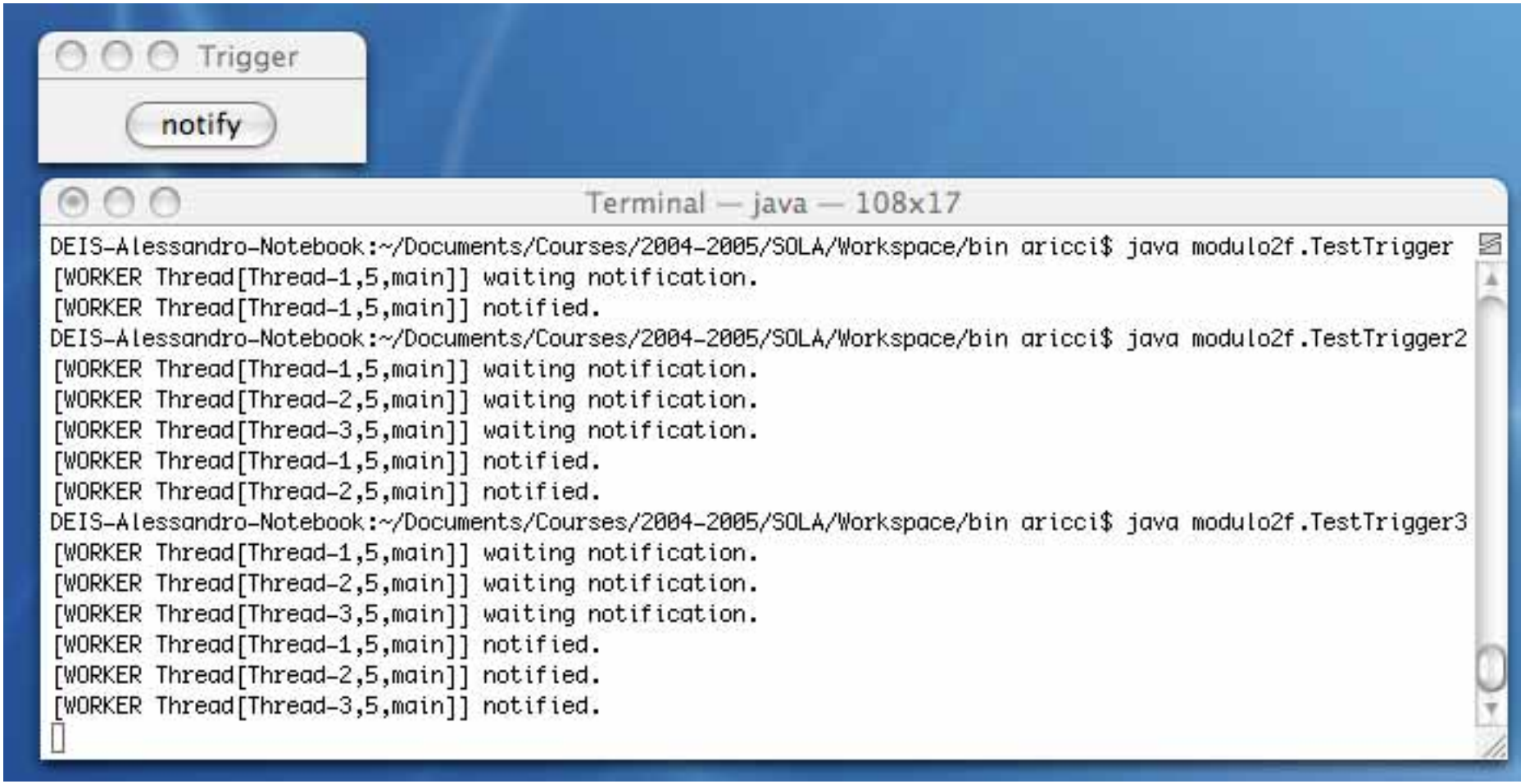
Test #3: Un notifier (all) e N worker

- Nel terzo test si considera il notifier che notifica tutti e tre worker

```
package modulo2f;

public class TestTrigger2 {
    public static void main(String[] args) {
        Object trigger = new Object();
        NotifierAll notifier = new NotifierAll(trigger);
       Worker  workerA  = new Worker(trigger);
       Worker  workerB  = new Worker(trigger);
       Worker  workerC  = new Worker(trigger);
        notifier.show();
         workerA.start();
         workerB.start();
         workerC.start();
    }
}
```

Test #3: risultato

- Lanciando l'applicazione, alla pressione del pulsante trigger viene visualizzato in standard output il messaggio da parte di tutti e tre i worker, notificati in un sol colpo dal notifier



Wait e rilascio del lock

- L'esecuzione del metodo wait **comporta il rilascio del lock**: ciò è fondamentale per permette ad altri thread in attesa e non di poter invocare metodi synchronized dell'oggetto stesso o porzioni di codice synchronzed sull'oggetto stesso.

Riassumendo...

- L'esecuzione del metodo wait da parte di uno thread P su un oggetto, comporta
  - il rilascio del lock da parte di P
  - cambiamento dello stato di P da runnable a blocked
  - inserimento di P nel wait set dell'oggetto
- L'esecuzione del metodo notify da parte di uno thread P su un oggetto, comporta
  - il prelevamento (*pick*) arbitrario di un thread T del wait set
  - l'inserimento del thread T nell'entry set
  - cambiamento dello stato di T da blocked a runnable - T quindi diviene elegibile di competere con gli altri thread per il lock dell'oggetto

Semafori in Java

- E' semplice realizzare un semaforo in Java sfruttando i meccanismi di base visti
  - Metodi synchronized per mutua esclusione
  - wait / notify per sospendere e riattivare thread in attesa (evitando attese attive)

```
package modulo2f;
public class Semaphore extends ISemaphore{
    private int value;

    public Semaphore(){
        value = 0;
    }

    public Semaphore(int value){
        this.value = value;
    }

    public synchronized void acquire(){
        if (value==0){
            try {
                wait();
            } catch (Exception ex){}
        }
        value--;
    }

    public synchronized void release(){
        value++;
        if (value>0){
            notify();
        }
    }
}
```

Produttori e consumatori con BoundedBuffer

- Mediante synchronized e wait/notify è possibile realizzare in modo agile il BoundedBuffer nel problema producers-consumers, senza utilizzare semafori e condition variables
  - la mutua esclusione si realizza mediante synchronized
  - le attese attive si evitano mediante wait e notify

BoundedBufferSynchronized (1/2)

```
package modulo2f;


public class BoundedBufferSynchronized implements IBoundedBuffer {

    private int in; // points to the next free position
    private int out; // points to the next full position
    private int count;
    private Object[] buffer;

    public BoundedBufferSynchronized(int size){
        in = 0;
        out = 0;
        count = 0;
        buffer = new Object[size];
    }

    private boolean isEmpty(){
        return count == 0;
    }

    private boolean isFull(){
        return count == buffer.length;
    }
    ...
```

BoundedBufferSynchronized (2/2)

```
    ...
    public synchronized void insert(Object item){
        while (isFull()){
            try {
                wait();
            } catch (Exception ex){}
        }
        count++;
        buffer[in] = item;
        in = (in + 1) % buffer.length;
        notify();
    }

    public synchronized Object remove(){
        while (isEmpty()){
            try {
                wait();
            } catch (Exception ex){}
        }
        Object item = buffer[out];
        out = (out + 1) % buffer.length;
        count--;
        notify();
        return item;
    }
}
```

Test del nuovo buffer

- (Il codice dei producers e consumers è identico)

```
package modulo2f;

public class TestProducerConsumer {

    public static void main(String[] args) {
        IBoundedBuffer buffer = new BoundedBufferSynchronized(20);
        Producer agentProd = new Producer(buffer,5000);
        Consumer agentCons = new Consumer(buffer);
        agentProd.start();
        agentCons.start();
    }
}
```

Interruzioni InterruptedException

- Il metodo wait può generare eccezioni del tipo **InterruptedException**.
- Tale eccezione è scatenata quando viene settato lo stato di interruzione (**interrupt state**) del thread che ha invocato la wait. Ciò avviene invocando il metodo **interrupt** direttamente sul thread in questione.
- Un thread bloccato su un oggetto con una wait viene di conseguenza sbloccato, e viene eseguito blocco catch.

```
try {
  wait();
} catch (InterruptedException ex){
  /* manage exception */
}
```

Timed wait

- Il metodo wait è fornito anche nella variante timed, in cui si specifica un tempo massimo di attesa in millisecondi, dopo il quale - se nessun thread ha ancora eseguito una notify - il thread in attesa (che ha eseguito la wait) viene sbloccato, con la generazione di una Interruzione InterruptException