Programmazione Java – Threads

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

- Cosa è: Processo
- Cosa è: Thread
- Thread in Java
- Ciclo di vita
- Scheduling
- Metodi: yield, sleep, interrupt, join
- Condivisione di risorse: synchronized, metodi wait, notify, notifyAll
- Esempio: Knock-Knock client-server

Cosa è: Processo

» Ambiente di esecuzione *self-contained* con un proprio insieme di risorse

> Memoria

> Dati: Stack e heap

> Codice

» Processi comunicano tra di loro per realizzare compiti complessi (Inter-process comunication, socket)

Cosa è: Thread

» Detto anche processo leggero, è una unità di esecuzione che consiste

> Stack

> Un insieme di registri

» Condivide con altri thread la sezione codice, la sezione dati (heap), e le risorse che servono per la loro esecuzione

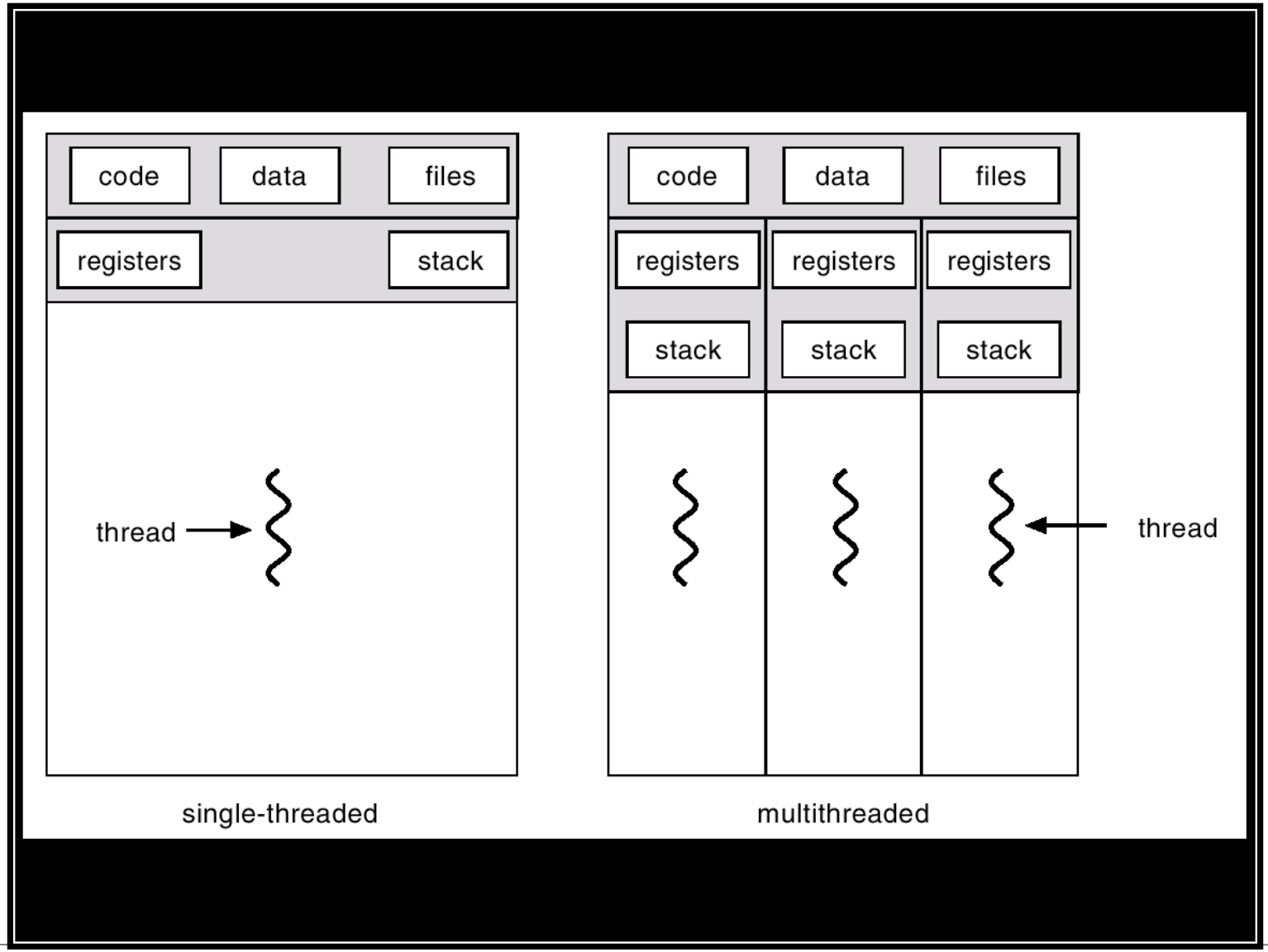
» Rendono più efficiente l'esecuzione di attività che condividono lo stesso codice

» Context-switch è più veloce rispetto a quello dei processi

» I thread non sono tra loro indipendenti perché condividono codice e dati

» E' necessario che le operazioni non generino conflitti tra i diversi thread di un task

Cosa è: Thread



Thread in Java

» Tutti i programmi Java comprendono almeno un thread

» Un programma costituito solo dal metodo main viene eseguito come un singolo thread

» Java fornisce

> Classi e interfacce che consentono di creare e manipolare thread aggiuntivi nel programma

> Meccanismi nel linguaggio per la sincronizzazione delle risorse

Thread in Java

» Esistono due modi per implementare in Java un thread

> Definire una classe estendola dalla classe `java.lang.Thread`

> Definire una classe implementando l'interfaccia `java.lang.Runnable`

(Vedi SimpleThread.java)

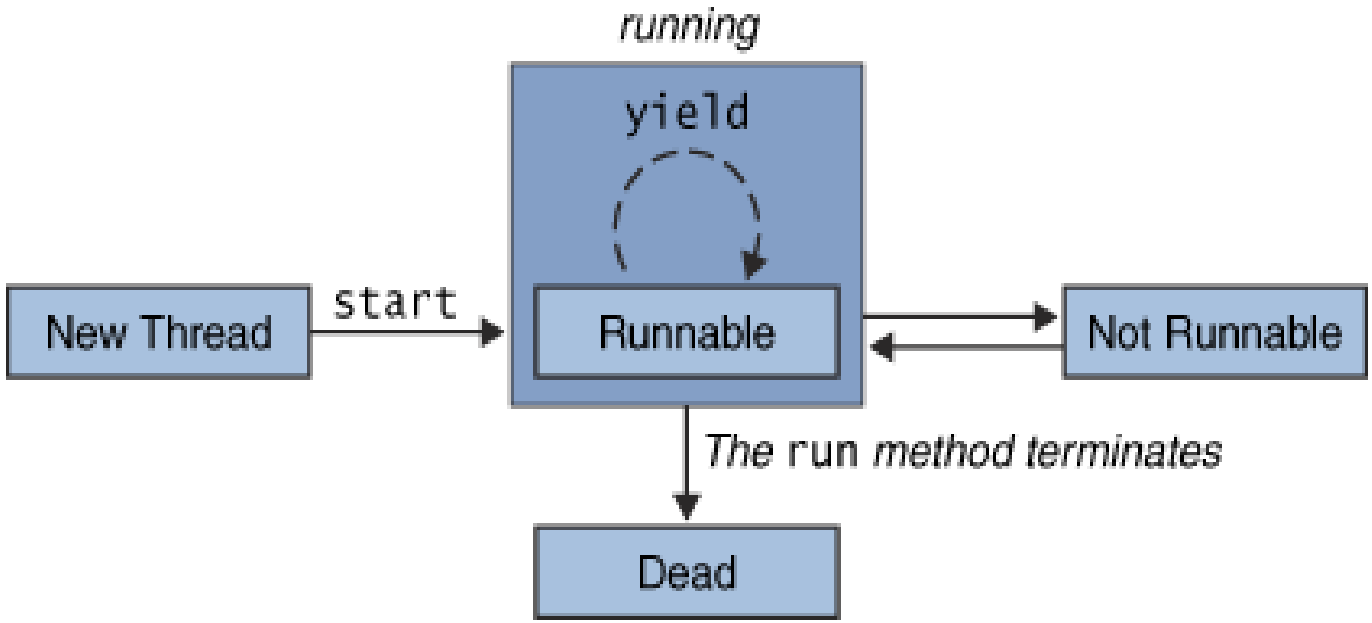
Esempio


```
public class SimpleThread extends Thread {

  private int countDown = 5;
  private static int threadCount = 0;
  public SimpleThread() {
    super("" + ++threadCount); // Store the thread name
    start();
  }
  public String toString() {
    return "#" + getName() + ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread();

  }
}
```

Esempio


```
public class SimpleThread implements Runnable {

  private int countDown = 5;
  private static int threadCount = 0;
  private String name;

  public SimpleThread() {
    this.name = "# SimpleThread(" + ++threadCount + ")";
  }

  public String toString() {
    return name + ": " + countDown;
  }

  public void run() {
    while (true) {
      System.out.println(this);
      if (--countDown == 0 )
        return;
    }
  }

  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      new Thread(new SimpleThread()).start();

    }
  }
}
```

Ciclo di vita di un thread (1)



Ciclo di vita di un thread (2)


```
Thread clockThread = new Thread(this, "Clock"); //this oggetto che implementa Runnable
clockThread.start();
```

- **New Thread**
  - Thread in tale stato è un oggetto vuoto ovvero nessuna risorsa di sistema è stata ancora allocata
  - Unico metodo invocabile è `start`
  - L'invocazione di qualsiasi altro metodo causa l'eccezione `IllegalThreadStateException`
- **Runnable**
  - Metodo `start` crea le risorse di sistema necessarie per eseguire il thread e invoca il metodo `run()`
- **Not Runnable:** E' in tale stato a seguito di uno dei seguenti eventi
  - Metodo `sleep` invocato
  - Viene invocato il metodo `wait` al verificarsi di una determinata condizione
  - Thread in blocco su una operazione di I/O
- **Dead**
  - Metodo usuale è il termine dell'esecuzione del metodo `run`
  - All'interno della classe Thread esiste il metodo `stop` **NON UTILIZZARE**

Testing stato dei Thread

» In Java 5 è stato introdotto il metodo `getState` all'interno della classe `Thread`. Il valore ritornato è

> NEW

> RUNNABLE

> BLOCKED

> WAITING

> TIMED_WAITING

> TERMINATED

» Inoltre, esiste (prima della versione 5) il metodo `isAlive`

> Ritorna `true` se il thread è started ma non è stopped (stato Runnable o Not Runnable)

> Ritorna `false` se il thread è negli stati New Thread o Dead

> Non è possibile differenziare tra i vari stati

Scheduling (1)

» Il sistema di run-time di Java supporta un semplice algoritmo di scheduling chiamato a *priorità-fissa*

» L'algoritmo schedula i thread con lo stato Runnable sulla base della loro priorità

» Quando un thread viene creato eredita la priorità del thread che lo sta creando. E' possibile modificare la priorità utiilzzando il metodo `setPriority` (costanti MIN_PRIORITY e MAX_PRIORITY definite all'interno della classe Thread)

» In un dato istante di tempo quando deve essere determinato il thread da eseguire, il sistema di run-time sceglie il thread con stato Runnable con priorità più alta

» Soltanto quando il thread finisce, oppure viene invocato il metodo `yield` oppure diventa Not Runnable un thread con priorità più bassa viene eseguito.

» Se vi sono thread con la stessa priorità lo scheduler ne sceglie arbitrariamente uno

Scheduling (2)

» Il thread scelto viene eseguito finché una delle seguenti condizioni si verificano

> Un thread con priorità più alta diventa Runnable

> Viene invocato il metodo `yield` oppure termina il metodo run

> Nei sistemi che supportano il time-slicing, lo slot di tempo del thread è terminato

» Lo scheduler è anche *preemptive* ovvero che se in un determinato istante di tempo un thread con priorità più alta diventa Runnable allora il sistema di runtime sceglie quello con priorità più alta (il nuovo thread si dice che *preempt* gli altri thread)

» **Nota**
In un determinato istante di tempo il thread con priorità più alta viene eseguito. Tuttavia questo non è garantito ovvero lo scheduler potrebbe scegliere un thread con priorità più bassa per eliminare starvation. Utilizzare la priorità soltanto per ragioni di eventuale efficienza e non di correttezza

Scheduling (3)


```
public class SelfishRunner extends Thread {

  private int tick = 1;
  private int num = 0;

  public SelfishRunner(int num) {
    this.num = num;
  }

  public void run() {
     while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.format("Thread #%d, tick = %d%n", num, tick);
        }
    }
  }
}
```

Scheduling (4)


```
public class RaceTest {

    private final static int NUMRUNNERS = 2;

    public static void main(String[] args) {
        SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];

        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new SelfishRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++)
            runners[i].start();
    }
}
```

yield()

» Serve per dare un *suggerimento* allo scheduler in modo che possa assegnare la CPU ad un altro thread

```
public class YieldingThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public YieldingThread() {
    super("" + ++threadCount);
    start();
  }
  public String toString() {
    return "#" + getName() + ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
      yield();
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new YieldingThread();
  }
}
```

sleep() e interrupt()

» Si può mettere un thread in uno stato d'attesa con il metodo statico `sleep` che prende il numero di millisecondi da aspettare

> Un oggetto può ottenere il suo thread d'esecuzione usando il metodo `Thread.currentThread()`

» Se un thread è in stato d'attesa, lo si può svegliare chiamando il metodo `interrupt`

> Tale metodo genera un'eccezione `InterruptedException`. Per questo è necessario ogni volta che si fa uno sleep prevedere l'eccezione `InterruptedException`

join()

» Un thread puo' invocare il metodo `join()` di un altro thread per attendere la terminazione di quest'ultimo prima di proseguire con l'esecuzione

» Se un thread chiama `t.join()` su un altro thread `t`, il thread chiamante viene sospeso finché il thread t non ha terminato l'esecuzione

» E' possibile utilizzare un'altra versione di `join()` che consente di specificare un tempo limite entro il quale attendere la terminazione (espresso in millisecondi o in millisecondi e nanosecondi). Se il thread sul quale è invocata l'operazione non termina entro il periodo di tempo fissato il metodo restituisce il controllo al chiamante

» `join()` puo' essere interrotta chiamando il metodo `interrupt()` sul thread chiamante quindi è necessario racchiudere la chiamata join dentro un blocco `try/catch`

Esempio 1


```
public class SleepingThread extends Thread {

  private int countDown = 5;
  private static int threadCount = 0;

  public SleepingThread() {
    super("" + ++threadCount);
    start();
  }

  public String toString() {
    return "#" + getName() + ": " + countDown;
  }

  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
      try {
        sleep(1000);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    for(int i = 0; i < 5; i++)
      new SleepingThread();
  }
}
```

Esempio 2


```
public class SleepingThread extends Thread {

  private int countDown = 5;
  private static int threadCount = 0;

  public SleepingThread() {
    super("" + ++threadCount);
    start();
  }

  public String toString() {
    return "#" + getName() + ": " + countDown;
  }

  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
      try {
        sleep(1000);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    for(int i = 0; i < 5; i++)
      new SleepingThread().join();
  }
}
```

Esempio 3


```
class Sleeper extends Thread {

  private int duration;

  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }

  public void run() {
    try {
      sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(getName() + " was interrupted. " +
        "isInterrupted(): " + isInterrupted());
      return;
    }
    System.out.println(getName() + " has awakened");
  }
}
```

Esempio 3


```
class Joiner extends Thread {

  private Sleeper sleeper;

  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }

  public void run() {
   try {
      sleeper.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println(getName() + " join completed");
  }
}
```

Esempio 3


```
public class Joining {

  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    Joiner
      dopey = new Joiner("Dopey", sleepy),
      doc = new Joiner("Doc", grumpy);
    grumpy.interrupt();
  }

}
```

Condividere risorse

» Nel caso di multi-threading potrebbe capitare di avere più thread che tentano di utilizzare nello stesso momento la stessa risorsa

» Tale situazione di programmazione è detta scenario *produttore-consumatore* ovvero un produttore (thread) genera uno stream di dati che un consumatore utilizza

» Esempi

> Thread che legge caratteri dalla tastiera che li inserisce in una coda di eventi che un thread consumatore legge tali eventi

> Thread scrive dati su un file e un altro thread legge dal file

Esempio 1 (1)


```
class Watcher extends Thread {

  private AlwaysEven ae;
  public Watcher(AlwaysEven ae) {
    this.ae = ae;
  }

  public void run() {
    while(true) {
      int val = ae.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }
}

public class AlwaysEven {
  private int i;
  public void next() { i++; i++; }
  public int getValue() { return i; }

  public static void main(String[] args) {
    final AlwaysEven ae = new AlwaysEven();
    Watcher w = new Watcher(ae);
    w.start();
    while(true)
      ae.next();
  }
}
```

Esempio 1 (2)


```
class Watcher extends Thread {

  private AlwaysEven ae;
  public Watcher(AlwaysEven ae) {
    this.ae = ae;
  }

  public void run() {
    while(true) {
      int val = ae.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }
}

public class AlwaysEven {
  private int i;
  public synchronized void next() { i++; i++; }
  public int getValue() { return i; }

  public static void main(String[] args) {
    final AlwaysEven ae = new AlwaysEven();
    Watcher w = new Watcher(ae);
    w.start();
    while(true)
      ae.next();
  }
}
```

Esempio 1 (3)


```
class Watcher extends Thread {

  private AlwaysEven ae;
  public Watcher(AlwaysEven ae) {
    this.ae = ae;
  }

  public void run() {
    while(true) {
      int val = ae.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }
}

public class AlwaysEven {
  private int i;
  public synchronized void next() { i++; i++; }
  public synchronized int getValue() { return i; }

  public static void main(String[] args) {
    final AlwaysEven ae = new AlwaysEven();
    Watcher w = new Watcher(ae);
    w.start();
    while(true)
      ae.next();
  }
}
```

Esempio 2 (1)

» `Producer`

> Genera un intero tra 0 e 9 (compreso) e lo memorizza all'interno dell'oggetto `CubbyHole`

> Per rendere l'esempio più interessante si addormenta (sleep) per un tempo random tra 0 e 100 millisecondi prima di ripetere il ciclo di generazione del numero

» `Consumer`

> Consuma tutti gli interi dall'oggetto `CubbyHole` (è lo stesso oggetto del `Producer`) non appena diventa disponibile

Esempio 2 (2)


```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(number, i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Esempio 2 (3)


```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get(number);
        }
    }
}
```

Esempio 2 (4)


```
public class CubbyHole {

    private int contents;
    private boolean available = false;

    public synchronized int get(int who) {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        System.out.format("Consumer %d got: %d%n", who, contents);
        notifyAll();
        return contents;
    }

    public synchronized void put(int who, int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.format("Producer %d put: %d%n", who, contents);
        notifyAll();
    }
}
```

Esempio 2 (5)


```
public class ProducerConsumerTest {


    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

wait()

- Si trova all'interno della classe `Object` e non di `Thread`
- Causa il blocco del thread corrente (stato Not Runnable) fino a quando un altro thread invoca il metodo `notify` o `notifyAll` per quell'oggetto
- Il thread corrente deve avere un lock (semaforo rosso) sull'oggetto ovvero il wait deve essere invocato all'interno di un metodo (o di un blocco) sincronizzato altrimenti viene sollevata un'eccezione `IllegalMonitorStateException`
- L'invocazione di `wait` rilascia il lock sull'oggetto permettendo ad un altro thread di acquisire il lock su quell'oggetto
- Altri metodi
  - `wait(long timeout)`: come `wait()` soltanto che passati timeout millisecondi il thread viene risvegliato
  - `wait(long timeout, int nanos)`: Si possono specificare anche i nanosecondi

notify, notifyAll

» Si trovano all'interno della classe `Object` e non di `Thread`

» `notify`

> Sveglia un singolo thread che è in attesa sul monitor dell'oggetto

> Se vi sono diversi thread in waiting sull'oggetto ne viene scelto uno a caso

» Una volta svegliato il thread competerà con gli altri thread nel riprendere la risorsa condivisa

» `notifyAll`

> E' come notify soltanto che sveglia tutti gli eventuali thread in wait

Riacquisto di un lock

» Un thread puo' invocare un metodo sincronizzato su un oggetto per il quale è già mantenuto il lock riacquistando perciò il lock

» Tale caratteristi elimina la possibilità per un singolo thread di mettersi in wait per un lock che già ha

» Esempio

```
public class Reentrant {
    public synchronized void a() {
        b();
        System.out.format("Here I am, in a().%n");
    }
    public synchronized void b() {
        System.out.format("Here I am, in b().%n");
    }
}
```

» Java supporta il lock rientrante quindi l'invocazione di `b()` all'interno di `a()` non causa un deadlock

Esempio: Knock-Knock client-server (1)


```
import java.net.*;
import java.io.*;

public class KKMultiServer {

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        boolean listening = true;

        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(-1);
        }

        while (listening)
              new KKMultiServerThread(serverSocket.accept()).start();

        serverSocket.close();
    }
}
```

Esempio: Knock-Knock client-server (2)


```
import java.net.*;
import java.io.*;

public class KKMultiServerThread extends Thread {
    private Socket socket = null;

    public KKMultiServerThread(Socket socket) {
        super("KKMultiServerThread");
        this.socket = socket;
    }

    public void run() {

    try {
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String inputLine, outputLine;
        KnockKnockProtocol kkp = new KnockKnockProtocol();
        outputLine = kkp.processInput(null);
        out.println(outputLine);

        while ((inputLine = in.readLine()) != null) {
            outputLine = kkp.processInput(inputLine);
            out.println(outputLine);
            if (outputLine.equals("Bye"))
                break;
        }
        out.close();
        in.close();
        socket.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
    }
}
```

Esempio: Knock-Knock client-server (3)


```
import java.net.*;
import java.io.*;

public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;
    private static final int NUMJOKES = 5;
    private int state = WAITING;
    private int currentJoke = 0;
    private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" };
    private String[] answers = { "Turnip the heat, it's cold in here!", "I didn't know you could yodel!", "Bless you!",
                                 "Is there an owl in here?", "Is there an echo in here?" };
    public String processInput(String theInput) {
        String theOutput = null;

        if (state == WAITING) {
            theOutput = "Knock! Knock!";
            state = SENTKNOCKKNOCK;
        } else if (state == SENTKNOCKKNOCK) {
            if (theInput.equalsIgnoreCase("Who's there?")) {
                theOutput = clues[currentJoke];
                state = SENTCLUE;
            } else {
                theOutput = "You're supposed to say \"Who's there?\"! " +
                "Try again. Knock! Knock!";
            }
        } else if (state == SENTCLUE) {
            if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
                theOutput = answers[currentJoke] + " Want another? (y/n)";
                state = ANOTHER;
            } else {
                theOutput = "You're supposed to say \"" +
                clues[currentJoke] +
                " who?\"" +
                "! Try again. Knock! Knock!";
                state = SENTKNOCKKNOCK;
            }
        } else if (state == ANOTHER) {
            if (theInput.equalsIgnoreCase("y")) {
                theOutput = "Knock! Knock!";
                if (currentJoke == (NUMJOKES - 1))
                    currentJoke = 0;
                else
                    currentJoke++;
                state = SENTKNOCKKNOCK;
            } else {
                theOutput = "Bye.";
                state = WAITING;
            }
        }
        return theOutput;
    }
}
```

Esempio: Knock-Knock client-server (4)


```
import java.io.*;
import java.net.*;

public class KnockKnockClient {
    public static void main(String[] args) throws IOException {
        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;
        try {
            kkSocket = new Socket("localhost", 4444);
            out = new PrintWriter(kkSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: localhost.");
            System.exit(1);
        }
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        String fromServer;
        String fromUser;
        while ((fromServer = in.readLine()) != null) {
          System.out.println("Server: " + fromServer);
          if (fromServer.equals("Bye."))
              break;

          fromUser = stdIn.readLine();
          if (fromUser != null) {
              System.out.println("Client: " + fromUser);
              out.println(fromUser);
          }
        }
        out.close();
        in.close();
        stdIn.close();
        kkSocket.close();
    }
}
```