

PCAD
Programmazione Concorrente
Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

PROCESSI E THREAD in Java

Processi

- Un processo è un **programma** (di natura statica) in corso di **esecuzione** (di natura dinamica). La sua esecuzione necessita un ambiente :
 - spazio di indirizzamento dedicato
 - input and output (per es. gli standard)
- Più processi possono eseguirsi sulla stessa macchina in un modo quasi-simultaneo
 - Se il sistema rispetta il **tempo condiviso** o è **multi-task**
 - Questo vale anche se c'è un solo processore
 - Il sistema operativo è in carica di allocare le risorse (memoria, tempo processore, input/output etc)
 - Un processo **non ha accesso** allo spazio di indirizzamento degli altri
 - Abbiamo l'**illusione del parallelismo**

Processi in Java

- Java permette di manipolare dei processi
- Questi processi non sono gestiti dalla JVM (Java Virtual Machine)
- Sono gestiti dal sistema
- Non c'è quindi una nozione di processo dentro la JVM, un processo è un oggetto del sistema
- Tuttavia, in un programma Java, possiamo
 - dire al sistema di eseguire un processo
 - recuperare informazione su questo processo (come ad esempio i suoi input e output)

L'ambiente di esecuzione

- Per prima cosa, bisogna recuperare l'ambiente di esecuzione
- È disponibile tramite un oggetto della classe **java.lang.Runtime**
- Osservazioni:
 - esiste solo un oggetto di questa classe
 - non è possibile crearlo
- Per recuperarlo, si usa la funzione static **Runtime.getRuntime()**
- Esistono tante funzione per la classe **java.lang.Runtime** (cf. API), in particolare:
 - funzione per avviare un processo, sono funzione con la forma **Process exec(...)**

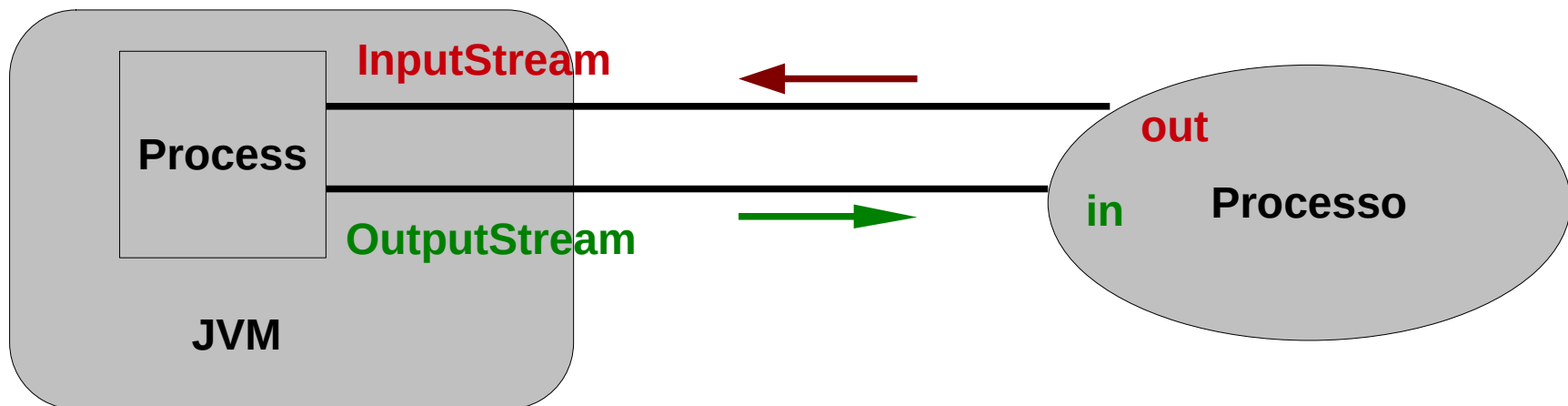
Processi in Java in pratica

- La JVM permette dunque di avviare processi esterni
- Sono rappresentati sotto la forma di oggetti **java.lang.Process**
- A cosa servono questi oggetti?
 - A comunicare con i processi esterni che rappresentano
 - A sincronizzarsi con la loro esecuzione (per esempio per aspettare la fine di un processo)
- Come creare questi processi?
 - Per esempio per avviare un processo che esegue ls -a

```
String[] cmd={"ls", "-a"};  
Process process = Runtime.getRuntime().exec(cmd);
```

Comunicare con i processi

- Vogliamo essere in grado di scambiare informazione con i processi
- Per esempio per leggere cosa stampano e farli stampare dalla JVM
- Per questo ci sono dei metodi che usano dei flussi
 - **InputStream** `getInputStream()`
 - **OutputStream** `getOutputStream()`
- **Warning:** i flussi sono da prendere in considerazione dal lato del programma Java



Input e Output in Java

- Gli input/output usano dei **flussi** (*streams*)
 - gli **Input stream** permettono di leggere dei dati
 - gli **Output stream** permettono di scrivere dei dati
- Esistono diverse classi di stream secondo i dati manipolati
 - per esempio :
 - **java.io.FileInputStream**
 - **sun.net.TelnetOutputStream**
- Tutti gli Input stream usano delle funzioni simili di lettura
- Tutti gli Output stream usano delle funzioni simili di scrittura
- Si usano dei filtri per facilitare la manipolazione dei dati
 - ad esempio per manipolare delle **String** piuttosto che dei **byte[]**

Output Stream

- La classe di base per gli Output Stream è:
 - **java.io.OutputStream** (abstract class)
- Dispone delle funzioni:
 - **public abstract void write(int b) throws IOException**
 - aspetta come input un intero b fra 0 e 255 et lo scrive sul flusso
 - **public void write(byte[] data) throws IOException**
 - **public void write(byte[] data, int offset, int length) throws IOException**
 - **public void flush() throws IOException**
 - Utile per svuotare il buffer associato allo stream
 - **public void close() throws IOException**

Dai dati ai bytes



- la classe **java.io.OutputStreamWriter**
- estende la abstract classe **Writer**
- Si da al costruttore un **OutputStream** (**OutputStreamWriter(OutputStream out))**
- Esempi di funzioni di questa classe
 - **public void close() throws IOException**
 - **public void flush() throws IOException**
 - **public void write(int c) throws IOException**
 - **public void write(char[] cbuf, int off, int len) throws IOException**
 - **public void write(String str, int off, int len) throws IOException**

Dai dati ai bytes



- la abstract class **java.io.PrintWriter** è più piacevole da usare
- per il costruttore: **PrintWriter(Writer out)**
- Esempi di funzioni di questa classe
 - **public void close() throws IOException**
 - **public void flush() throws IOException**
 - **public void print(char c)**
 - **public void print(int i)**
 - **public void print(String s)**
 - **public void println(String s)**
 - **public void print(Object o),....**

Per riassumere

- **OutputStream** : per scrivere dei bytes
- **OutputStreamWriter** : per scrivere dei caratteri
- **PrintWriter** : per scrivere ogni tipo di dati
- Per esempio: se **out** è un oggetto della classe **OutputStream**

```
PrintWriter pw=new PrintWriter(new OutputStreamWriter(out));  
pw.println("Hello");
```

- **close()** sul **PrintWriter** chiude l'Output Stream corrispondente

Input Stream

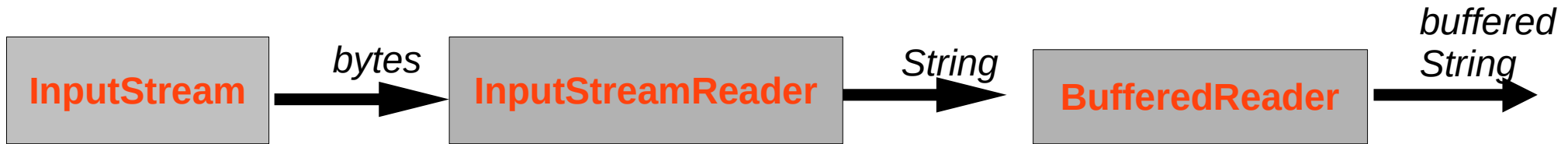
- La classe di base per gli Input Stream è:
 - **java.io.InputStream** (abstract class)
- Dispone delle funzione:
 - **public abstract int read() throws IOException**
 - legge un intero e lo rimanda
 - **public int read(byte[] input) throws IOException**
 - **public int read(byte[] input, int offset, int length) throws IOException**
 - **public void close() throws IOException**

Dai bytes ai String



- la classe **java.io.InputStreamReader**
- estende la abstract classe **Reader**
- Si da al costruttore un **InputStream (InputStreamReader(InputStream in))**
- Esempi di funzione di questa classe
 - **public void close() throws IOException**
 - **public int read() throws IOException**
 - **public int read(char[] cbuf,int offset,int length) throws IOException**

Dai bytes ai String



- la abstract class **java.io.BufferedReader** è più piacevole da usare
- per il costruttore: **PrintWriterBufferedReader(Reader in)**
- Esempi di funzione di questa classe
- **public void close() throws IOException**
- **public int read() throws IOException**
- **public int read(char[] cbuf,int offset,int length) throws IOException**
- **public String readLine()**

Per riassumere

- **InputStream** : per legger dei bytes
- **InputStreamReader** : per leggere dei caratteri
- **BufferedReader** : per leggere dei caratteri tramite un buffer
- Per esempio: se **in** è un oggetto della classe **InputStream**

```
BufferedReader bf=new BufferedReader(new InputStreamReader(in));  
String st=bf.readLine();
```

- **close()** sul **BufferedReader** chiude l'Input Stream corrispondente
- Le metodi di lettura aspettano dati da leggere se il flusso non è chiuso

Sincronizzarsi con un processo

- Esistono due funzioni per sincronizzarsi con un processo (disponibile nella classe **java.lang.Process**)
 - 1) **int waitFor() throws InterruptedException**
 - aspetta la fine dell'esecuzione di un processo
 - ritorna il valore ritornato dal processo
 - 2) **int exitValue()**
 - ritorna il valore ritornato dal processo
- È importante aspettare la fine dell'esecuzione del processo
- Se il programma Java finisce prima, non possiamo più recuperare informazione

Esempio

```
public class ExecLS{
    public static void main(String[] args){
        try{
            String[] cmd={"ls", "-la"};
            Process pr=Runtime.getRuntime().exec(cmd);
            BufferedReader bf=new BufferedReader(new
InputStreamReader(pr.getInputStream()));
            String line=bf.readLine();
            while(line!=null){
                System.out.println(line);
                line=bf.readLine();
            }
            bf.close();
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Problemi dei processi



Ma come passiamo delle
variabile ai processi ?

- Per fare programmazione concorrente non useremmo **exec(...)** della classe **Runtime**
- Come in C, piuttosto che usare processi del sistema useremmo dei **threads** che esistono nella JVM

Threads vs Processi

- **Un thread** è un filo d'esecuzione dentro un programma
- Il thread è eseguito da un processo
- Un processo può gestire più thread
 - Si parla di processo **multi-thread**
 - Al minimo c'è uno thread
- Ogni filo d'esecuzione è diverso e ha come attribuiti:
 - Un **puntatore di esecuzione** o **PC (Program Counter)**
 - Una stack d'esecuzione (**stack**)
- Uno thread condivide tutto l'ambiente di esecuzione con gli altri threads
- **La JVM est multi-threaded** e offre al programmatore la possibilità di manipolare gli threads

Gli threads in java

- Per manipolare degli threads in java, useremmo due classe importante
 - L'interaccia **java.lang.Runnable**
 - Contiene una unica funzione da implementare
 - **void run()**
 - Questa funzione darà il codice da eseguire dallo thread
 - La classe **java.lang.Thread**
 - È lei che ci permette di manipolare gli thread
- Riassumendo, nella funzione **run()** si troverà il codice dello thread (la parte statica) et useremmo un oggetto Thread per gestire il filo d'esecuzione (la parte dinamica)

Collegamenti fra Thread et Runnable

- Ad ogni thread viene associato un oggetto implementando **Runnable**
 - Ad esempio uno dei costruttori in **java.lang.Thread** è
 - **public Thread(Runnable target)**
- Lo stesso oggetto implementando **Runnable** puo essere associato a più di uno thread
 - In questo ultimo caso, ogni thread esegue in un modo concorrente la funzione run() del oggetto associato

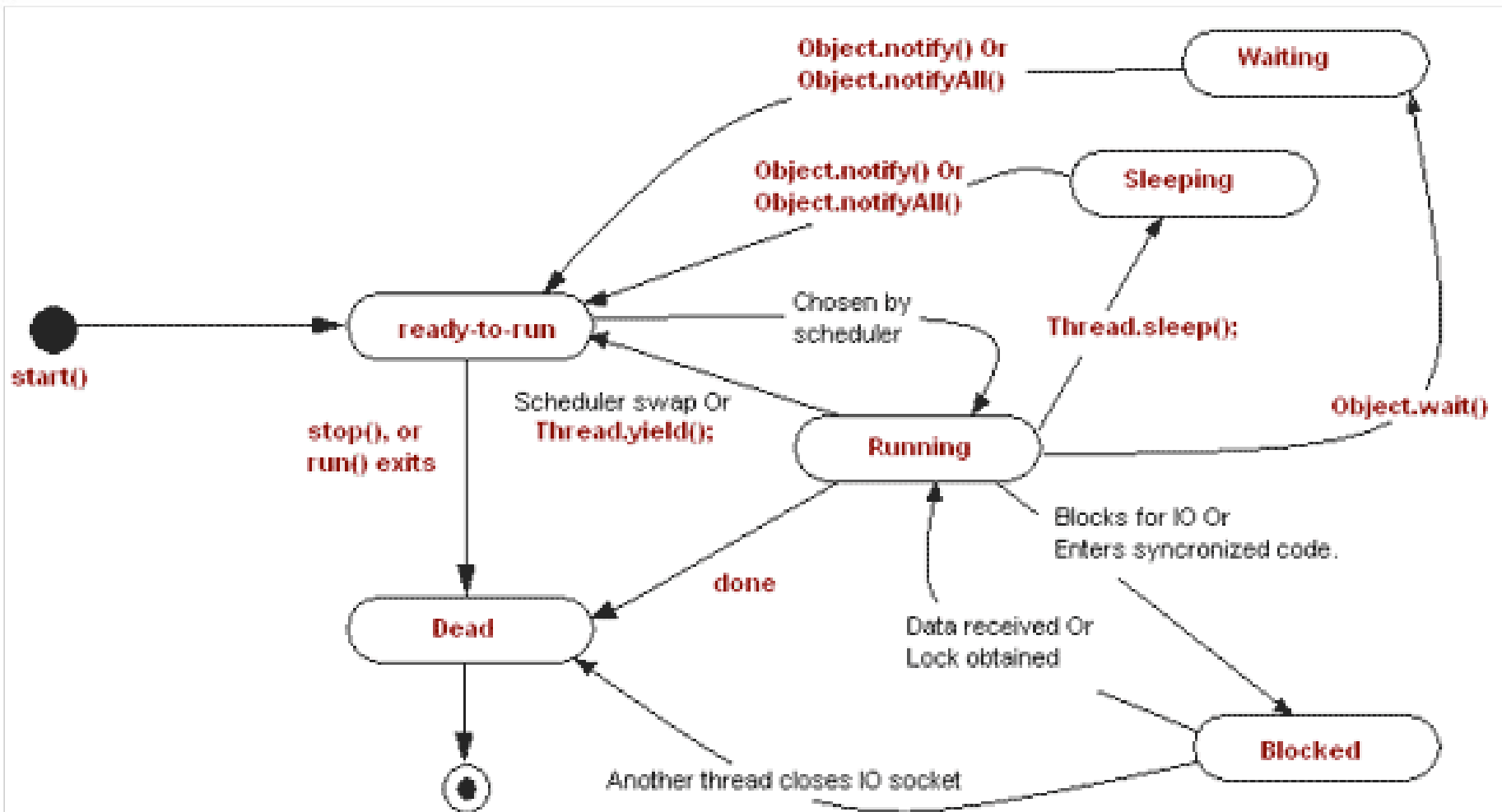
L'interfaccia `java.lang.Runnable`

- **`java.lang.Runnable`** è quindi una interfaccia con un singolo funzione da implementare
 - **`public void run()`**
- Quando uno thread inizierà la sua esecuzione
 - Comincerà per chiamare la funzione **`run()`** du **`Runnable`** associato
 - Finirà la sua esecuzione quando questa funzione **`run()`** terminerà
- **Warning:** per avviare uno thread, uno non chiama direttamente la funzione **`run()`** ma una altra funzione che si chiama **`start()`**

La classe `java.lang.Thread`

- Les Threads Java hanno diversi attributi:
 - **String name**: il nome dello thread
 - **long id**: l'identità dello thread
 - **int priority**: la sua priorità (gli threads sono ordinati secondo questa priorità)
 - **boolean daemon**: il suo modo di esecuzione (daemon o no, cf più lontano)
 - **Thread.state state**: il suo stato
 - **NEW, RUNNABLE, BLOCKED, WAITING, TERMINATED**,...
 - La sua stack per la quale si può solo vedere lo stato
 - il suo gruppo di thread
 - etc (cf la documentazione)
- In **java.lang.Thread**, abbiamo gli getters per questi attributi

Gli stati di uno thread



Gli stati di uno thread

- **Start**
 - Thread appena creato
 - Quando viene chiamato start() entra nello stato ready-to-run
- **Ready-to-run**
 - Pronto per essere eseguito
- **Running**
 - Il thread è effettivamente in esecuzione
 - Quando run() termina entra nello stato dead
 - **Lo scheduler fa passare da ready-to-run a running e vice-versa**
- **Dead**
 - Il thread può essere rimosso dal sistema
 - Quando run() termina oppure per eccezione non gestita

Gli stati di uno thread

- **Blocked**
 - Mentre è in running può entrare in questo stato
 - Ad esempio in attesa di un input
- **Sleeping**
 - Quando viene chiamata la funzione sleep(...)
 - Rientra nello stato ready-to-run quando il tempo associato allo sleep è passato
- **Waiting**
 - Verremo dopo a cosa corrisponde

Terminazione della JVM

- Spesso si dice che un programma termina quando si esce dal main:
 - Il programma non termina ma il processo associato che termina
 - Ma non basta uscire dal main, bisogna terminare la prima chiamata al main (se per esempio ci sono state chiamate ricorsive)
 - Bisogna aspettare che tutti gli thread che non sono daemon finiscano
 - Esiste almeno uno thread daemon
 - **Il garbage collector**
 - Spesso ce ne sono altri
 - Ad esempio lo thread responsabile per l'interfaccia grafica

Creazione e controllo degli thread

- Esistono più costruttori per creare thread
 - **Thread(Runnable target), Thread(Runnable target, String name)**
- Esistono più finzione per controllare l'esecuzione di uno thread
 - **void start()**
 - Permette di avviare lo thread
 - Questo chiamerà la funzione **run()** del **Runnable** associato
 - **void join()**
 - Permette di aspettare la fine di esecuzione dello thread
 - **void interrupt()**
 - metto lo stato dello thread a interrotto
 - non ha nessuno effetto diretto (permette solo allo thread di sapere che un altro thread desidera interromperlo)
 - **IMPORTANTE : Non ci sono funzioni per fermare uno thread**, bisogna a fare in modo che finisca la sua prima chiamata a run

Le funzione static di Thread

- **Thread currentThread()**
 - Per recuperare l'oggetto thread che esegue la riga
 - Utile per esempio per recuperare il nome
- **boolean isInterrupted()**
 - Per testare se lo thread ha ricevuto una richiesta di interruzione
- **void sleep(long ms)**
 - Per fare dormire lo thread in millisecondi (si tratta di un tempo minimo ma lo thread potrebbe dormire di più)
- **void yield()**
 - Lo thread rinuncia temporaneamente alla sua esecuzione e il suo stato diventa ready-to-run

Eempio Runnable

```
import java.lang.*;
import java.io.*;

public class ServiceTest implements Runnable {
    public void run(){
        try{
            while(true){
                Thread.sleep(1000);
                System.out.println("Hello"+Thread.currentThread().getName());
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Esempio

```
import java.lang.*;
import java.io.*;

public class TestThread {
    public static void main(String[] args){
        try{
            Thread t1=new Thread(new ServiceTest(),"Bob");
            Thread t2=new Thread(new ServiceTest(),"Alice");
            //t1.setDaemon(true);
            //t2.setDaemon(true);
            t1.start();
            t2.start();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

I problemi della concorrenza (1)

```
public class Counter{  
  
    private int val;  
  
    public Counter(){  
        val=0;  
    }  
  
    public int getVal(){  
        return val;  
    }  
  
    public void setVal(int v){  
        val=v;  
    }  
}
```


I problemi della concorrenza (2)

```
import java.io.*;
import java.lang.*;

public class CodeCounter implements Runnable{

    private Counter c;

    public CodeCounter(Counter _c){
        this.c=_c;
    }

    public void run(){
        for(int i=0; i<10000; i++){
            c.setVal(c.getVal()+1);
        }
    }
}
```

I problemi della concorrenza (3)

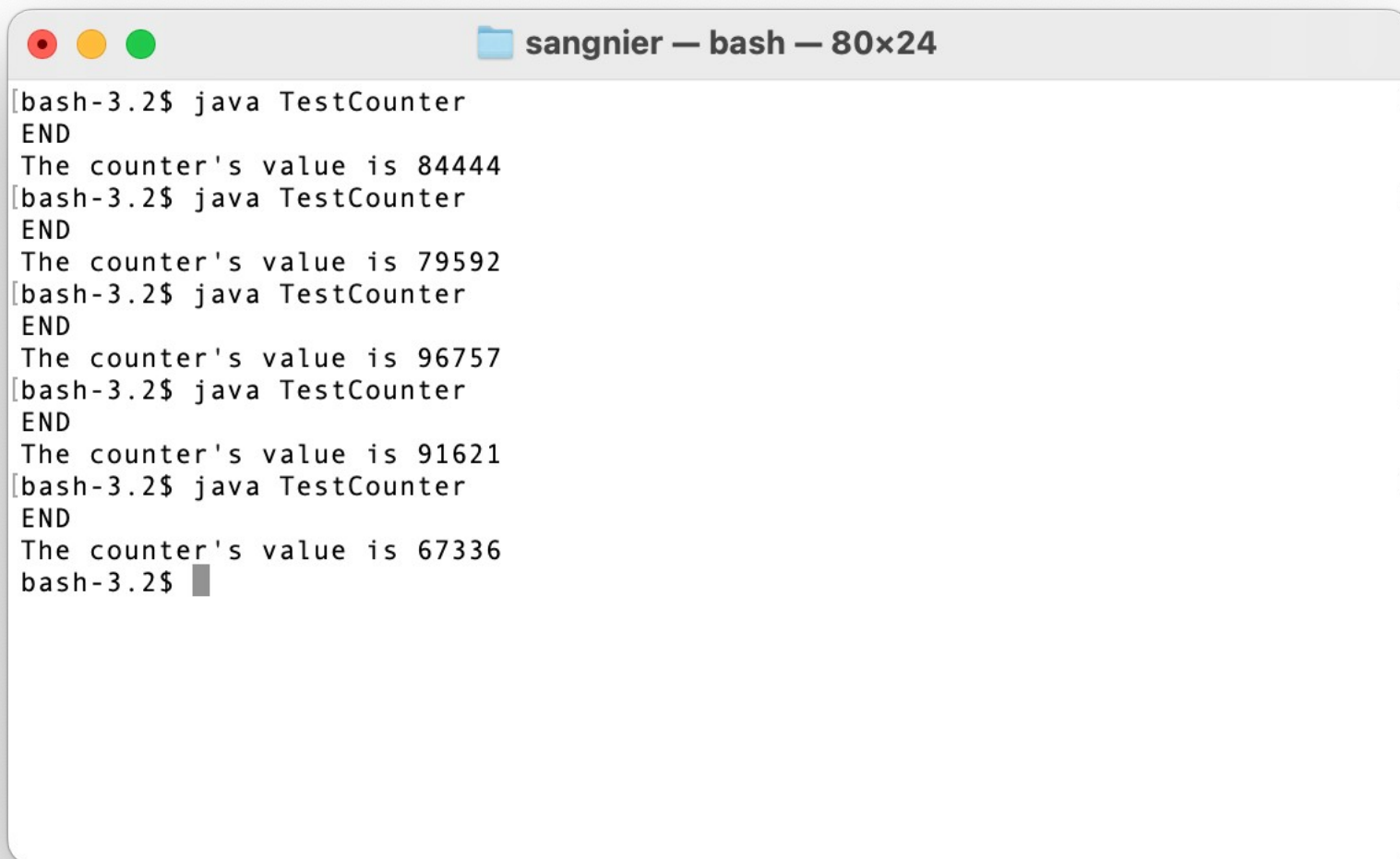
```
import java.lang.*;
import java.io.*;

public class TestCounter {
    public static void main(String[] args){
        try{
            Counter c=new Counter ();
            CodeCounter code=new CodeCounter(c);
            Thread []t=new Thread[20];
            for(int i=0; i<20; i++){
                t[i]=new Thread(code);
            }
            for(int i=0; i<20; i++){
                t[i].start();
            }
            for(int i=0; i<20; i++){
                t[i].join();
            }
            System.out.println("END");
            System.out.println("The counter's value is "+c.getVal());
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Da dove viene il problema

- L'operazione `c.setVal(c.getVal()+1);` **non è atomica** !!!
 - Più thread possono fare questa operazione in parallelo
- Scenario possibile
 - Thread 1 prende la valore del counter (per esempio 0)
 - Thread 2 prende la valore del counter (sempre 0)
 - Thread 1 mette la nuova valore nel counter (a 1)
 - Thread 2 mette la nuova valore nel counter (à 1)
 - A questo punto i due thread hanno aumentato il counter di 1 ma non si sono messi d'accordo su come farlo
- Il risultato può cambiare ad ogni esecuzione !!!

Risultati dell' esecuzione



```
sangnier — bash — 80x24
[bash-3.2$ java TestCounter
END
The counter's value is 84444
[bash-3.2$ java TestCounter
END
The counter's value is 79592
[bash-3.2$ java TestCounter
END
The counter's value is 96757
[bash-3.2$ java TestCounter
END
The counter's value is 91621
[bash-3.2$ java TestCounter
END
The counter's value is 67336
bash-3.2$
```

Nessuna di queste valore è uguale 200000 !!!!!

Come risolvere il problema

- Principe in programmazione concorrente
 - Non possiamo assumere nulla sullo scheduling degli thread
 - Ogni ordine è quindi possibile
- Bisogna quindi offrire delle garanzie sul codice
 - Si protegge la sequenze d'istruzione
 - Possiamo usare una variante dei lock

La parola chiave synchronized

- La parola chiave **synchronized** viene usata nel codice per proteggere l'accesso ad una parte del codice e garantire che al massimo uno thread eseguirà il codice
- Corrisponde ad un lock (**ogni oggetto Java ha il suo lock**)
- Due modi di usarla:
 - Si dichiara una funzione **synchronized**
 - **public synchronized int f(int a){...}**
 - Ogni chiamata **synchronized** dello stesso oggetto non può essere eseguita in parallelo
 - Oppure si controlla un pezzo del codice
 - **synchronized(object) {.../*codice a proteggere*/...}**
 - Si dà l'oggetto per il quale vogliamo usare il lock

Nel caso del counter

```
import java.io.*;
import java.lang.*;

public class CodeCounter implements Runnable{

    private Counter c;

    public CodeCounter(Counter _c){
        this.c=_c;
    }

    public void run(){
        for(int i=0; i<10000; i++){
            synchronized(c) {
                c.setVal(c.getVal()+1);
            }
        }
    }
}
```

Consigli

- Pensare bene a dove mettere gli synchronized
- Ricordarsi che i lock sono associati a degli oggetti
 - Due pezzi di codice usando delle funzione synchronized dello stesso oggetto non saranno eseguiti in parallelo
- Mettere in synchronized parte di codice che finiscono (altrimenti il lock rimane preso per sempre)
- Attenzione ai deadlocks !!!!
- **Osservazione:**
 - `synchronized int f(...) { ... }` è la stessa cosa di:
 - `int f(....) { synchronized(this){...}}`
 - Il lock usato è sempre il lock del oggetto associato

Reentrant synchronization

```
public synchronized int f(int x){  
    return x+g(x*x);  
}  
  
public synchronized int g(int x){  
    return v*5;  
}
```

- Quando lo thread ha il lock per f e chiama g, non viene bloccato
- Una volta ottenuto il lock su un oggetto, lo thread lo tiene anche se entra in altre parte synchronized

Producer/consumer

- Nel problem dei produttori/consumatori
 - I produttori scrivono in una variabile
 - I consumatori leggono le valori scritte
 - Ogni valore deve essere letta una unica volta
- Se i consumatori sono più veloce, allora le valori possono essere lette più di una volta
- Se i produttori sono più veloce, alcune valori possono essere perse
- Ovviamente, produttori e consumatori non possono scrivere e leggere allo stesso tempo

Prima soluzione (1)

- Si crea un oggetto **SharedVariable** che contiene un valore **val** et un boolean **readyToWrite**
- Se il boolean è true, si può scrivere e dopo si mette a false
- Se il boolean è false, si può leggere la variabile e si mette a true

Prima soluzione (2)

```
public class SharedVariable {
    public int val;
    public boolean readyToWrite;

    public SharedVariable() {
        val=0;
        readyToWrite=true;
    }

    public synchronized int read() {
        while(readyToWrite==true){}
        readyToWrite=false;
        return val;
    }

    public synchronized void write(int v) {
        while(readyToWrite==false){}
        readyToWrite=true;
        val=v;
    }
}
```

Producer

```
public class CodeProducer implements Runnable{

    private SharedVariable var;

    public CodeProducer(SharedVariable _var){
        this.var=_var;
    }

    public void run(){
        for(int i=0; i<100; i++){
            var.write(i);
        }
    }
}
```

Consumer

```
public class CodeConsumer implements Runnable{

    private SharedVariable var;

    public CodeConsumer(SharedVariable _var){
        this.var=_var;
    }

    public void run(){
        for(int i=0; i<100; i++){
            System.out.println(var.read());
        }
    }
}
```

Programma

```
public class TestProdCons{
    public static void main(String[] args){
        try{
            SharedVariable var=new SharedVariable();
            CodeProducer prod=new CodeProducer(var);
            CodeConsumer cons=new CodeConsumer(var);
            Thread []t=new Thread[20];
            for(int i=0; i<10; i++){
                t[i]=new Thread(prod);
            }
            for(int i=10; i<20; i++){
                t[i]=new Thread(cons);
            }
            for(int i=0; i<20; i++){
                t[i].start();
            }
            for(int i=0; i<20; i++){
                t[i].join();
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Problema

- C'è una attesa attiva con i cicli while, e uno producer o consumer potrebbe rimanere per sempre nel ciclo senza rilasciare il lock della variabile
- Soluzione :
- Usare i metodi **wait()**, **notify()** et **notifyAll()**
- **wait()** permette di rilasciare il lock e aspettare una notification
- **notify()/notifyAll()** permette di svegliare UNO/TUTTI gli thread in attesa che possa/possono riprovare a prendere il lock
- **ATTENZIONE:** meglio avere il lock per fare queste chiamate

Soluzione

```
public synchronized int read(){
    try{
        while(readyToWrite==true){wait();}
        readyToWrite=true;
        notifyAll();
    }
    catch(Exception e){
        System.out.println(e);
        e.printStackTrace();
    }
    return val;
}

public synchronized void write(int v){
    try{
        while(readyToWrite==false){wait();}
        readyToWrite=false;
        notifyAll();
        val=v;
    }
    catch(Exception e){
        System.out.println(e);
        e.printStackTrace();
    }
}
```