POLITECNICO DI MILANO

# 4. Concorrenza e Thread

Informatica 3

Andrea Mocci, mocci@elet.polimi.it

# Threads in Java


- Extend the Thread class (java.lang.Thread)

or

- Implement the Runnable interface (java.lang.Runnable)

# The Thread Class


- When creating a new thread by extending the **`Thread`** class, you should override the **`run()`** method

- ```
  public class FooThread extends Thread
  {
    public FooThread()
    {//initialize parameters
    }

    @Override
    public void run() {...}
  }
  ```
  ```

# Starting the thread


- To start a new thread, use the inherited method `start()`

```
FooThread ft = new FooThread();
ft.start();
```

# The `start()` and `run()` methods


- `start()` is responsible for two things:
  - Instructing the JVM to create a new thread
  - Call your Thread object's `run()` method in the new thread
  - You might think of `run()` as being similar to `main()`
- Like `main()`, `run()` defines a starting point for the JVM
- When the `run()` method exits, the thread ends

# The Runnable Class


- A thread can also be created by implementing the `Runnable` interface instead of extending the Thread class
- 
```
public class FooRunnable
implements Runnable
{
  public FooRunnable(){...}
  public void run(){...}
}
```

# Starting a Runnable


- Pass an object that implements Runnable to the constructor of a Thread object, then start the thread as before.
- 
```
FooRunnablefr= new FooRunnable();
new Thread(fr).start()
```

# The `join()` method


- The `join()` method is used to wait until thread is done.
- The caller of `join()` blocks until thread finishes.
- Why might this be bad?
  Blocking the main thread will make UI freeze.
- Other versions of `join` take in a timeout value.
- If thread doesn't finish before timeout, `join` returns.
- Alternatively, can check on a thread with `isAlive()` that returns true if running, false otherwise.

# The `sleep()` method


- Pauses execution of the current thread
- Why would we want to do this?
  - Periodic actions
  - Need to wait for some period of time before doing the action again
  - Allow other threads to run
- Implemented as a class method, so you don't actually need a Thread object
- ```
  Thread.sleep(1000);
  // Pause here for 1 second.
  ```
  ```

# Esercizio 1: thread Java


- Si costruisca un programma che lancia due thread, il primo di nome Paolo, il secondo di nome Egidio. Il thread principale deve far partire Egidio 1 s dopo Paolo, dunque mettersi in attesa di Paolo.

- Paolo e Egidio si limitano a contare fino a 5, aspettando 1/2 s ad ogni passo.

# Esercizio 1: soluzione


```
public class Filo extends Thread {
  int counter = 0;
  String name;
  public Filo(String name) {
    this.name = name;
  }
  public void run() {
    while (counter < 5) {
    try {
       sleep(500);
    } catch (InterruptedException e)  { }
    System.out.println(name + ": " + counter++);
  }
  ...
}
```

# Esercizio 1: soluzione


```
public static void main(String[] args) {
   Filo Egidio = new Filo("Egidio");
   Filo Paolo = new Filo("Paolo");

   Paolo.start();
   try {
       Thread.sleep(1000);
   } catch (InterruptedException e) {}
   Egidio.start();

   try {
       // Il thread corrente (main) prima di continuare, attende
       // che Paolo finisca
       Paolo.join();
   } catch (InterruptedException e1) {}
   System.out.println("Main: fine");
}
```

# Esercizio 1: output


```
Paolo : 1
Paolo : 2
Paolo : 3
Egidio : 1
Paolo : 4
Egidio : 2
Paolo : 5
Egidio : 3
main : fine
Egidio : 4
Egidio : 5
```

# Access synchronization


- Every object has a lock associated with it
- If you declare a method to be “synchronized”, then lock will be obtained before executing the method
- Can use this to make sure that only one thread at a time is accessing an object
- Example of synchronized method declaration:

```
synchronized int some_method(...)
```

# Wait and notify


- It's important to understand that `sleep()` does not release the lock when it is called. On the other hand, the method `wait()` does release the lock, which means that other `synchronized` methods in the thread object can be called during a `wait()`.
- Methods associated with ANY object.
- Every object maintains a list of "waiting" threads.
- A thread that is waiting at/on a particular object is suspended until some other thread wakes it.
- Notify() or NotifyAll awakens a thread that is waiting.

# Esercizio 2: un impianto con valvola

- Un impianto puo` portarsi dallo stato di funzionamento normale N in uno stato di gestione di malfunzionamento M.
- Entrato in tale stato, entro 5 s deve essere aperta una valvola di scarico.
- Se non si apre, l'impiano passa ad uno stato di fermo (F).
- Se la valvola viene aperta, essa rimane in tale stato per un tempo non inferiore a 20 s e non superiore a 30 s, poi l'impianto ritorna nello stato N.
- Compilatore ADA: ftp://cs.nyu.edu/pub/gnat

# Esercizio 2: soluzione Java

# Esercizio 2: impianto


```
public class Impianto extends Thread {
    private int stato; // 1 = N, -1 = M, 0 = F
    private Valvola valvola;
    public Impianto() {
       stato = 1;
    }
    public void run() { ... }
    public static void main(String[] args) {
      Impianto imp = new Impianto();
      imp.start();
    }
}
```

# Esercizio 2: impianto


```
public void run() {
  while (stato != 0) {
      System.out.println("sto lavorando!");
      while (Math.random() < .8) { // affidabilita` 80%
     System.out.println("tutto bene!");
      }
      valvola = new Valvola("Valvola Di Sfogo");
      stato = -1;
      valvola.start();
      try {
    System.out.println("Aspetto la valvola di sfogo...");
    synchronized (valvola) {
        valvola.wait(5000);
    }
```

# Esercizio 2: impianto


```
        } catch (InterruptedException ie) {
      ie.printStackTrace();
        }
        System.out.println("Controllo valvola aperta!");
        // Controllo se è cambiato lo stato della valvola
        if (valvola.getStato() == 0) {
        System.out.println("La valvola non si e' aperta!");
        stato = 0;
        } else
      try {
        System.out.println("La valvola si e' aperta,”);
            System.out.println(“aspetto...”);
            valvola.join();
            // Aspetto che la valvola abbia finito
    } catch (InterruptedException ie) { ... }
  }
}
```

# Esercizio 2: soluzione Java


```
public class Valvola extends Thread {
   private String name;
   private int stato; // 0 chiuso, 1 aperto
   public Valvola(String name) {
    this.name = name;
    this.stato = 0;
   }
   public void run() { ... }
   public int getStato() {
      return stato;
   }
}
```

# Esercizio 2: soluzione Java


```
public void run() {
  long t = 0; System.out.println("sono la " + name);
  try {
    t = (long) (5500 * Math.random()); sleep(t);
  } catch (InterruptedException ie) { ... }
  System.out.println(name + ": c'ho messo " + t +
   "ms");
  // Appena sveglio, cambio il mio stato e poi notifico
  stato = 1;
  synchronized (this) {
    notify();
  }
  try { System.out.println("Ora mi sfogo...");
    t = (long) (20000 + 10000 * Math.random());
   sleep(t);
  } catch (InterruptedException ie) { ... }
    stato = 0;
  }
```

# Esercizio 2: soluzione ADA

```
procedure MAIN is

 task IMPIANTO;

 task VALVOLA is
  entry APRI;
 end VALVOLA;

task body IMPIANTO is

 type STATO_T is ( N , M , F );

 STATO : STATO_T;

begin

 loop
    .....
    determinazione dello stato [ (M)alfunzioname
    .....
  if STATO = M then

  select
     VALVOLA.APRI;
     STATO := N;
or
     delay 5.0;
     -- la valvola non ha risposto entro 5 secondi
     STATO := F;
end select;

end if;
```

```
if STATO = F then
    ......
    azioni da eseguire in stato di fermo
    ......
 end if;
end loop;
end IMPIANTO;
```

# La valvola

```
task body VALVOLA is
    begin
     loop
      select
           accept APRI do

            -- apri valvola

            delay ( tempo variabile tra 20 e 30 secondi )

           end APRI;
```

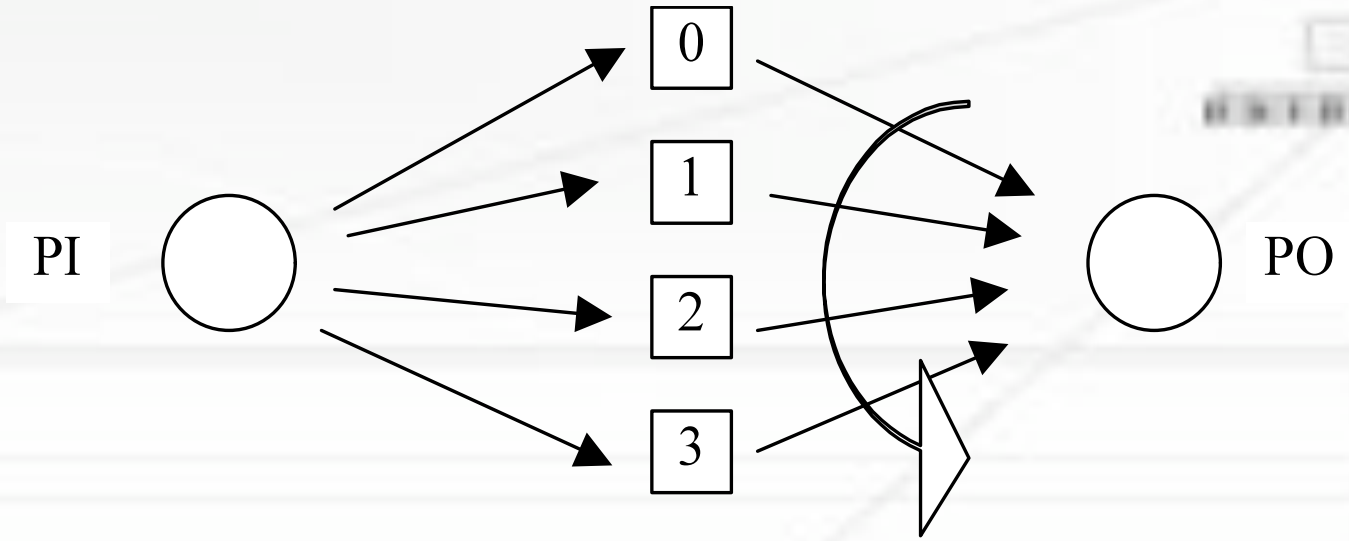

```
else
    -- azione eseguita nel caso non vi sia nessuna chiamata pendente
    -- per APRI:
    -- ritardo variabile durante il quale non può essere accettata
    -- alcuna chiamata per APRI. Se questo tempo è maggiore di
    -- 5 secondi e la sospensione ha inizio subito prima della chiamata
    -- ad APRI da parte dell'impianto, questo dopo 5 secondi entrerà
    -- nello stato di malfunzionamento.

    delay (tempo scelto casualmente tra 0 e N secondi)

    -- N è un parametro che caratterizza la risposta della valvola:
    -- quanto più N supera i 5 secondi maggiore sarà la probabilità che
    -- la valvola non risponda entro l'intervallo di tempo richiesto.
end select;

end loop;

end VALVOLA;
```

# Esercizio 3


- Il modulo PI esegue ripetutamente le seguenti operazioni: legge da tastiera una coppia di valori **<i, ch>**, dove i è un numero tra 0 e 3, **ch** un carattere, e inserisce il carattere **ch** nel buffer **i**
- Ognuno dei quattro buffer contiene al più un carattere.

- Il modulo PO considera a turno in modo circolare i quattro buffer e preleva il carattere in esso contenuto, scrivendo in uscita la coppia di valori **<i, ch>** se ha appena prelevato il carattere **ch** dal buffer **i**.
- L'accesso a ognuno dei buffer è in mutua esclusione: PI rimane bloccato se il buffer a cui accede è pieno, PO rimane bloccato se è vuoto.

# Esercizio 3



# Esercizio 3


- Data la seguente sequenza di valori letta da PI,
  <1, c> <0, b> <2, m> <0, f> <1, h> <3, n>
  scrivere la sequenza scritta in corrispondenza da PO.
- R.<0, b> <1, c> <2, m> <3, n> <0, f> <1, h>
- Descrivere brevemente in quali casi si può verificare una situazione di deadlock tra PI e PO. Illustrare con un semplice esempio.
- R: Deadlock: <1,a> <1,b>

# Esercizio 3: ADA

```
With TEXT_IO, Ada.Integer_text_IO;
use Ada.Integer_text_IO, TEXT_IO;
procedure PI_PO is
   task type BUF is
      entry PUT (X: in CHARACTER);
      entry GET (X: out CHARACTER);
   end BUF;
   BFS : array(0..3) of BUF;
   task PI;
   task PO;
```

```
task body BUF is
   Q : CHARACTER;
   FULL : BOOLEAN := FALSE;
begin
   loop
      select
         when not FULL =>
         accept PUT(X: in CHARACTER) do
            Q := X;
            FULL := TRUE;
         end PUT;
      or
         when FULL =>
         accept GET(X: out CHARACTER) do
            X := Q;
            FULL := FALSE;
         end GET;
      end select;
   end loop;
end BUF;
```

```
task body PI is
   B : INTEGER;
   V : CHARACTER;
begin
   loop
      TEXT_IO.PUT("Buff? >");
      Ada.Integer_Text_IO.GET(B);
      TEXT_IO.PUT("Char? >");
      TEXT_IO.GET(V);
      TEXT_IO.PUT_LINE(" ");
      BFS(B).PUT(V);
   end loop;
end PI;
```

```
task body PO is
   I : INTEGER := 0;
   V : CHARACTER;
begin
   loop
      for I in 0..3 loop
         BFS(I).GET(V);
         Ada.Integer_Text_IO.PUT(I);
         TEXT_IO.PUT(":");
         TEXT_IO.PUT(V);
         TEXT_IO.PUT_LINE(" ");
      end loop;
   end loop;
end PO;
begin -- corpo
   null;
end PI_PO;
```

# Esercizio 3: soluzione Java


```
public class Buf {
    private char q;
    private boolean full;
    Buf() {full = false;}

    public synchronized void put (char item) {
      while (full)
        try { wait(); }
        catch (InterruptedException e) { }
      q = item;
      full = true;
      notify();
    }
```

# Esercizio 3: soluzione Java


```
public synchronized char get () {
   while (!full)
        try { wait(); }
        catch (InterruptedException e) { }
   full = false;
   notify();
   return q;
   }
}
```

# Esercizio 3: soluzione Java


```
public class Pi extends Thread {
  private Buf[] buff;
  private String[] commands;
  Pi (Buf[] b, String[] c) {
    buff = b;
    commands = c;
  }
  public void run() {
    for(int i=0; i < commands.length; i++)
      buff[(int)commands[i].charAt(0)-(int)'0'].
        put(commands[i].charAt(2));
  }
}
```

# Esercizio 3: soluzione Java


```
public class Po extends Thread {
    private Buf[] buff;
    Po (Buf[] b) {
      buff = b;
    }

  public void run() {
    while(true) {
      for(int i=0; i < buff.length; i++)
      System.out.println("Buff "+i+":"+buff[i].get());
    }
  }
}
```

# Esercizio 3: soluzione Java


```
public class pi_po {
    public static void main (String [] args){
     Buf[] bfs = new Buf[4];
      Pi pi0;
      Po po0;

      bfs[0] = new Buf();
      bfs[1] = new Buf();
      bfs[2] = new Buf();
      bfs[3] = new Buf();

      pi0 = new Pi(bfs, args);
      po0 = new Po(bfs);

     pi0.start();
     po0.start();
   }
}
```

# Tema d'esame 8/5/2003


- Un tipico problema di programmazione concorrente considera un conto corrente bancario (modellato attraverso un *task* del linguaggio Ada o una classe di Java) al quale più utenti (modellati anch'essi da *task* Ada e da classi Java) possono accedere in modo concorrente per effettuare operazioni di versamento e di prelievo (realizzate mediante *entry* in Ada e metodi in Java aventi, ad esempio, il nome di "deposita" e di "ritira").
- Si considerino due varianti (a) e (b)

# Varianti

a) È definito un importo massimo prelevabile, e l'accesso al conto corrente da parte di un utente che vuole effettuare un prelievo è possibile solo se, indipendentemente dall'importo che l'utente intende effettivamente prelevare, nel conto corrente è presente un importo tale da rendere possibile il prelievo della cifra massima senza "andare in rosso"; in caso contrario il processo che intende effettuare il prelievo viene sospeso fino a quando la somma disponibile sul conto corrente non raggiunge un valore adeguato a permettere il prelievo dell'importo massimo.

b) L'accesso all'utente che vuole effettuare un prelievo è permesso se l'importo che si intende prelevare è inferiore all'attuale disponibilità, altrimenti il processo cliente viene sospeso fino a quando l'importo disponibile sul conto corrente raggiunge il valore del prelievo che si intende effettuare.

# Quesiti


- In entrambe le varianti del problema, le operazioni di versamento sono sempre ammesse (pur dovendo ovviamente garantire per ognuna di esse la mutua esclusione con qualsiasi altra operazione).
- Si dica quali delle soluzioni applicative (a) o (b) sono realizzabili con semplici programmi in Ada o in Java, motivando sinteticamente la risposta nel caso negativo.
- Ove possibile, tratteggiare, per le sole parti relative alla sincronizzazione tra i processi, i programmi che realizzano le operazioni di versamento e prelievo.
- NB: Non è necessario che le soluzioni siano rispettose della sintassi del linguaggio (in particolare, ciò vale per Ada). E' invece importante che la soluzione fornita sia rispettosa della semantica dei costrutti del linguaggio.

# Variante (a) Java


```
public class ContoCor {
  private int disponibile;
  private int soglia;
  ContoCor ()
  {... disponibile=0; soglia=2000; ...}

  public synchronized void deposita(int importo){
    disponibile += importo;
    notifyAll();
  }
  public synchronized void ritira(int importo){
    while (!(disponibile >= soglia) )
    try { wait(); } //aspetta di raggiungere la soglia
    catch(InterruptedException e) {}
    disponibile -= importo;
  }
}
```

# Variante (a) Ada


```
task ContoCor is
  entry deposita(importo: in INTEGER);
  entry ritira(importo: in INTEGER);
end ContoCor;
task body ContoCor is
  disponibile: INTEGER := 0; soglia := 2000;
  loop
    select
      when disponibile >= soglia
        accept ritira(importo: in INTEGER) do
          disponibile := disponibile - importo;
        end ritira;
      or
      when true
        accept deposita(importo: in INTEGER) do
          disponibile := disponibile + importo;
        end deposita;
    end select;
  end loop;
end ContoCor;
```

# Variante (b) Java


```
public class ContoCor {
  private int disponibile;
  private int soglia;
  ContoCor ()
  {... disponibile=0; soglia=2000; ...}

  public synchronized void deposita(int importo){
    disponibile += importo;
    notifyAll();
  }
  public synchronized void ritira(int importo){
    while (!(disponibile >= importo) )
    try { wait(); } //aspetta di raggiungere l'importo
    catch(InterruptedException e) {}
    disponibile -= importo;
  }
}
```

# Variante (b) Ada


- In Ada non esiste una semplice variante del caso (b) perché nella clausola *when* di una *select* non è possibile valutare il parametro della *entry* citata nella corrispondente *accept*;
- Il valore del parametro risulta noto solo DOPO l'esecuzione della *accept* e quindi non può essere utilizzato per decidere l'esecuzione dell'*accept* stessa.

# Domande varie...


- Quale politica segue ADA per la gestione dei task sospesi? E Java?
- Ada segue una politica FIFO mentre Java lascia la scelta non deterministica
- E' possibile imporre in Java di utilizzare la stessa politica di ADA ?
- Si potrebbe definire un oggetto CODA (tipo FIFO) da associare al monitor.
  - Prima di ogni wait, viene inserito un riferimento al task che viene sospeso.
  - Dopo ogni risveglio (NotifyAll()) ogni task verifica se esso è il primo task di coda.
  - Se si, si toglie dalla coda e procede, altrimenti si rimette in wait

# Deadlock and starvation


- Deadlock
  - Because threads can become blocked and because objects can have synchronized methods that prevent threads from accessing that object until the synchronization lock is released, it's possible for one thread to get stuck waiting for another thread, which in turn waits for another thread, etc.Youget a continuous loop of threads waiting on each other, and no one can move.
- Starvation
  - Starvation occurs when a process waits for a resource that continually becomes available, BUT is NEVER assigned to that process because of priority or a flaw in the design of the scheduler.

# Example: Dining Philosophers


- Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.
- One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

http://www-dse.doc.ic.ac.uk/concurrency/book_applets/Diners.html

# Dining Philosopher


- A Philosopher can
  - Think
  - Eat
  - Take the fork left
  - Take the fork right
  - Put down the fork left
  - Put down the fork right
- It is obvious that
  - No two neighbours can eat at the same time
  - Only two philosophers can eat at the same time

# Dining Philosopher: deadlock solution


- Philosophers grab whatever forks are available as soon as they are hungry
- Philosophers do not release the fork unless they have got a chance to eat
- Example of deadlock occurs when all the philosophers decide to take the fork on the right then decide to lift the fork on the left.
- If nobody puts down a fork, a circular dependency of the condition on the resource exists, i.e., a deadlock

# Dining Philosopher: deadlock-free


- Taking the fork is placed in critical section
- Philosopher tries to take the left fork first
- If they have the left fork they are allowed to take the right fork
- If the right is unavailable, they must put down the left fork
- Forks are only in use when a philosopher is eating
- Two philosopher can eat at the same time and one fork will always be free

# Solution


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

# Solution


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

# Solution


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

# Solution


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