Università di Ferrara
Laurea Triennale in Informatica
A.A. 2021-2022
Sistemi Operativi e Laboratorio

Lab-07. I Thread in Java

Prof. Carlo Giannelli

http://www.unife.it/scienze/informatica/insegnamenti/sistemi-operativi-laboratorio
http://docente.unife.it/carlo.giannelli
https://ds.unife.it/people/carlo.giannelli

I Thread in Java

Due principali modalità per creare Thread in Java.

Implementazione interfaccia Runnable

1. Definire una classe che implementi l'interfaccia Runnable, quindi definendone il metodo run().
2. Creare un’istanza di tale classe.
3. Creare un’istanza della classe Thread, passando al costruttore un reference all’oggetto Runnable creato precedentemente.
4. Invocare il metodo start() sull'oggetto Thread appena creato.

Estendere classe Thread

1. Definire una sottoclasse della classe Thread, facendo un opportuno override del metodo run().
2. Creare un’istanza di tale sottoclasse.
3. Invocare il metodo start() su tale istanza, la quale, a sua volta, richiamerà il metodo run() in un thread separato.

I Thread in Java

Materiale Aggiuntivo

Un altro metodo consiste nello sfruttare entrambi i metodi:

- definire una classe che implementa l'interfaccia Runnable
- definire il metodo run()
- definire un metodo start() che va a inizializzare un'istanza locale della classe Thread e a invocare il metodo start() sull'istanza appena creata

Per utilizzare la classe appena definita dobbiamo poi **crearne un'istanza e chiamare il metodo start()** su di essa

Vediamo un esempio…

I Thread in Java

Materiale Aggiuntivo

```
class MyThread implements Runnable {

      private Thread t;

      //eventuale costruttore con parametri
      public MyThread(...){...}

      public void start(){
            // this passa l'istanza corrente a Thread
            t = new Thread(this);
            t.start();
      }

      public void run() {
            //codice Thread
      }
...
}
```

I Thread in Java

Materiale Aggiuntivo

È possibile invocare MyThread nel seguente modo:

```
public static void main(String args[]) {

      MyThread mythread = new MyThread();
      // mando in esecuzione
      mythread.start();

}
```

Esercizio 1: Creazione thread

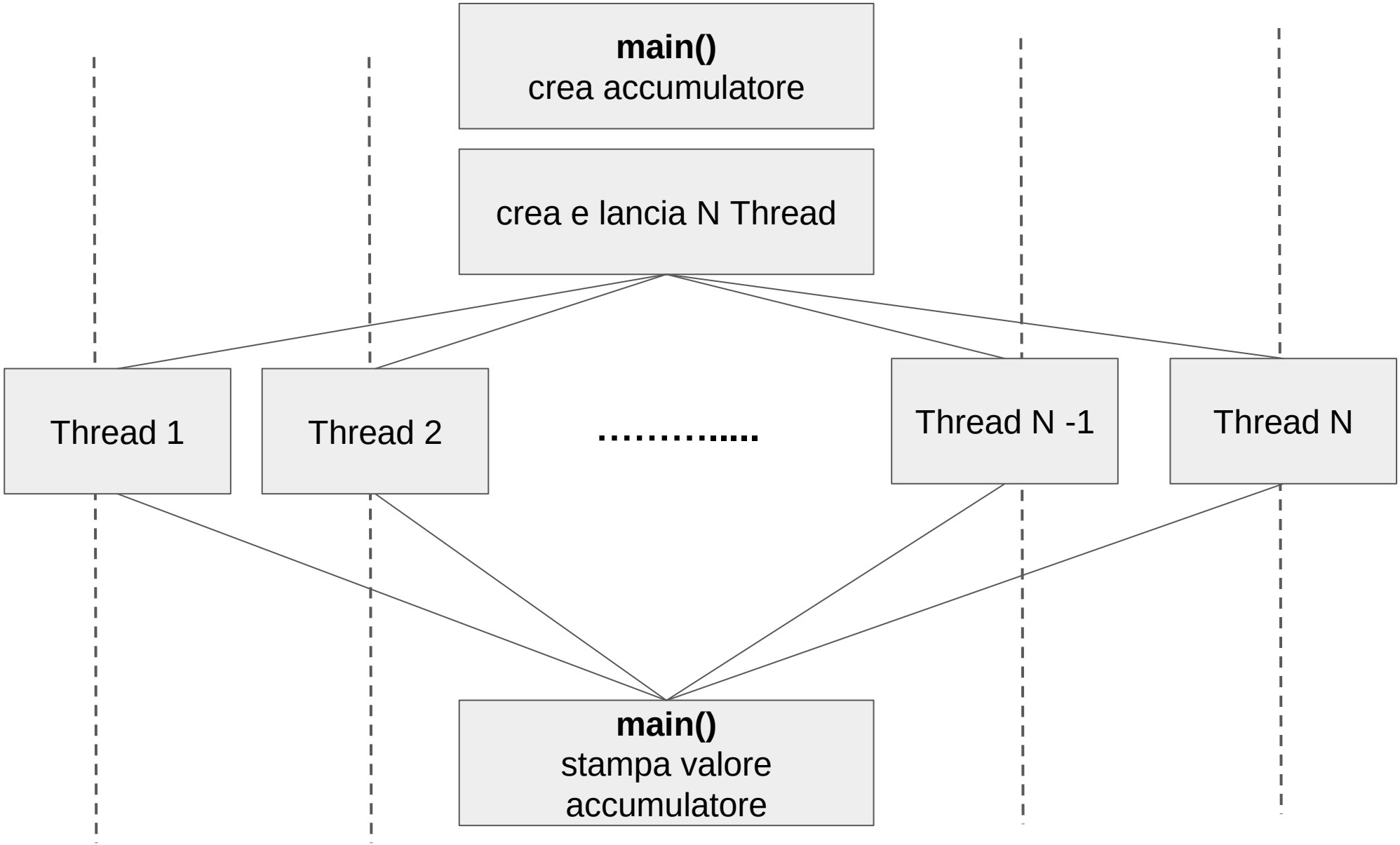
Si implementi un programma che deve creare un numero N di thread, dove N è specificato in input dall'utente.

Ciascuno degli N thread ha il compito di generare un valore casuale e di aggiungerlo a un **accumulatore presente nel main**. Infine, il thread main si dovrà occupare di stampare a video il valore dell'accumulatore **solamente dopo** che gli N thread hanno terminato la loro esecuzione.

Si definisca il thread contatore utilizzando entrambe le modalità viste a lezione: estendendo la classe Thread o implementando l'interfaccia Runnable.

Nota: si trascurino eventuali problemi causati dall'accesso concorrente all'accumulatore.

Esercizio 1: traccia 1/2



Esercizio 1: traccia 2/2

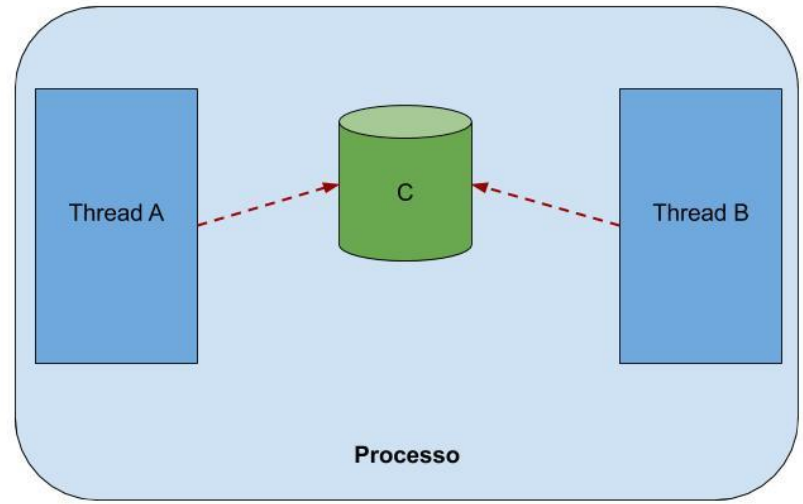
- Definire una classe **Accumulatore**:
  - il costruttore di accumulatore riceve come parametro un double con cui viene inizializzato il contatore
  - definire un metodo addValue(double value) per aggiornare il valore del contatore
  - definire un metodo getValue() per restituire il valore del contatore

- Definire una classe **CounterThread**:
  - implementare il metodo run() che aggiunge un valore random all'accumulatore

- Metodo **main**:
  - crea l'oggetto Accumulatore
  - crea e lancia N Thread (passando l'oggetto Accumulatore)
  - aspetta la terminazione dei thread
  - stampa il valore all'interno della classe Accumulatore

Comunicazione fra thread: memoria condivisa

In un ambiente a **memoria globale più thread comunicano direttamente accedendo direttamente alla memoria condivisa**. Questa rappresenta la principale differenza tra il modello ad ambiente locale (come abbiamo visto per i processi in C) e il modello ad ambiente globale (multi-thread in Java).



Questo significa che due thread (A e B) possono **accedere a uno stesso oggetto C che è condiviso fra i thread** di un unico processo/JVM. Ovviamente, la condivisione crea tutti i problemi legati all'uso corretto di risorse condivise:
si vogliono **evitare interferenze nell'accesso alle aree di memoria comuni** che possano pregiudicare il corretto funzionamento di un programma.

Comunicazione fra thread: stream

Altri modelli di comunicazione prevedono l'utilizzo di uno stream di dati come Socket o **Piped Stream**.

Gli stream sono utili in quanto permettono di svolgere operazioni come:

- lettura di stringhe System.in (stdin)
- scrittura su File e Socket

Questi stream possono essere incapsulati in oggetti di più alto livello che permettono una gestione più user-friendly: **la lettura di stringhe invece che di byte** da uno stream di input ne è un esempio.

```
BufferedReader br =
   new BufferedReader(new InputStreamReader(System.in));
String line = br.readLine();
```

BufferedReader/Writer utilie per stringhe, ma per oggetti? → Serializable

Comunicazione fra thread: serializable 1/2

Java mette a disposizione un meccanismo per **salvare e caricare lo stato di un oggetto attraverso Serializable**. Una classe è serializzabile quando implementa **l'interfaccia *Serializable***.

Vogliamo definire una classe Rectangle che espone un metodo pubblico getArea(). Un oggetto di tipo Rectangle deve poter essere **scritto e letto da file**.

```
class Rectangle implements Serializable {
...
   private int x;
   private int y;
   public int getArea() {
        return x * y;
   }
...
}
```

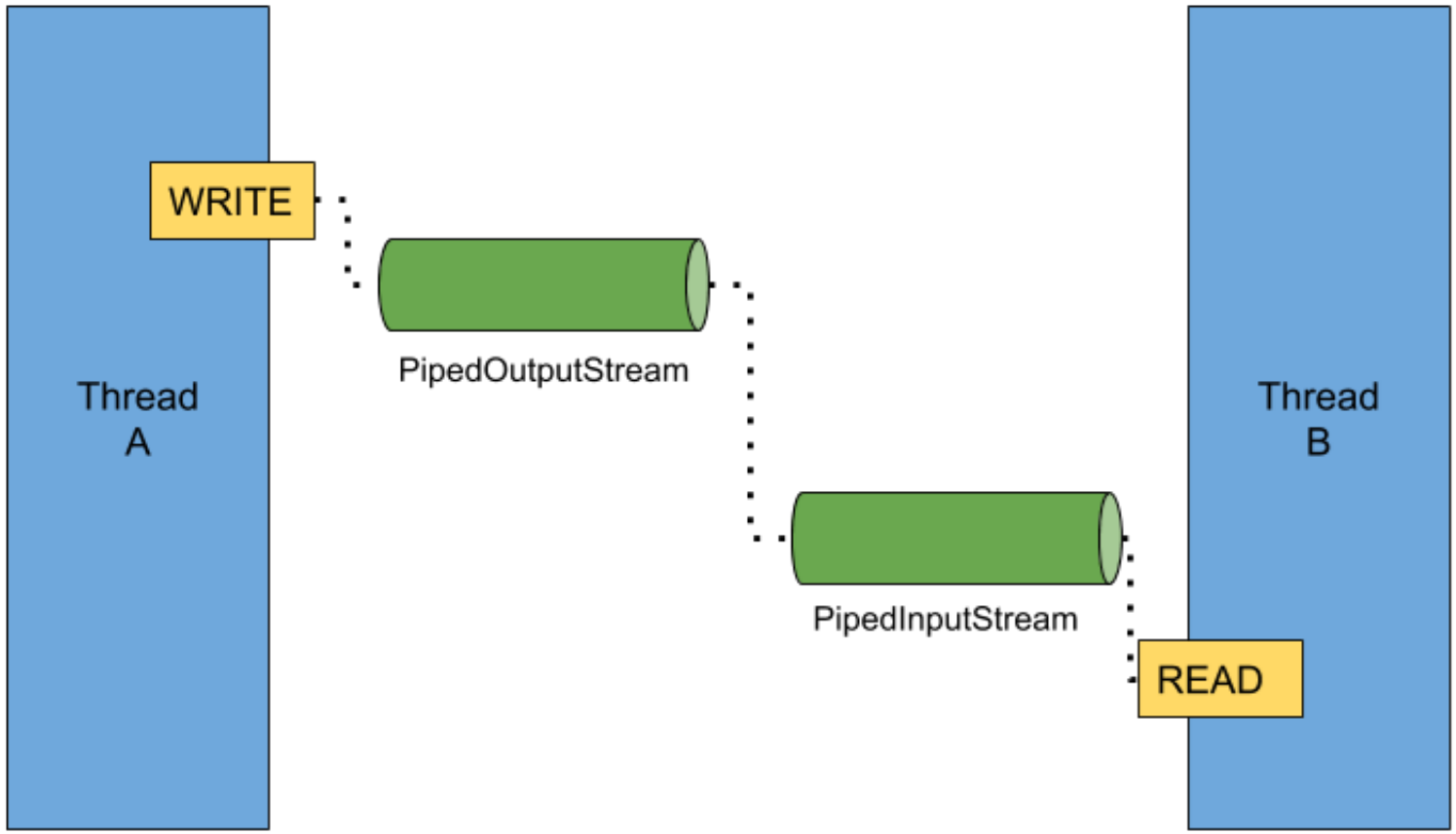
Comunicazione fra thread: serializable 2/2

```
public static void main(String args[]) {
    Rectangle rect = new Rectangle(5, 3);
    ...
    ObjectOutputStream oos = new ObjectOutputStream(
                        new FileOutputStream("rect"));
   ObjectInputStream ois = new ObjectInputStream(
                                new FileInputStream("rect"));
   // serializzo rect su file
   oos.writeObject(rect);
   // leggo rect da file
   Rectangle rect2 = (Rectangle)ois.readObject();
    ...
}
```

Attenzione a non dimenticare il try/catch per il controllo delle eccezioni!

Comunicazione fra thread: piped stream 1/2

Utilizzando PipedOutputStream e PipedInputStream è possibile creare uno **stream di comunicazione fra due Thread**.



Comunicazione fra thread: piped stream 2/2

Per realizzare una comunicazione, PipedInputStream e PipedOutputStream devono essere connessi, ossia **il PipedInputStream deve leggere da PipedOutputStream**.

```
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream(pos);
```

In questo modo, il PipedInputStream è collegato al PipedOutputStream.
Gli stream devono poi essere passati ai due thread A e B.

→ Comunicazione tra thread scambiando tramite Piped Stream istanze di oggetti che implementano l'interfaccia Serializable

Esercizio 2: Comunicazione fra thread

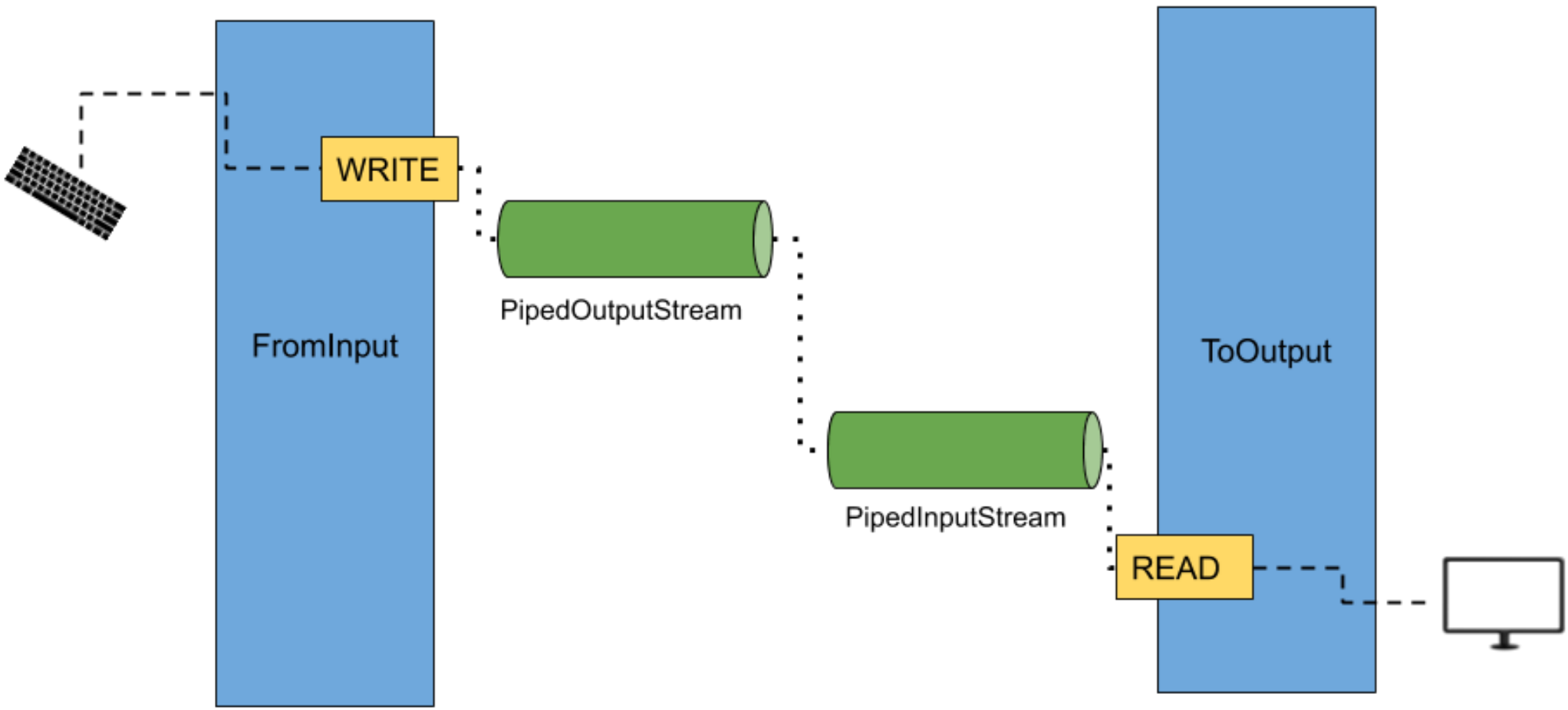
Definire due classi **FromInput e ToOutput come thread** creati dallo stesso main. Si vuole che i due thread **comunichino utilizzando le Piped Stream**. In particolare, FromInput deve leggere dallo standard input un messaggio e scriverlo sulla Piped Stream collegata a ToOutput, che si deve occupare invece di leggere lo stream e riportarlo sullo standard output.

Si provi a realizzare la comunicazione utilizzando:

1. inviando direttamente il **byte array** della stringa letta in input
2. una scrittura/lettura di tipo **buffered**
3. un **oggetto** serializzabile Message

Si modifichi poi la soluzione per leggere e scrivere su File, utilizzando due file separati.

Esercizio 2: traccia 1/2



Esercizio 2: traccia 2/2

Nella classe **FromInput**:

- while:
  - leggere una riga da tastiera
  - scrivere su PipedOuputStream

Nella classe **ToOutput**:

- while:
  - leggere da PipedInputstream
  - stampare i dati letti a video

Nel main:

- creare PipedOutputStream
- creare PipedInputStream da PipedOutputStream
- passare PipedInputStream a ToOutput
- passare PipedOutputStream a FromInput
- mandare in esecuzione i due thread FromInput e ToOutput