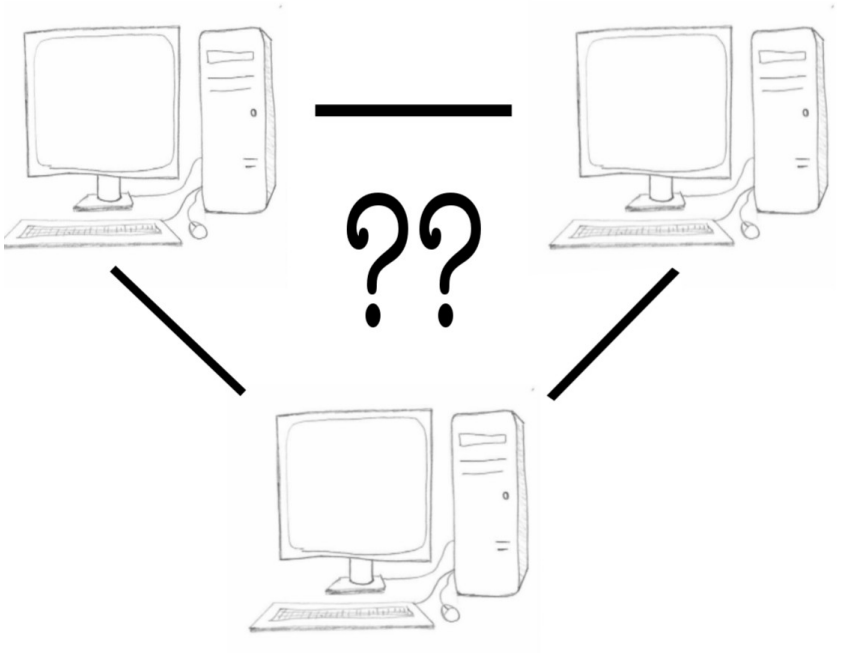
PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@liafa.univ-paris-diderot.fr

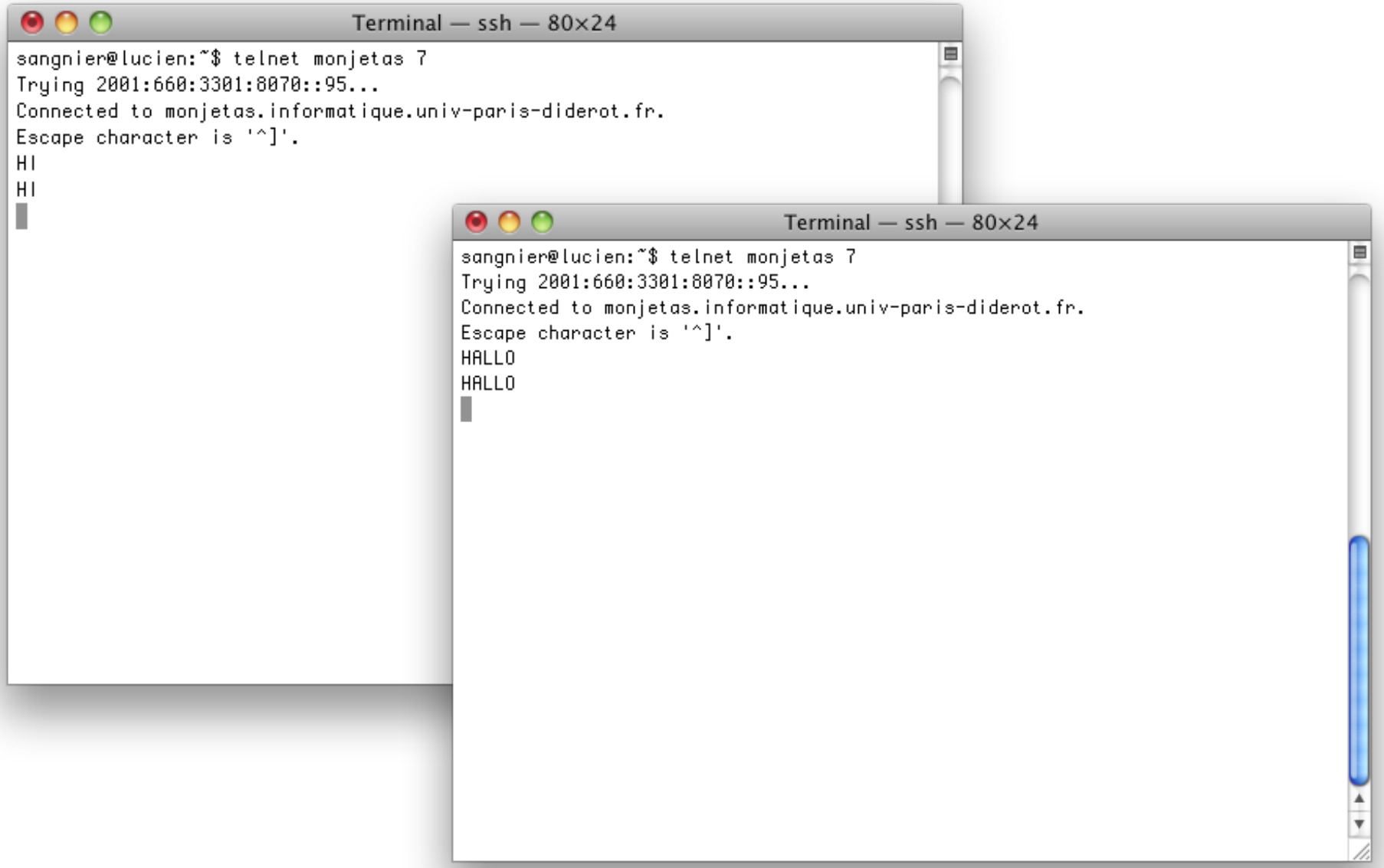
La concurrence en Java



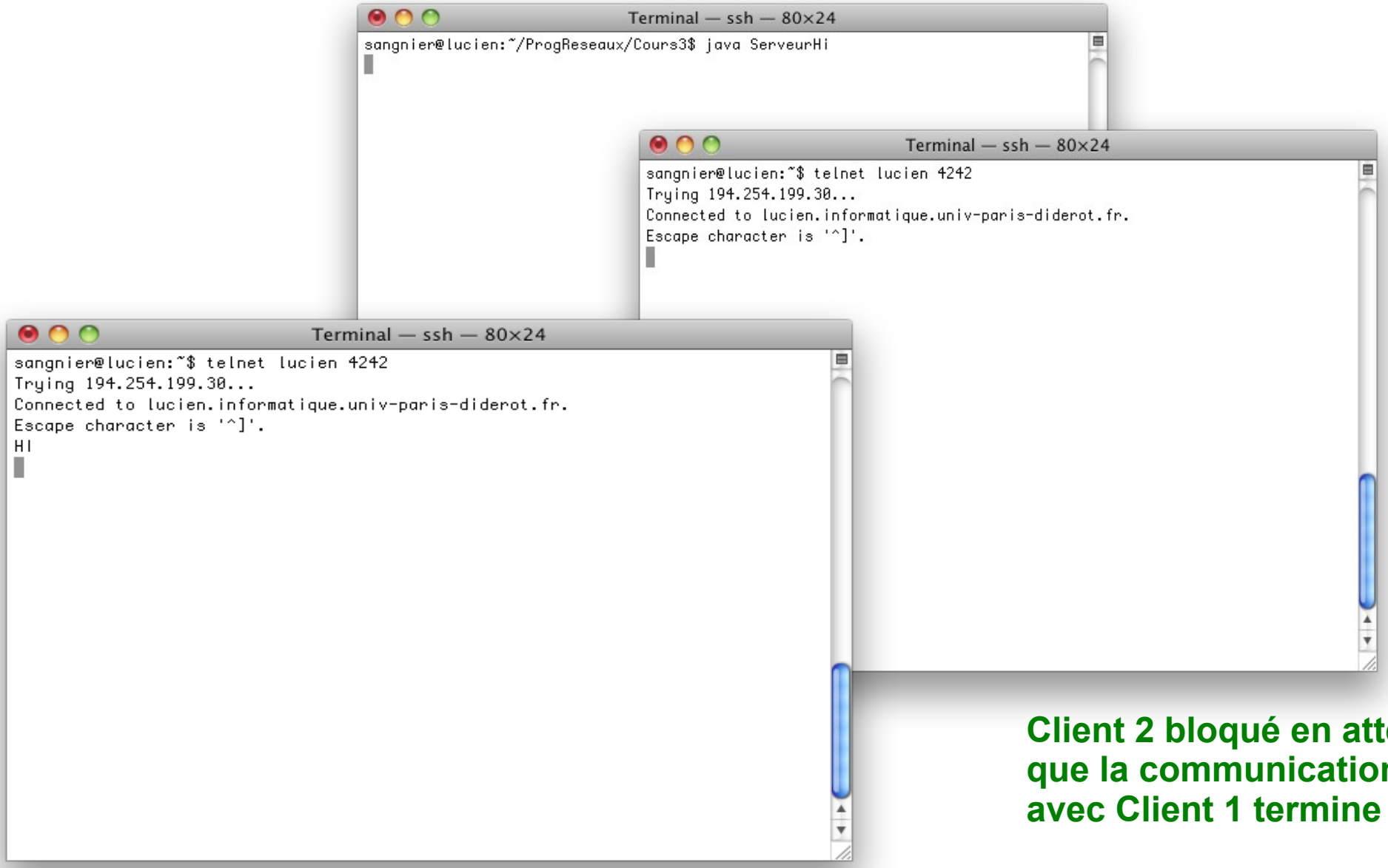
Plusieurs connexions sur un serveur

- En général, on veut qu'un serveur accepte plusieurs connexions en même temps
- Prenons l'exemple du service **echo tcp** (**port 7**) sur **monjetas**
  - Il y a un seul service qui tourne
  - Ouvrons deux terminaux
    - Sur chacun des deux on fait **telnet monjetas 7**
    - Le service accepte les deux connexions
    - Les deux terminaux peuvent communiquer avec le service en parallèle
    - Si on ferme l'un, l'autre continue de pouvoir communiquer

Deux clients en parallèle



Que se passe-t-il avec deux clients (1)



Client 2 bloqué en attente que la communication avec Client 1 termine

Client 1

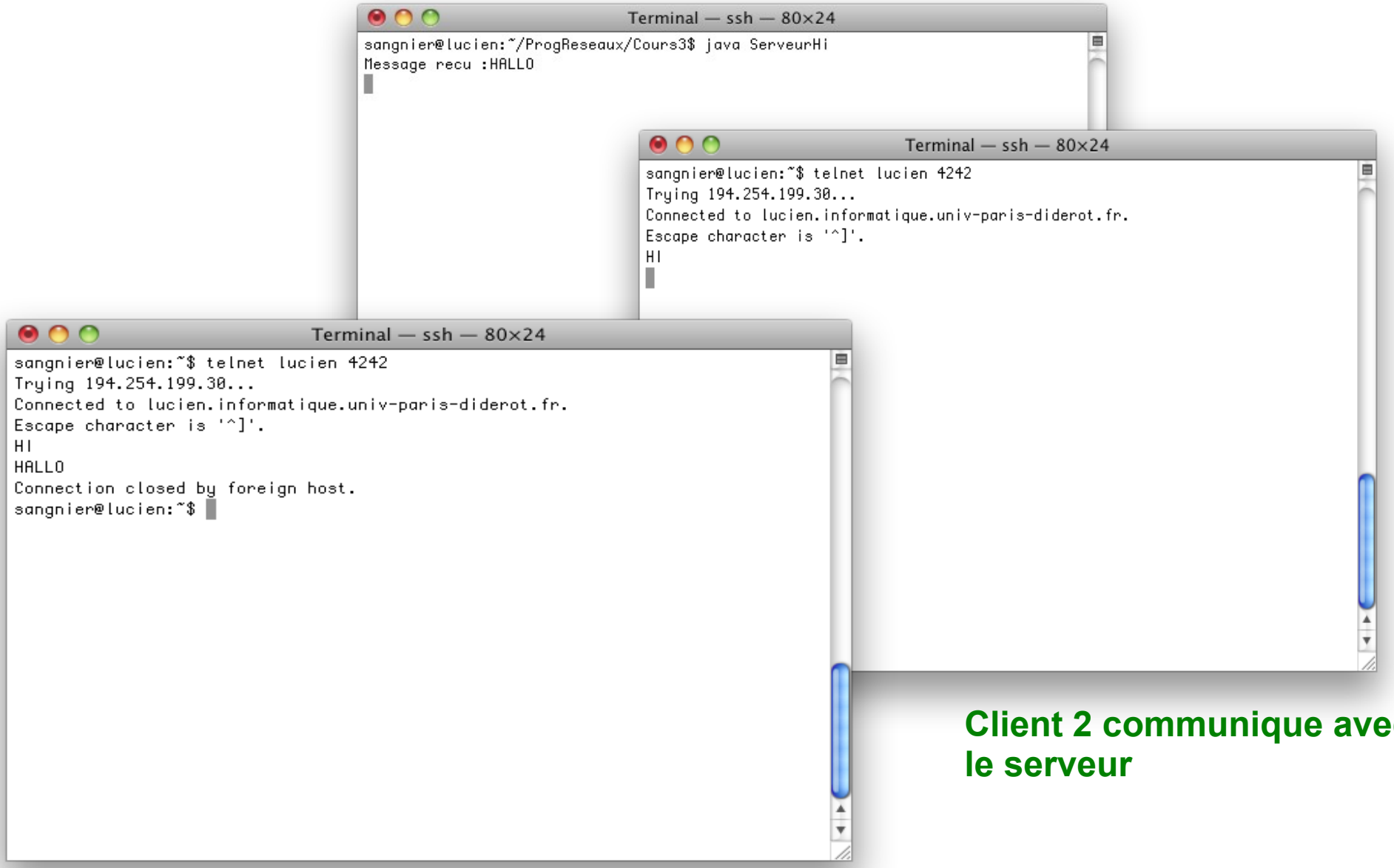
Reprenons notre serveur exemple

- Rappel du comportement :
  - Attendre une connexion sur le port 4242
  - Envoyer un message `"Hi\n"`
  - Attendre un message du client
  - Afficher le message du client
  - Et recommencer à attendre une communication
- Scénarios avec deux clients
  - Un premier client se connecte
    - il reçoit Hi
  - Un deuxième client se connecte avant que le premier client ait envoyé son message
    - Le second client reste bloqué

Code du serveur

```
import java.net.*;
import java.io.*;
public class ServeurHi{
    public static void main(String[] args){
        try{
            ServerSocket server=new ServerSocket(4242);
            while(true){
                Socket socket=server.accept();
                BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
                pw.print("HI\n");
                pw.flush();
                String mess=br.readLine();
                System.out.println("Message recu :"+mess);
                br.close();
                pw.close();
                socket.close();
            } }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Que se passe-t-il avec deux clients (2)



Client 2 communique avec le serveur

Client 1 a terminé

Comment résoudre le problème

- Ce que l'on voudrait
  - Que deux ou plus clients puissent communiquer en même temps avec le serveur
  - C'est-à-dire sur notre exemple
    - Si un premier client a reçu `"Hi\n"`
    - Si il ne répond pas et qu'un deuxième client arrive
    - Le deuxième client reçoit aussi `"Hi\n"`
- Pour cela
  - Notre serveur doit pouvoir en même temps
    - Communiquer avec un client
    - Attendre les demandes de connexions sur **accept()**
- Comment fait-on cela :
  - Avec un serveur **concurrent** ou **multi-threadé**

À propos des processus

- Un processus est un **programme** (de nature statique) en cours d'**exécution** (de nature dynamique)
  - son exécution nécessite un environnement
    - espace d'adressage
    - objets entrées/sorties (par exemple sortie standard et entrée standard)
- Plusieurs processus peuvent s'exécuter sur une même machine de façon quasi-simultanée
  - Si le système d'exploitation est **à temps partagé** ou **multi-tâche**
  - Ce même sur une machine mono-processeur
  - Le système d'exploitation est chargé d'allouer les ressources
    - mémoire, temps processeur, entrées/sorties
  - On a *l'illusion du parallèlisme*

Les processus en Java

- Java permet de manipuler des **processus**
- Ces processus ne sont toutefois pas pris en charge par la JVM (Java Virtual Machine) où s'exécute le programme Java
- Ils sont pris en charge par le système
- Donc il n'y a pas de notion de processus au sein de la JVM, un processus est un objet du système hôte
- Au sein d'un programme Java :
  - On va pouvoir dire au système hôte d'exécuter un processus
  - On pourra également récupérer des informations sur ce processus (comme par exemple ses entrées/sorties standard)

L'environnement d'exécution

- Il d'abord récupérer l'environnement d'exécution de la JVM
- Il est disponible sous la forme d'un objet de la classe **java.lang.Runtime**
- Remarques :
  - il n'existe qu'un seul objet de cette classe
  - On ne peut pas en créer
- Pour le récupérer on utilise la méthode statique **Runtime.getRuntime()**
- De nombreuses méthodes sont disponibles dans la classe java.lang.Runtime (cf documenation de l'API), en particulier
  - Des méthode permettant de dire au système de lancer un processus
    - C'est la famille des méthodes **Process exec(...)**

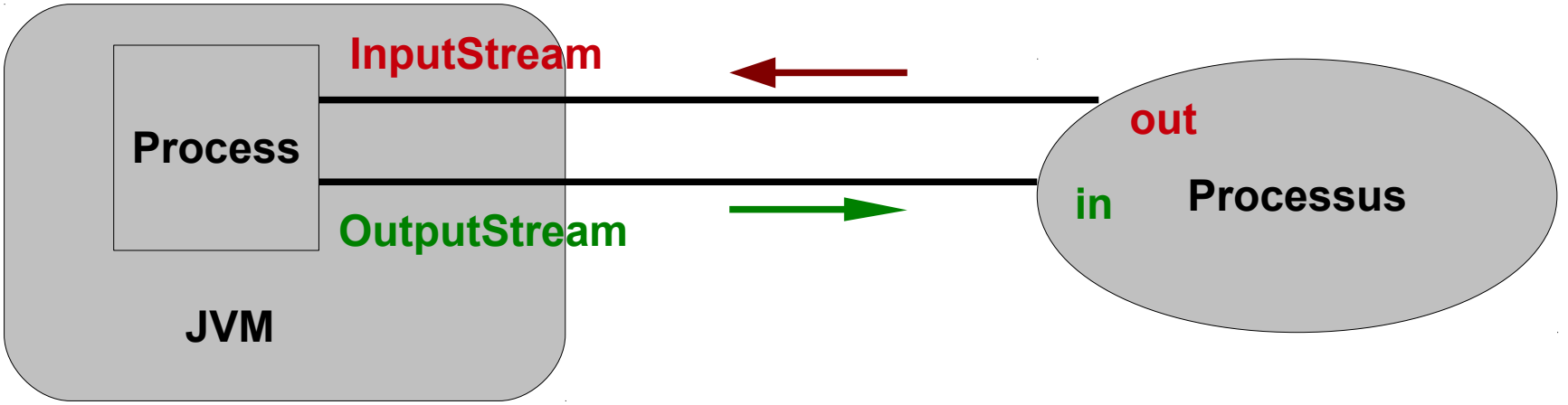
Les processus en Java

- La JVM permet donc de lancer des processus externes
- Ils sont représentés dans celle-ci sous forme d'objets de type **java.lang.Process**
- À quoi servent ces objets :
  - À communiquer avec les processus externes correspondants
  - À se synchroniser avec leur exécution
    - Par exemple pour attendre qu'un processus est fini de s'exécuter
- Comment créer de tels processus :
  - Par exemple, pour lancer un processus qui fait **ls -a**

```
Process process = Runtime.getRuntime().exec("ls -a");
```

Communiquer avec les processus

- On veut pouvoir communiquer avec les processus
- Par exemple pour lire ce qu'ils affichent et le faire afficher par la JVM
- Pour ça on retrouve dans nos méthodes préférées à base de flux
  - **InputStream getInputStream()**
  - **OutputStream getOutputStream()**
- **Attention** : ici, les flux sont à comprendre du côte du programme



Synchronisation avec un processus

- Deux méthodes de synchronisation avec un processus lancé (toujours dans la classe **java.lang.Process**
  - **int waitFor() throws InterruptedException**
    - permet d'attendre la fin de l'exécution d'un processus
    - retourne la valeur retournée à la fin de l'exécution du processus
      - Par convention 0 si tout se passe bien
    - Cette méthode est bloquante
  - **int exitValue()**
    - retourne la valeur retournée à la fin de l'exécution du processus
- Il est important d'attendre la fin du process
- Si le programme Java termine avant la fin de l'exécution du process, on n'a plus moyen de récupérer les informations

Exemple

```
import java.io.IOException;
import java.io.InputStreamReader;
public class ExecLs
{
    public static void main(String[] args){
        try {
            Process process = Runtime.getRuntime().exec("ls -a");
            BufferedReader stdout = new BufferedReader(new
InputStreamReader( process.getInputStream()));
            String line = stdout.readLine() ;
            while(line != null){
                System.out.println(line);
                line = stdout.readLine() ;
            }
            stdout.close();
        }
        catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Problème des processus

- En fait, on en va pas utiliser la commande **exec(...)** de la classe **Runtime**
- Au lieu de manipuler des processus du système hôte, on va utiliser des **processus légers** (t**hreads**) qui existent dans la jvm

Les processus légers (threads)

- Un **thread** est un fil d'exécution dans un programme, qui est lui même exécuté par un processus
- Un processus peut avoir plusieurs threads
  - Il est alors **multi-threadé**
  - Au minimum il y a un thread
- Chaque fil d'exécution est distinct des autres et a pour attributs
  - Un point courant d'exécution (**pointeur d'intstruction** ou ***PC (Program Counter)***)
  - Une pile d'exécution (***stack***)
- Un thread partage tout le reste de l'environnement avec les autres threads qui lui sont concurrents dans le même processus
- **La JVM est multi-threadée** et offre au programmeur la possibilité de manipuler des threads
  - Il n'est pas précisé comment ces threads sont pris en charge par le système

Les threads en java

- Le mécanisme est plus complexe que les celui des processus car il est interne au système Java
- Il repose sur deux types importantes
  - L'interface **java.lang.Runnable**
    - Elle ne contient qu'une méthode à implémenter
      - **void run()**
      - C'est cette méthode qui contiendra le programme exécuter par un thread
  - La classe **java.lang.Thread**
    - C'est elle qui nous permettra de manipuler les threads
- En bref, dans la méthode **run()**, on aura le code du thread (la partie statique) et on va se servir d'un objet Thread pour gérer le fil d'exécution (la partie dynamique)

Liens entre Thread et Runnable

- À tout thread on associe un objet implémentant **Runnable**
  - Cf un des constructeurs dans **java.lang.Thread**
    - **public Thread(Runnable target)**
- Le même objet implémentant **Runnable** peut-être associé à plusieurs threads
  - Dans ce cas chaque thread exécute de façon concurrente la méthode run() de l'objet passé au constructeur

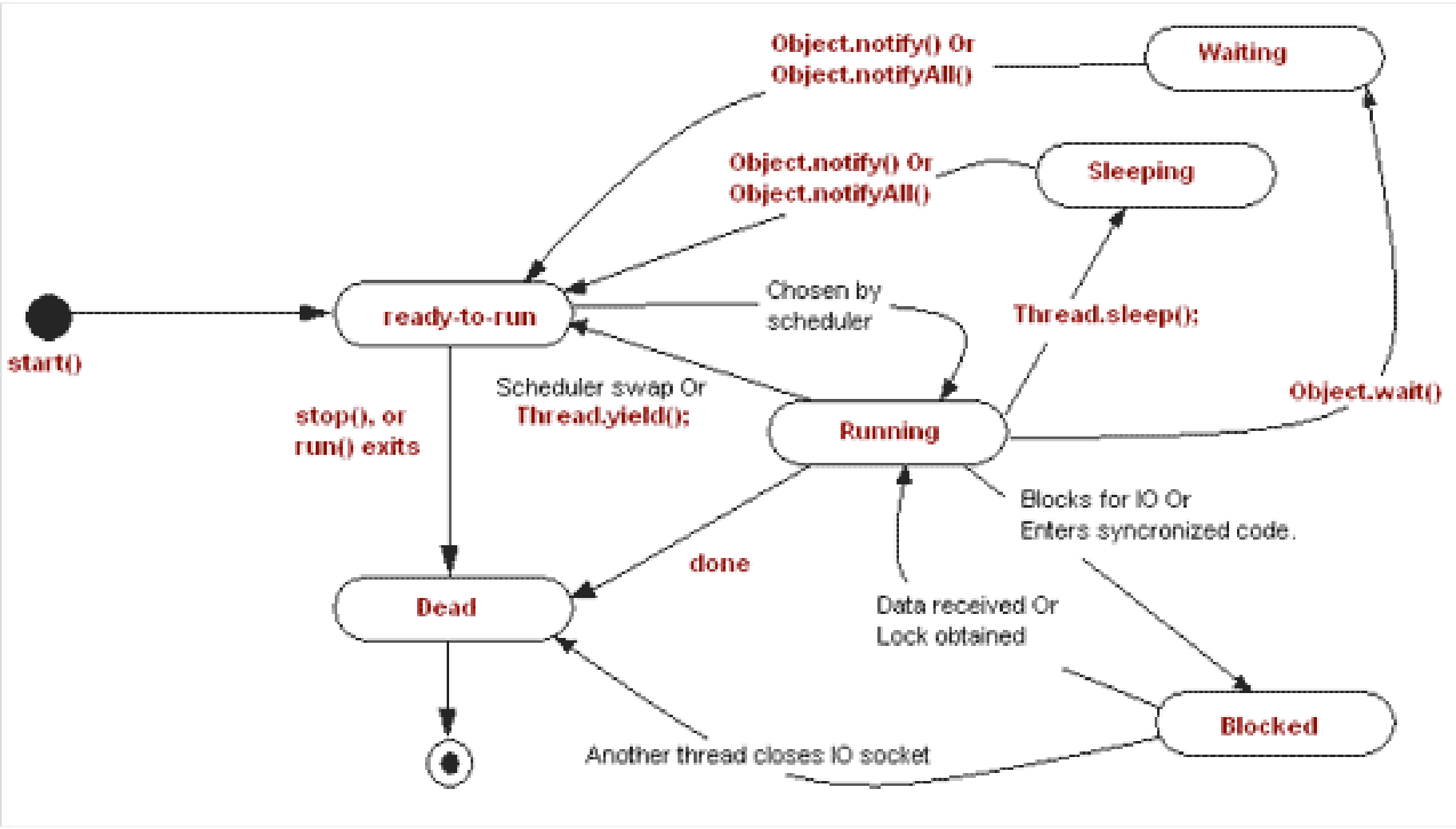
L'interface java.lang.Runnable

- **java.lang.Runnable** est donc une interface avec une seule méthode à implémenter
  - **public void run()**
- Lorsqu'un thread démarrera son exécution
  - Il débutera par un appel à la méthode **run()** du **Runnable** qui lui est attaché
  - Il terminera son exécution lorsque cette méthode **run()** terminera
- Attention pour lancer un thread on ne fait pas appel à **run()** mais à une méthode qui s'appelle **start()**

La classe java.lang.Thread

- Les Threads Java ont plusieurs attributs :
  - **String name** : le nom du thread
  - **long id** : l'identité du thread
  - **int priority** : sa priorité (les threads sont ordonnancés selon cette priorité)
  - **boolean daemon** : son mode d'exécution (démon ou non, voir plus loin)
  - **Thread.state state** : son état
    - **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TERMINATED**,...
  - Sa pile (stack) dont on peut seulement observer son état
  - son groupe de thread
  - etc (cf la documentation)
- Dans **java.lang.Thread**, on a les accesseurs pour ces attributs

Les états d'un thread



Terminaison d'une JVM

- On indique souvent qu'un programme s'arrête lorsqu'on sort du main:
  - Un programme ne s'arrête pas, c'est le processus sous-jacent qui s'arrête
  - Mais surtout il ne suffit pas de sortir du main, il faut sortir du premier appel au main (il est possible de faire des appels récursifs)
  - Il faut aussi attendre que TOUS les threads qui ne sont pas des démons s'arrêtent
  - Il existe au moins un thread démon
    - **Le garbage collector**
  - Souvent il en existe un autre
    - Le thread qui gère l'interface graphique

Création et contrôle des thread

- Pour créer un thread, on a plusieurs constructeurs possibles :
  - **Thread(Runnable target), Thread(Runnable target, String name)**
- Il existe plusieurs méthodes pour contrôler l'exécution d'un thread
  - **void start()**
    - Elle permet de démarrer le thread
    - Celui-ci va alors appeler la méthode **run()** du **Runnable** qui lui est associé
  - **void join()**
    - Elle permet d'attendre la fin de l'exécution du thread
  - **void interrupt()**
    - positionne le statut du thread à interrompu
    - n'a aucun effet immédiat (permet au thread de savoir qu'un autre thread souhaite l'interrompe)
  - **IMPORTANT : Il n'existe pas de techniques pour arrêter un thread**, il faut se débrouiller pour qu'il finisse son premier appel à run

Les méthodes statiques de Thread

- **Thread currentThread()**
  - Permet de récupérer l'objet Thread courant (qui exécute la ligne)
  - Utile par exemple pour récupérer le nom
- **boolean isInterrupted()**
  - Pour tester si le thread a reçu une demande d'interruption
- **void sleep(long ms)**
  - Pour faire dormir le thread courant pour la durée exprimée en millisecondes (temps minimum)
- **void yield()**
  - Le thread renonce à la suite de son exécution temporairement et est placé dans l'ordonnanceur

Exemple 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();
        }
    }
}
```

Exemple

```
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");
            //t.setDaemon(true);
            t1.start();
            t2.start();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }

}
```

Et avec le réseau ?

- On va créer un thread qui va prendre en charge les communications avec le réseau
- Après chaque accept, on va créer un thread à qui on donnera la socket de communication

Les threads pour notre serveur

- Quelques règles de base :
  - On va créer une classe implémentant **Runnable** dont la méthode **run()** prendra en charge la communication
  - À cette classe on associera une socket, pour cela il suffit de la passer au constructeur
    - **public Service(Socket s)**
    - Ainsi la méthode **run()** aura accès à cette socket
  - Après un **accept**
    - On récupère la socket
    - On crée un nouvel objet implémentant **Runnable**
    - On démarre le thread correspondant
  - À la fin de **run()**, on oublie pas de fermer la socket correspondante

Structure d'un service

```
import java.net.*;
import java.io.*;
import java.lang.*;
public class ServiceHi implements Runnable{
    public Socket socket;

    public ServiceHi(Socket s){
        this.socket=s;
    }

    public void run(){
        try{
            BufferedReader br=new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
            pw.print("HI\n");
            pw.flush();
            String mess=br.readLine();
            System.out.println("Message recu :"+mess);
            br.close();
            pw.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Le serveur concurrent associé

```
import java.net.*;
import java.io.*;
public class ServeurHiConcur{
    public static void main(String[] args){
        try{
            ServerSocket server=new ServerSocket(4242);
            while(true){
                Socket socket=server.accept();
                ServiceHi serv=new ServiceHi(socket);
                Thread t=new Thread(serv);
                t.start();
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Les problèmes de la concurrence (1)

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

public class Compteur{

    private int valeur;

    public Compteur(){
        valeur=0;
    }

    public int getValeur(){
        return valeur;
    }

    public void setValeur(int v){
        valeur=v;
    }
}
```

Les problèmes de la concurrence (2)

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

public class CodeCompteur implements Runnable{

    private Compteur c;

    public CodeCompteur(Compteur _c){
        this.c=_c;
    }

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

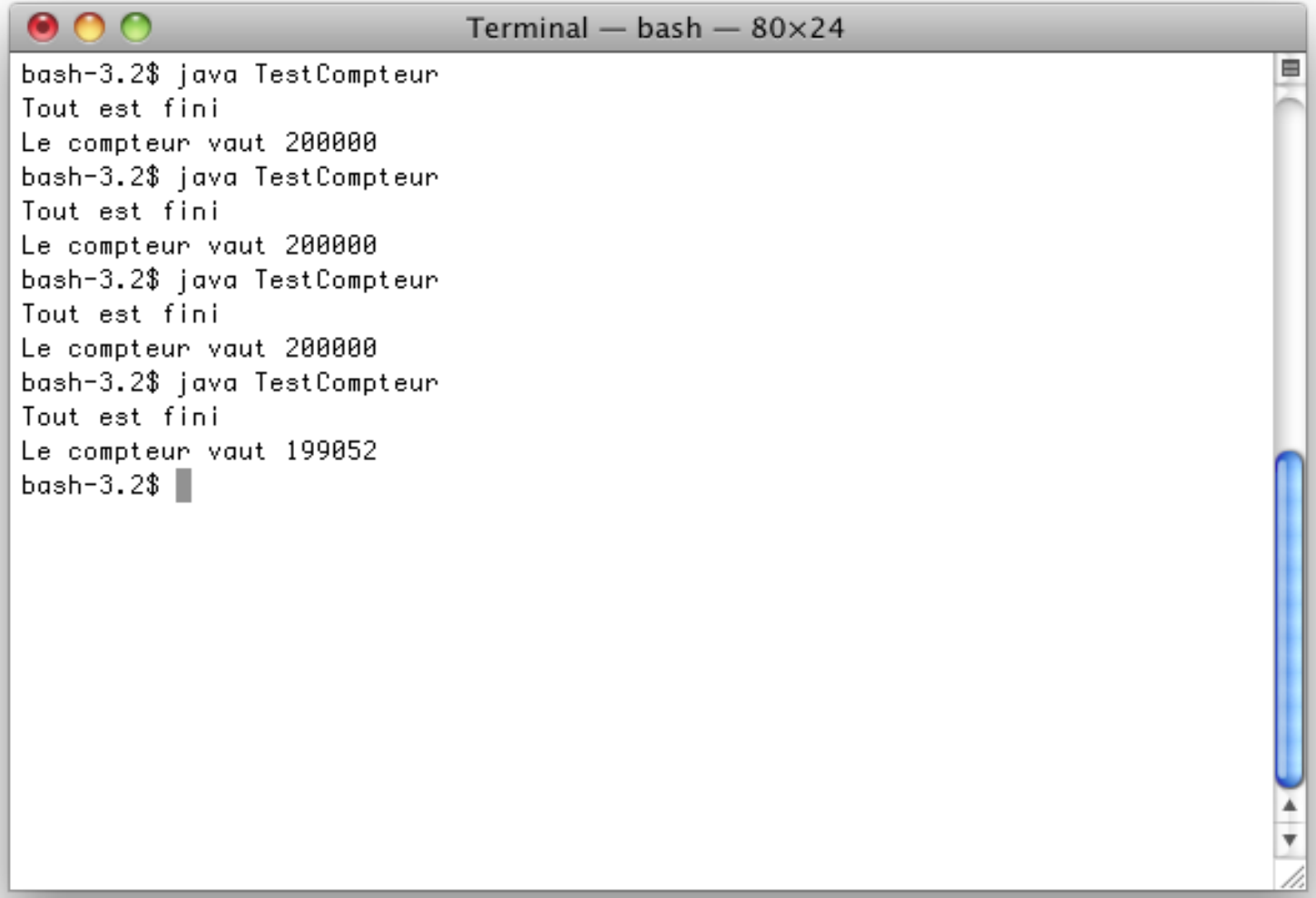
Les problèmes de la concurrence (3)

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

public class TestCompteur {
    public static void main(String[] args){
        try{
            Compteur c=new Compteur();
            CodeCompteur code=new CodeCompteur(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("Tout est fini");
            System.out.println("Le compteur vaut "+c.getValeur());
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }

}
```

Résultat de l'exécution



199052 n'est pas égal à 200000 !!!!!!

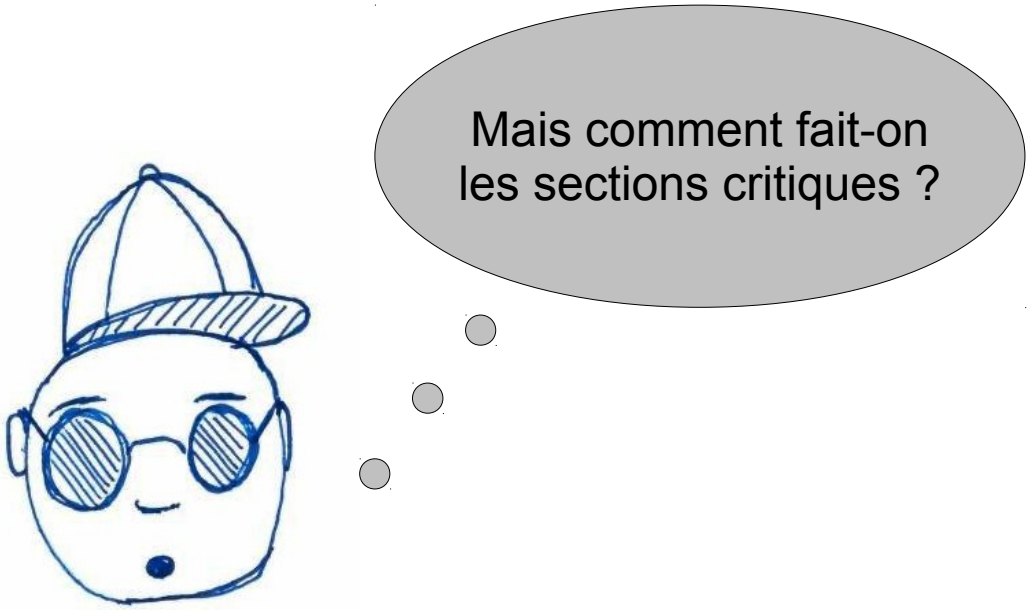
D'où vient le problème

- Le problème est la non atomicité de l'opération c.setValeur(c.getValeur()+1);
  - C'est à dire que plusieurs threads (qui partagent le même compteur) peuvent faire cette opération en même temps
- Scénario possible
  - Thread 1 prend la valeur du compteur (par exemple 0)
  - Thread 2 prend la valeur du compteur (toujours 0)
  - Thread 1 met la valeur du compteur à jour (à 1)
  - Thread 2 met la valeur du compteur à jour (à 1)
  - À ce point les deux threads ont incrémenté le compteur mais ils ne se sont pas mis d'accord pour le faire
- On remarque aussi qu'on a pas le même résultat à chaque exécution

Comment y remédier

- Principe en programmation concurrente
  - On ne peut faire aucune supposition sur l'ordonnancement des exécutions
  - Tout ordre des instructions des processeurs est possible
- Il faut donc prendre des précautions
  - Par exemple s'assurer que lorsque l'on exécute une instruction, il n'y a pas d'autres threads qui exécute la même instruction
  - On rend la suite d'instructions **atomique**
  - On parle alors de partie du code en **section critique**
  - Dans cette section il n'y a à tout moment qu'au plus un thread
  - Si un thread est présent les autres qui veulent accéder à cette partie du code doivent attendre leur tour

Liens avec les flux et le réseau



- Pour faire assurer qu'un seul processus accède à une partie du code (exclusion mutuelle), on utilise un système de verrou
- Le processus qui rentre dans le code ferme le verrou et il le rouvre quand il sort du code pour qu'un autre puisse le fermer de nouveau
- En java, on a le mot clef **synchronized**

Fonctionnement de synchronized

- Le mot clef **synchronized** est utilisé dans le code pour garantir qu'à certains endroits et à un moment donné au plus un processus exécute la portion du code
- Deux utilisations
  - Soit on déclare une méthode **synchronized**
    - **public synchronized int f(int a){...}**
    - À ce moment la méthode **synchronized** d'un même objet ne peut pas être exécuté par deux threads en même temps
  - Soit on verrouille un bloc de code en utilisant
    - **synchronized(objet) {.../*code à verouiller*/...}**
    - ici objet est donné car on utilise le verrou associé à cet objet
    - tout objet possède un verrou

Retour sur notre exemple

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

public class CodeCompteurConcur implements Runnable{

    private Compteur c;

    public CodeCompteur(Compteur _c){
        this.c=_c;
    }

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

- **Attention :** Ne pas synchroniser n'importe quoi
- Synchronizer des parties de codes qui terminent (sinon le verrou reste fermé !!!
- Trouver où mettre les verrous est difficile !!!
- Attention aux deadlocks !

ATTENTION

- Ne pas synchroniser n'importe comment
- Rappeler vous que les verrous sont associés à des objets
  - Deux codes synchronisés sur le même objet ne pourront pas être exécutés en méme temps
- Synchronizer des parties de code qui terminent sinon le verrou reste bloqué pour toujours
- Attention aux deadlocks !!!!
  - Deux thread attendent que le verrou pris par un autre thread se libère
- **Remarque :**
  - synchronized int f(...) {... } est pareil que
  - int f(....) { synchronized(this){...}}
  - Le verrou des méthodes et le verrou de l'objet associé

Un autre problème de la concurrence

- On peut avoir des dépendances entre les sections critiques
- Par exemple, dans le problème des producteurs/consommateurs
  - Les producteurs écrivent une valeur dans une variable
  - Les consommateurs lisent les valeurs écrites dans la variable
  - On ne veut pas qu'un producteur écrase une valeur qui n'a pas été lue
  - On ne veut pas qu'une valeur soit lue deux fois
- Si les consommateurs sont plus rapides que les producteurs, alors les valeurs risquent d'être lues deux fois
- Si les producteurs sont trop rapides, les valeurs d'être perdues
- On ne veut pas que les producteurs et les consommateurs lisent et écrivent en même temps
- Comment faire ?

Première solution (1)

- On crée un objet **VariablePartagee** qui contient une valeur entière **val** et un booléen **pretaecrire**
- Si le boolén est à vrai, on peut écrire une fois la variable et on met le booléen à false
- Si le boolén est à faux, on peut lire une fois la variable et on met le booléen à vrai
- On garantit ainsi que chaque valeur ait écrite une fois et lue une fois

Première solution (2)

```
public class VariablePartagee {
    public int val;
    public boolean pretaecrire;

    public VariablePartagee(){
        val=0;
        pretaecrire=true;
    }

    public int lire(){
        while(pretaecrire==true){}
        pretaecrire=true;
        return val;
    }

    public void ecrire(int v){
        while(pretaecrire==false){}
        pretaecrire=false;
        val=v;
    }
}
```

Code Producteur

```
public class CodeProducteur implements Runnable{

    private VariablePartagee var;

    public CodeProducteur(VariablePartagee _var){
        this.var=_var;
    }

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

Code Consommateur

```
public class CodeConsommateur implements Runnable{

    private VariablePartagee var;

    public CodeConsommateur(VariablePartagee _var){
        this.var=_var;
    }

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

Code Principal

```
public class TestProdCons{
    public static void main(String[] args){
        try{
            VariablePartagee var=new VariablePartagee();
            CodeProducteur prod=new CodeProducteur(var);
            CodeConsommateur cons=new CodeConsommateur(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();
        }
    }
}
```

Problème de cette méthode

- Les méthodes lire et ecrire de la variable partagée doivent être déclarées **synchronized** car elles manipulent le booléen **pretaecrire** et l'entier **val** de façon concurrente
- Cela ne suffit pas, pourquoi :
  - Un consommateur arrive
  - Il fait lire, il prend le verrou et il reste bloqué dans la boucle while en gardant le verrou
  - Si un producteur veut produire il doit prendre le verrou mais il ne peut pas car c'est le consommateur qu'il l'a
  - On doit éliminer cette attente active

Idée de solution

- Mettre un booléen pour savoir si la valeur doit être lue et écrite
- Faire synchronized pour assurer que plusieurs threads ne modifient pas ce booléen en même temps
- Utiliser les méthodes **wait()**, **notify()** et **notifyAll()**
  - **wait()** permet de relacher un verrou que l'on possède et attendre une notification
  - **notify()**/**notifyAll()** permet d'autoriser un/ tous les threads en attente de tenter de reprendre le verrou
    - ATTENTION : il est mieux de posséder le verrou pour faire ces opérations

Les nouvelles méthodes lire/ecrire

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

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