Création et contrôle des threads

* 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

PR - API TCP Java 13

Les méthodes statiques de Thread

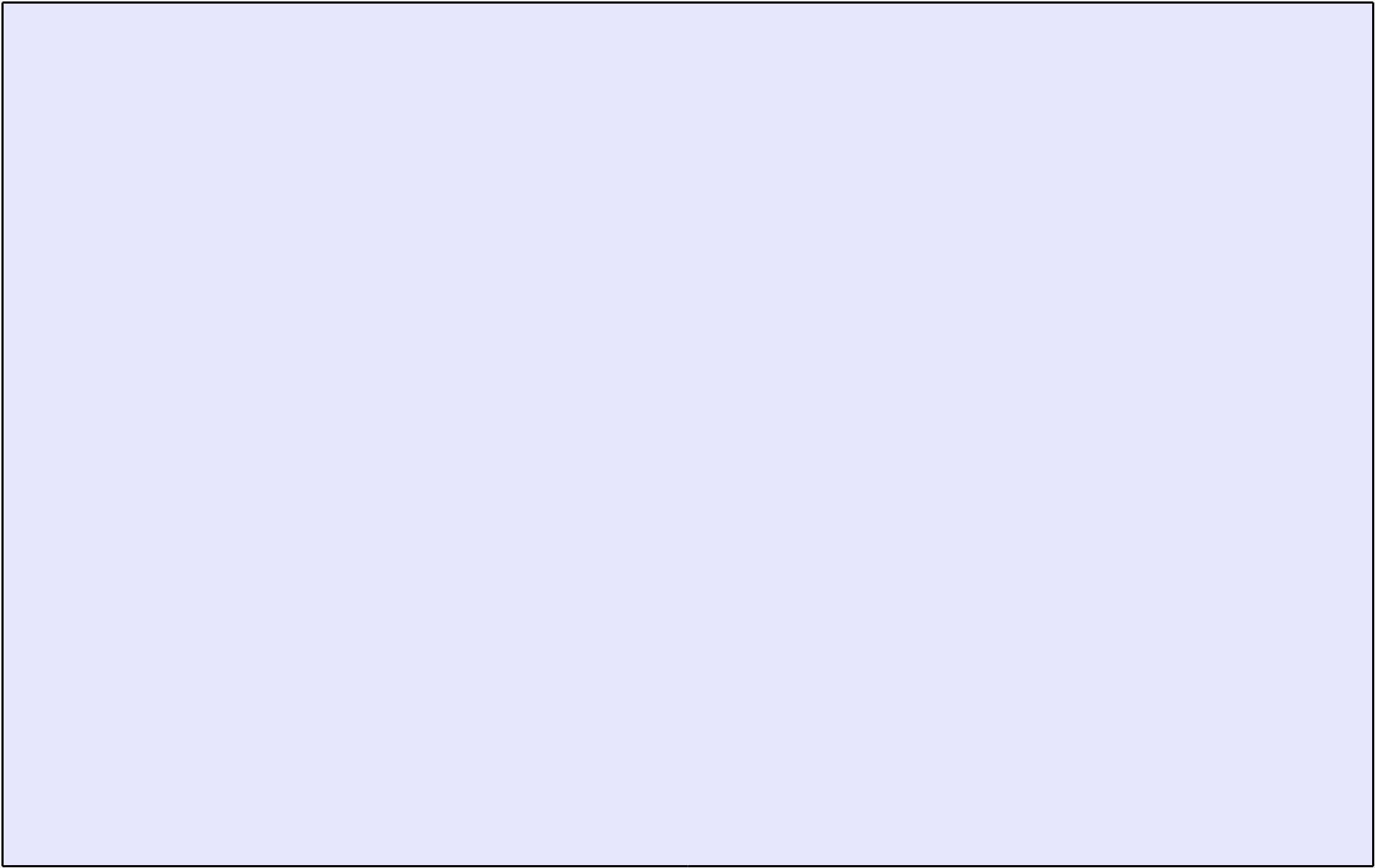
* **Thread currentThread()**
  + Permet de récupérer l'objet Thread courant (qui exécute la ligne)
  + Utile par exemple pour récupérer le nom
* **boolean isInterrupted()**
  + Pour tester si le thread a reçu une demande d'interruption
* **void sleep(long ms)**
  + Pour faire dormir le thread courant 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

PR - API TCP Java 14

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

PR - API TCP Java 18

Structure d'un service

**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.getOutputStream()));**

**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();**

**}**

**}**

**}**

PR - API TCP Java 19

Le serveur concurrent associé

**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();**

**}**

**}**

**}**

PR - API TCP Java 20

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

PR - API TCP Java 28

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

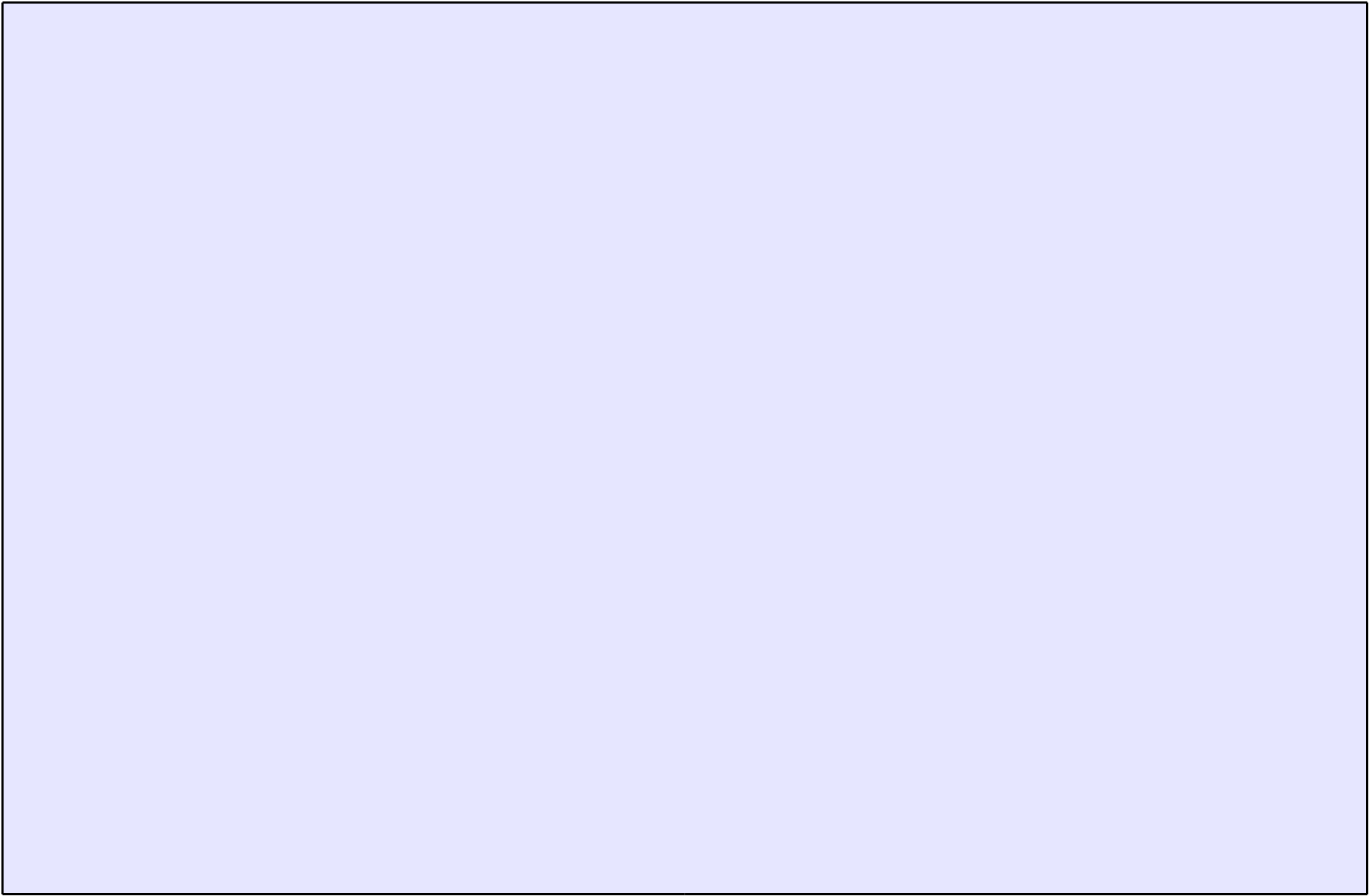
PR - API TCP Java 30

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 ?

PR - API TCP Java 31

Solution avec wait et notify



**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;**

**}**

39