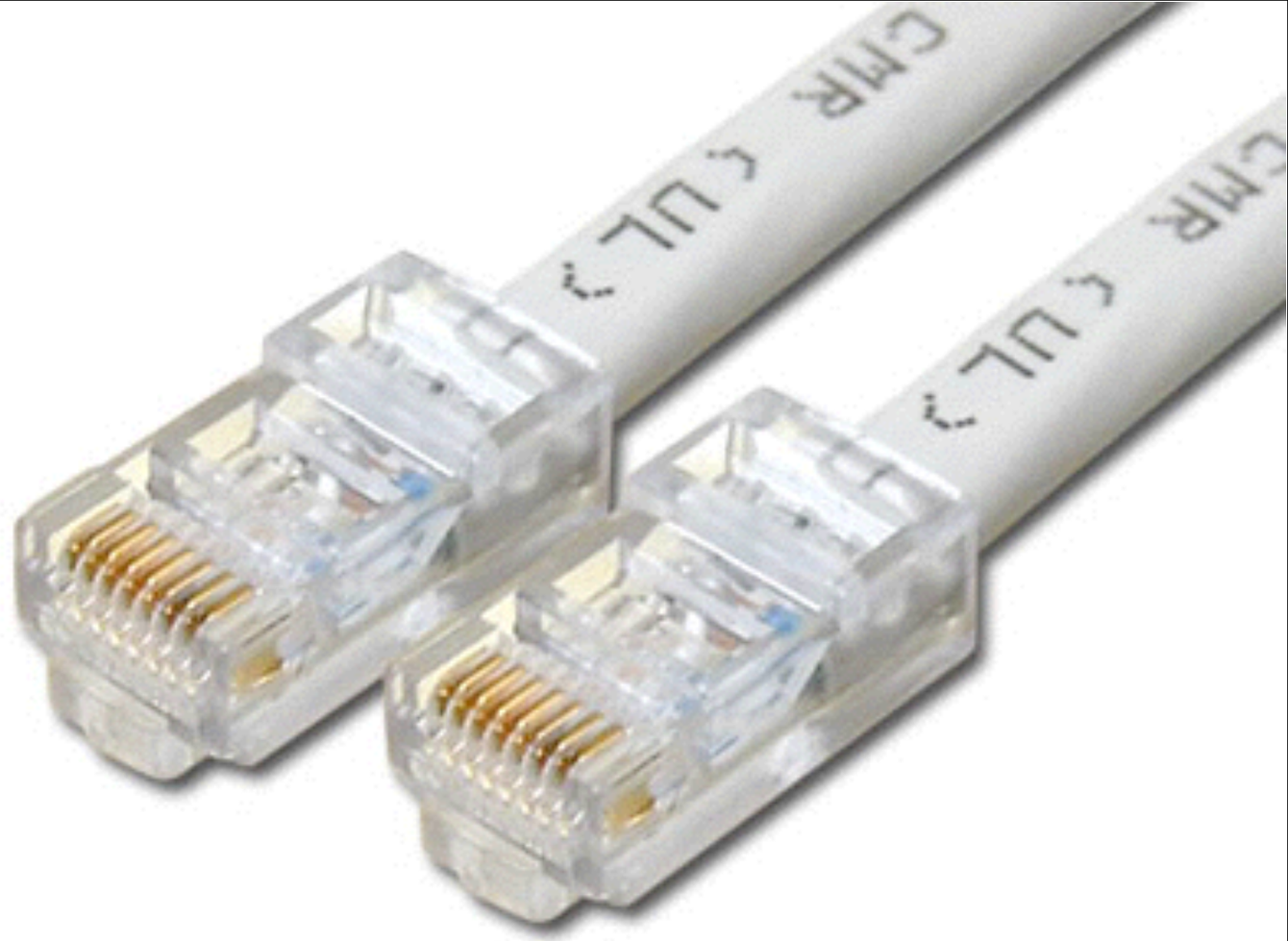
Programmation Réseau

Système d'exécution Java



Jean-Baptiste.Yunes@univ-paris-diderot.fr

UFR Informatique

2013-2014

Les Threads

- on rappelle qu'un **processus** est un **programme** (de nature statique) **en cours d'exécution** (de nature dynamique)
  - son exécution nécessite un environnement (espace d'adressage, objets E/S, etc.)
- un **thread** est un **fil d'exécution** dans un processus donné
- un processus peut être **multi-threadé** (mais au minimum mono-threadé)

- un fil d'exécution est distinct des autres et a pour attributs :
  - un **point courant d'exécution** (pointeur d'instruction, ou PC **program counter**)
  - une **pile d'exécution** (**stack**)
- on notera qu'un `Thread` partage tout le reste de l'environnement avec les autres `Threads` qui lui sont concurrents dans le même processus

- La JVM de Java est multi-threadé et offre au programmeur la possibilité de gérer des threads à sa guise

  - il n’est pas précisé comment ces threads sont pris en charge par le système sous-jacent

- On notera au passage que Java permet aussi de manipuler des processus (lesquels sont pris en charge par le système)

  - il n’y a pas de notion processus dans Java lui-même,un processus est un objet du système hôte

L’environnement d’exécution en Java

- L’environnement d’exécution d’une JVM est disponible dans la JVM elle-même sous la forme d’un objet de type `java.lang.Runtime`
  - attention, il n’existe qu’un seul exemplaire d’un tel objet (Singleton); impossible de créer un objet de cette classe
- L’instance unique peut être retrouvée par appel à la méthode statique

```
Runtime getRuntime();
```

- de nombreuses méthodes sont disponibles dans la classe `Runtime`

- entre autres celles permettant de demander au système hôte de créer un processus concurrent (l'exécution d'un programme en dehors de la JVM elle-même)

- c'est la famille des méthodes

  `Process exec(…);`

- voir la documentation…

Les processus en Java

- les processus dont l'exécution (externe) a été commandée par une JVM sont représentés dans celle-ci sous la forme d'objet `java.lang.Process`

  - ces objets permettent de

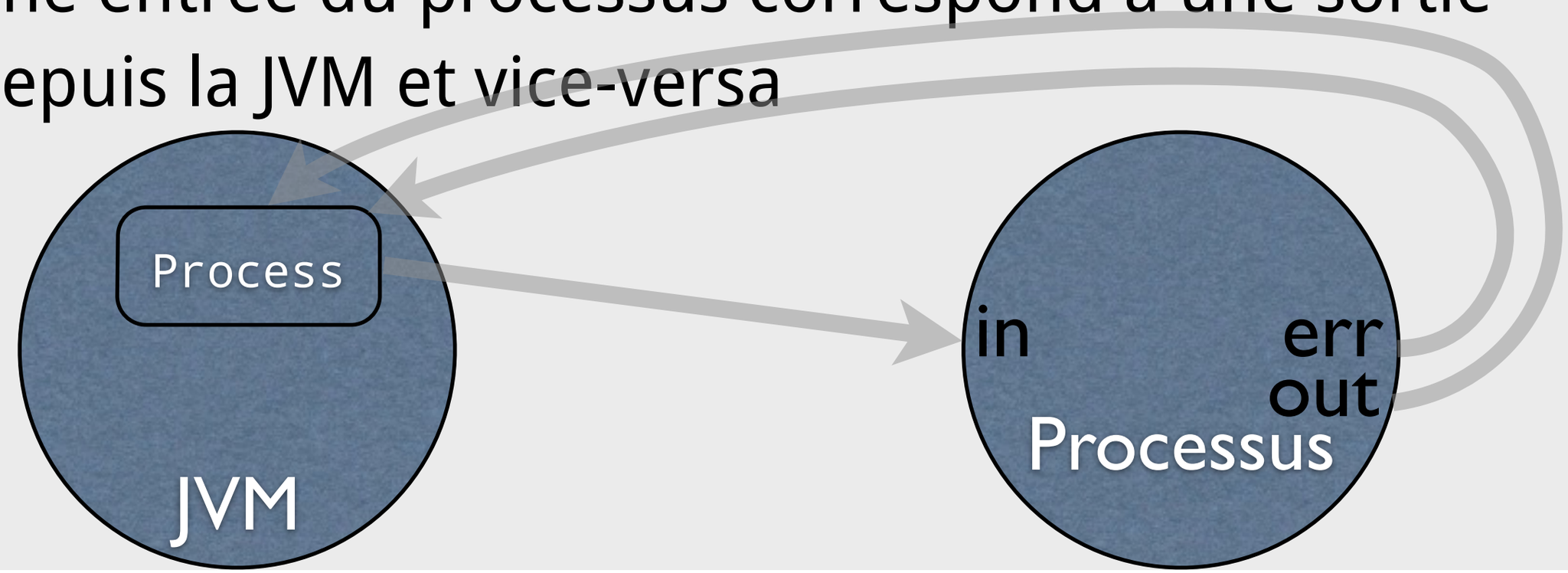
    - communiquer avec les processus externes correspondants

    - se synchroniser avec leur exécution

- on peut obtenir les flux d'entrée et sorties du processus externe avec :

```
InputStream  getErrorStream();
InputStream  getOutputStream();
OutputStream getIntputStream();
```

- une entrée du processus correspond à une sortie depuis la JVM et vice-versa



- il est possible de se synchroniser avec l'exécution du processus externe :

```
int exitValue();
int waitFor();
```

- qui renvoient la valeur de terminaison du processus externe, en mode bloquant avec `waitFor()` et non-bloquant avec `exitValue()`

Les threads en Java

- le mécanisme est plus complexe que celui des processus car il est interne au système Java

- il repose sur deux types d'objets

- les `Runnables`, qui représentent des objets contenant du code qui servira de code principal (équivalent du main pour les threads); la nature statique d'une exécution

- les `Threads`, qui représentent l'exécution elle-même; la nature dynamique d'une exécution

- à tout `Thread` doit être associé un `Runnable`
  - le point d'entrée du code qu'il exécutera…
- bien entendu, à un `Runnable` donné il est possible d'attacher autant de `Threads` que l'on désire

`java.lang.Runnable`

- il s'agit d'une interface qui ne déclare qu'une seule méthode à implémenter :

  `void run()`

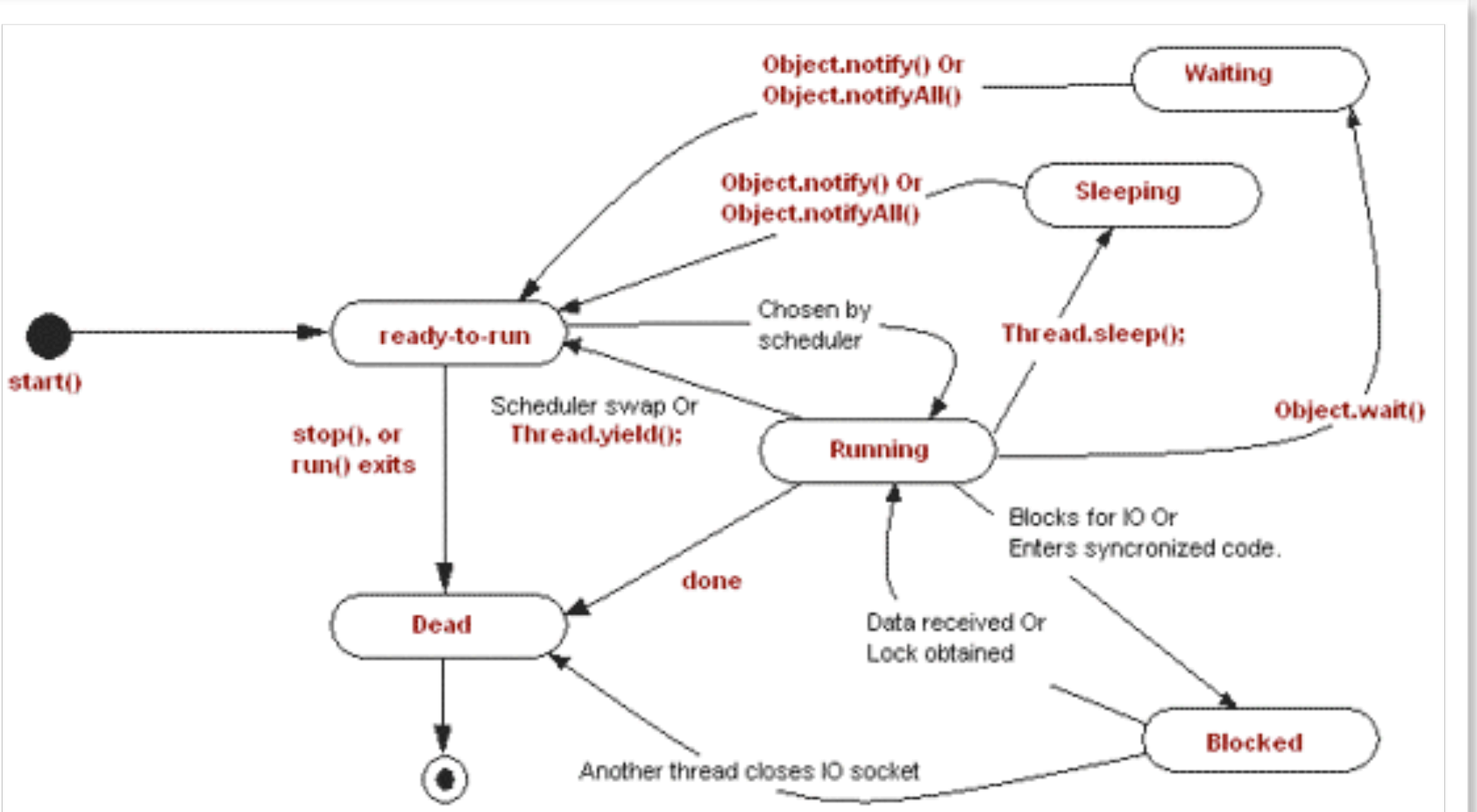
- lorsqu'un `Thread` démarrera, son exécution
  - débutera par un appel à la méthode `run` du `Runnable` qui lui est attaché
  - terminera lorsque cet appel initial terminera

java.lang.Thread

- les Threads Java ont plusieurs **attributs**
  - `String name` [rw] : son nom
  - `long id` [ro] : son identité
  - `int priority` [rw] : sa priorité (les Threads sont ordonnancés à l'aide de cette priorité)
  - `boolean daemon` [rw] : son mode d'exécution (démon ou non, voir plus loin)

- `Thread.State state` [ro] : son état parmi
  - `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`
- sa pile (mais dont on ne peut qu'observer la trace)
- son groupe de Thread
- quelques autres attributs mineurs...

- les états possibles d'un thread et l'automate de transition sont :



bpurcell.org

- Point sur **la terminaison d'une JVM**
  - on indique généralement qu'un programme Java s'arrête lorsqu'on sort du main
    - un programme ne s'arrête pas, seul un processus s'arrête (abus de langage très commun)
    - mais surtout il ne suffit pas sortir du main, encore faut-il sortir du premier appel au main (il est possible de faire des appels récursifs à main)
      - mais en plus, **il faut attendre que TOUS les Thread**s qui ne sont pas démons **s'arrêtent**

- il existe au moins un `Thread` démon :
  - le garbage collector…
- il en existe souvent un autre :
  - le `Thread` de l'interface graphique…

- le contrôle de l'exécution d'un `Thread` s'effectue à travers les méthodes suivantes :
  - `void start()`, qui permet de démarrer le `Thread` (il va alors appeler la méthode `run()` du `Runnable` associé)
  - `void join()`, qui permet d'attendre que le `Thread` s'arrête

- `void interrupt()`, qui permet de positionner le statut du `Thread` à interrompu
  - ce qui n'a aucun effet immédiat direct, sauf terminer certains appels en cours ou permettre au `Thread` concerné de savoir qu'un autre `Thread` souhaite l'interrompre

- Important : **il n'existe pas d'autre technique pour arrêter un** `Thread` que de se débrouiller pour qu'il sorte de son premier appel à `run()`...

- la classe `Thread` possède plusieurs méthodes statiques :
  - `Thread currentThread()`, qui permet de récupérer l'objet `Thread` courant (le « this » du système de `Thread`)
  - `boolean interrupted()`, qui permet de déterminer si le `Thread` courant a reçu une demande d'interruption

- `void sleep(long ms)`
  `void sleep(long ms,long ns)`
  qui permettent au `Thread` courant de renoncer à son exécution pour la durée exprimée (au minimum)

- `void yield()`, qui permet au `Thread` courant de renoncer à la suite de son quota d'exécution et de reprendre une place dans l'ordonnanceur

- la classe `Thread` possède plusieurs constructeurs, dont les plus utiles permettent de lui associer un `Runnable`

- les deux constructeurs les plus fréquemment employés sont :

  - `Thread(Runnable)`

  - `Thread(Runnable,String)`

```
class MyCode implements Runnable {
  public void run() {
    int N = (int)(Math.random()*5);
    for (int i=0; i<10000*N; i++) {
      System.out.println(
          Thread.currentThread().getName()
          +" i="+i);
    }
  }
}
```

```
public class Example {
  public static final int N = 20;
  public static void main(String []args) {
    MyCode code = new MyCode();
    Thread []t = new Thread[N];
   // création des N threads
    for (int i=0; i<N; i++) t[i]=new Thread(code,"T"+i);
   // lancement des N threads
    for (int i=0; i<N; i++) t[i].start();
   // attente de la terminaison des N threads…
    for (int i=N-1; i>=0; i--) {
      try {
        t[i].join();
      } catch(InterruptedException e) {}
      System.out.println("Terminaison de "+t[i].getName());
    }
    System.out.println("tout est fini…");
  }
}
```

- il est à noter qu'un `Thread` implémente l'interface `Runnable` par une méthode `run()` qui ne fait rien

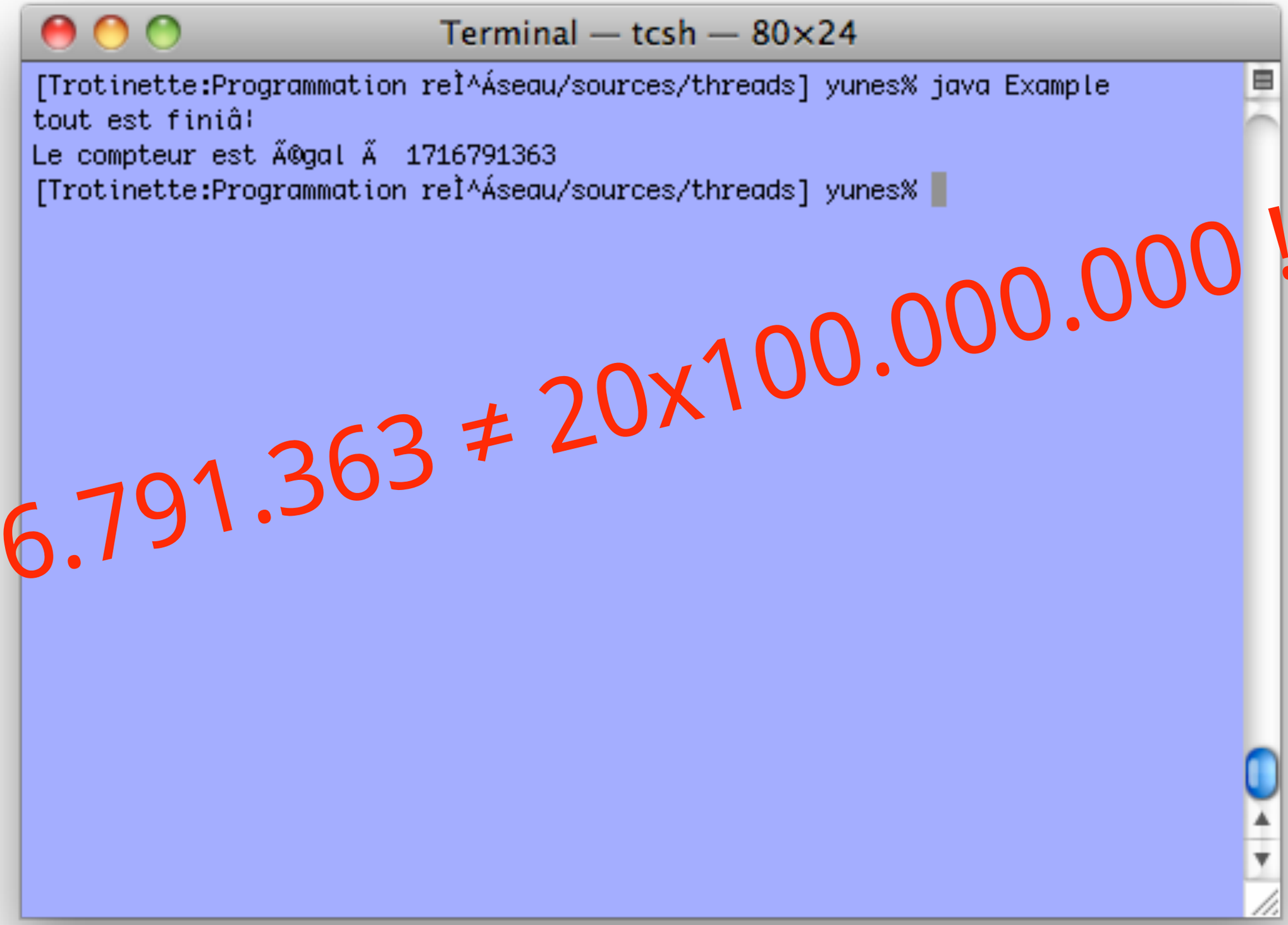
  - ainsi on peut dériver la classe `Thread` et redéfinir la méthode `run()` afin d'obtenir un objet `Thread` qui exécute un code qu'il contient lui-même

Le(s) problème(s) de la concurrence…

```
class Compteur {
    private int valeur;
    public Compteur() { valeur = 0; }
    public int getValeur() { return valeur; }
    public void setValeur(int v) { valeur=v; }
}
```

```
class MyCode implements Runnable {
    private Compteur c;
    public MyCode(Compteur c) { this.c = c; }
    public void run() {
        for (int i=0; i<100000000; i++) {
           c.setValeur(c.getValeur()+1);
        }
    }
}
```

```
public class Example {
    public static final int N = 20;
    public static void main(String []args) {
        Compteur c = new Compteur();
        MyCode code = new MyCode(c);
        Thread []t = new Thread[N];
        for (int i=0; i<N; i++)
            t[i] = new Thread(code,"T["+i+"]");
        for (int i=0; i<N; i++) t[i].start();
        for (int i=0; i<N; i++) {
            try {
                t[i].join();
            } catch(InterruptedException e) {}
        }
        System.out.println("tout est fini…");
        System.out.println("Le compteur est égal à "+
                           c.getValeur());
    }
}
```



1.716.791.363 ≠ 20x100.000.000 !!!

- le problème est la non-atomicité de l'opération d'incrémentation du compteur
  - le compteur est une ressource à laquelle les Threads accèdent concurremment
    - T_i prend la valeur disons v
    - T_i calcule v+1
    - T_i passe la main à T_{i+1}
    - T_{i+1} prend la valeur (c'est encore v)
    - T_{i+1} calcule v+1
    - T_{i+1} range la valeur v+1
    - T_{i+1} passe la main à T_i
    - T_i range la valeur v+1 (à ce point les deux ont bien incrémenté chacun une fois, le malheur est qu'ils ne se sont pas entendus pour le faire correctement)

- Principe en programmation concurrente :
  - on ne doit jamais faire aucune supposition sur l'ordonnancement des exécutions
- par conséquent, il faut être **offensif** :
  - prendre des précautions afin d'éviter les problèmes
    - **défensif** : attendre que le problème arrive pour le corriger

- dans le cas précédent (celui du compteur) la solution est, par exemple, de rendre atomique la suite d'opérations concernée

  - on veut éviter de ne faire qu'une partie de la suite d'opérations : prendre la valeur, ajouter un, stocker le résultat

  - une telle suite d'opération est appelée une **section critique**

  - on souhaite donc contrôler quels sont les Threads autorisés à entrer dans la section critique (penser à un sas, un portillon de contrôle)

- le mécanisme usuel permettant de contrôler l'entrée d'une section critique est le **verrou** qui assure une propriété d'**exclusion mutuelle**

  - un verrou est posé avant l'entrée en section critique et déposé ensuite

    - la pose assure qu'une seule demande ne peut être effectivement servie, les autres demandes sont placées en attente

- en Java il existe le mot-clé `synchronized` qui permet de verrouiller une section critique

- deux utilisations possible de synchronized
  - on peut verrouiller une méthode entière, auquel cas le mot-clé est employé dans la signature de la méthode
  - on peut verrouiller un bloc de code en particulier par emploi de

```
synchronized(o) {
    …
}
```

- où o est l'objet (de type quelconque) représentant le verrou

- une méthode synchronisée

```
synchronized … méthode(…) {
        …
}
```

- correspond exactement à

```
… méthode(…) {
    synchronized(this) {
        …
    }
}
```

```
class MyCode implements Runnable {
    private Compteur c;
    public MyCode(Compteur c) { this.c = c; }
    public void run() {
        for (int i=0; i<100000000; i++) {
            synchronized(c) {
                c.setValeur(c.getValeur()+1);
            }
        }
    }
}
```

- Attention à ne pas « synchroniser » n'importe quoi
- Note : on remarquera qu'une forte pénalité est appliquée à l'exécution; les verrous ont un coût non négligeable…
- Il est en général difficile de cerner la bonne granularité

Un autre problème de concurrence

- une section critique n'est pas toujours, pour son bon fonctionnement, isolée du reste de l'application

  - en particulier, on peut vouloir garantir l'unicité en section critique mais aussi assurer une interdépendance entre sections critiques

- le problème des producteurs/consommateurs

- songeons au dynamisme d'une structure tripartite :
  - les producteurs produisent des objets et les stockent
  - le stockage possède une capacité limitée
  - les consommateurs retirent des objets du stockage pour les utiliser à leur propre fin
- on peut remarquer qu'il s'agit d'un modèle de communication à canal de capacité limitée

- le problème est dû aux vitesses relatives des opérations de consommation et production :

  - si les producteurs sont plus rapides que les consommateurs, il va falloir assurer que lorsque le stockage est plein, la chaîne de production soit mise en attente…

  - si les consommateurs sont plus rapides que les producteurs, il va falloir assurer l'attente des consommateurs lorsque le stockage est vide

```
class Produit {
    private String nom;
    public Produit(String nom) {
        this.nom = nom;
    }
    public String toString() {
        return nom;
    }
}
```

```
class Stock {
    private Produit []leStock;
    private int niveauCourant;
    private static final int capacitéMaximale = 10;
    public Stock() {
        leStock = new Produit[capacitéMaximale];
        niveauCourant = 0;
    }
    public boolean addProduit(Produit p) {
        if (niveauCourant==capacitéMaximale) return false;
        leStock[niveauCourant++] = p;
        return true;
    }
    public Produit removeProduit() {
        if (niveauCourant==0) return null;
        Produit p = leStock[niveauCourant-1];
        leStock[--niveauCourant] = null;
        return p;
    }
}
```

```
class Producteur extends Thread {
  private Stock stock; private String nomProduit; private Random random;
  Producteur(Stock stock,String nomProduit) {
    this.stock = stock; random = new Random(); this.nomProduit = nomProduit;
  }
  public void run() {
    while (true) {
      Produit p = new Produit(nomProduit);
      try {
        Thread.sleep(random.nextInt(1000)+1000);
      } catch(InterruptedException e) {}
      if (!stock.addProduit(p)) {
        do {
          try {
            System.out.println("Producteur "+getId()+" plein");
            Thread.sleep(random.nextInt(100)+100);
            } catch(InterruptedException e) {}
          } while (!stock.addProduit(p));
        }
      System.out.println("Producteur "+getId()+" a rajoute "+p);
    }
  }
}
```

```
class Consommateur extends Thread {
  private Stock stock; private Random random;
  public Consommateur(Stock stock) {
    random = new Random(); this.stock = stock;
  }
  public void run() {
    while (true) {
      try {
        Thread.sleep(random.nextInt(10000)+1000);
      } catch(InterruptedException e) {}
      Produit p = stock.removeProduit();
      if (p==null) {
        do { try {
              System.out.println("Consommateur "+getId()+ " vide");
              Thread.sleep(random.nextInt(100)+100);
            } catch(InterruptedException e) {}
            p = stock.removeProduit();
        } while (p==null);
      }
      System.out.println("Consommateur "+getId()+" a enlevé "+p);
    }
  }
}
```

```
public class ProdCons {
    public static void main(String []args) {
        Stock stock = new Stock();
        Producteur []p = new Producteur[2];
        p[0] = new Producteur(stock,"banane");
        p[1] = new Producteur(stock,"carambar");
        p[0].start();
        p[1].start();
        Consommateur []c = new Consommateur[3];
        c[0] = new Consommateur(stock);
        c[1] = new Consommateur(stock);
        c[2] = new Consommateur(stock);
        c[0].start();
        c[1].start();
        c[2].start();
    }
}
```

- deux problèmes dans ce code :
  - la non-atomicité des opérations d'ajout et retrait du stock
  - l'attente active

- nous savons corriger le problème d'atomicité de l'accès au stock :
  - `synchronized`

- le problème subsistant est que le mécanisme mis en place encourage l'**attente active**
  - une boucle testant sans arrêt si la condition de continuation est vraie ou non…
    - très consommatrice en ressources (CPU)
- pour empêcher cette attente active, nous avons un problème :
  - nous souhaitons nous mettre en attente que la condition recherchée soit réalisée alors que nous avons la clé

- pour se faire, nous allons employer les méthodes `wait()` et `notify()` héritées d'`Object`

  - `wait()` permet de relâcher un verrou que l'on possède et de se mettre en attente jusqu'a ce que quelqu'un nous autorise à tenter de le reprendre

  - `notify()` / `notifyAll()` permet d'autoriser un (resp. tous les) thread en attente de tenter de reprendre le verrou qu'il possédait

  - Attention : il FAUT déjà détenir le verrou pour effectuer ces opérations

```
class Stock {
  private Produit []leStock;
  private int niveauCourant;
  private static final int capacitéMaximale = 10;
  public Stock() {
    leStock = new Produit[capacitéMaximale];
    niveauCourant = 0;
  }
  public synchronized void addProduit(Produit p) {
    while (niveauCourant==capacitéMaximale) {
      System.out.println("plein!");
      try {
        wait();
      } catch(InterruptedException e) {}
    }
    leStock[niveauCourant++] = p;
    notifyAll();
  }
// to be continued...
```

```
  public synchronized Produit removeProduit() {
    while (niveauCourant==0) {
      System.out.println("vide!");
      try {
        wait();
      } catch(InterruptedException e) {}
    }
    Produit p = leStock[niveauCourant-1];
    leStock[--niveauCourant] = null;
    notifyAll();
    return p;
  }
} // fin classe Stock
```

```
class Producteur extends Thread {
  private Stock stock;
  private String nomProduit;
  private Random random;
  Producteur(Stock stock,String nomProduit) {
    this.stock = stock;
    random = new Random();
    this.nomProduit = nomProduit;
  }
  public void run() {
    while (true) {
      Produit p = new Produit(nomProduit);
      try {
        Thread.sleep(random.nextInt(1000)+1000);
      } catch(InterruptedException e) {}
      stock.addProduit(p);
      System.out.println("Producteur "+getId()+" a rajoute "+p);
    }
  }
}
```

```
class Consommateur extends Thread {
  private Stock stock;
  private Random random;
  public Consommateur(Stock stock) {
    random = new Random();
    this.stock = stock;
  }
  public void run() {
    while (true) {
      try {
        Thread.sleep(random.nextInt(10000)+1000);
      } catch(InterruptedException e) {}
      Produit p = stock.removeProduit();
      System.out.println("Consommateur "+getId()
                         +" a enlevé "+p);
    }
  }
}
```

java.util.concurrent

- depuis la version 1.5, il existe trois packages très complets contenant des mécanismes de gestion de la concurrence

  - `java.util.concurrent` qui contient (entre autres) des sémaphores, des futurs, des barrières, etc.

  - `java.util.concurrent.atomic` qui contient diverses classes de nombres autorisant des opérations atomiques de modification

  - `java.util.concurrent.locks` qui contient divers types de verrous