

# Réseaux IP : IHM

## Licence Informatique 3<sup>ème</sup> année

### Compléments de programmation Java : Flux & Threads

Eric Cariou

Université de Bretagne Occidentale  
UFR Sciences & Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

1

### Flux Java

- ◆ En Java, toutes les entrées/sorties sont gérées via des flux
  - ◆ Entrées/sorties standards (clavier/console)
  - ◆ Fichiers
  - ◆ Sockets
  - ◆ ...
- ◆ Flux : tuyaux dans lesquels on envoie ou lit des séries de données
  - ◆ Information de base qui transite dans un flux : l'octet

3

### Hierarchie de flux Java

- ◆ Java définit une hiérarchie de flux composée de plusieurs dizaines de classes (de types de flux différents)
  - ◆ Package `java.io`
- ◆ Deux classifications transverses
  - ◆ Flux est soit d'entrée, soit de sortie
    - ◆ Entrée : le programme lit des informations à partir du flux
    - ◆ Sortie : le programme écrit des informations dans le flux
  - ◆ Nature de l'information transitant sur le flux
    - ◆ Binaire : octet par octet
    - ◆ Caractère : 2 octets par 2 octets
      - ◆ Codage unicode sur 16 bits

5

### Flux Java

2

### Flux Java standards

- ◆ Flux d'entrées/sortie standards
  - ◆ `System.out`
    - ◆ Sortie standard, flux de type `PrintStream`
    - ◆ `System.out.println('nombre = '+nb);`
  - ◆ `System.err`
    - ◆ Sortie d'erreur standard, flux de type `PrintStream`
  - ◆ `System.in`
    - ◆ Entrée standard, flux de type `InputStream`
    - ◆ 

```
while ((c = (char)System.in.read()) != 'z')
    System.out.print(c);
```

4

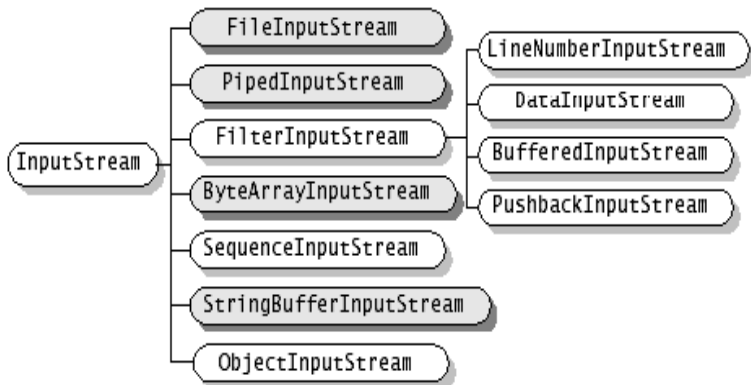
### Hierarchie de flux Java

- ◆ Hiérarchie principale
  - ◆ Flux de base
  - ◆ Flux avec tampon
  - ◆ Flux d'accès aux fichiers
  - ◆ Flux de filtrage
  - ◆ Flux d'impression
  - ◆ Flux enchaînés par des « pipes »
  - ◆ Flux de concaténation de plusieurs flux en un seul
  - ◆ Flux de conversion flux caractère/flux binaire
  - ◆ Flux de lecture/écriture de différents types
    - ◆ `int`, `char` ... ou bien encore un objet quelconque (`Object`)
    - ◆ Données codées indépendamment de la plateforme/système

6

## Hierarchie de flux Java

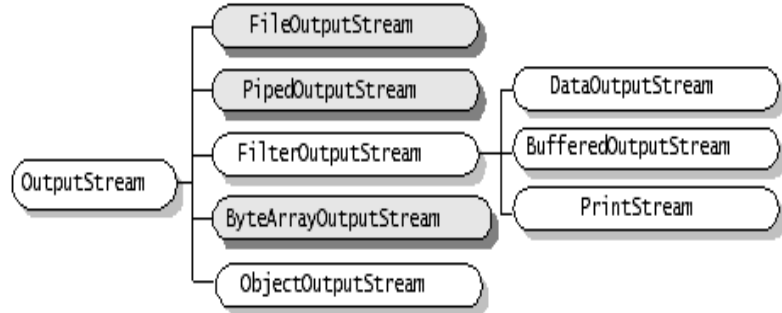
### ◆ Flux binaire, entrée



7

## Hierarchie de flux Java

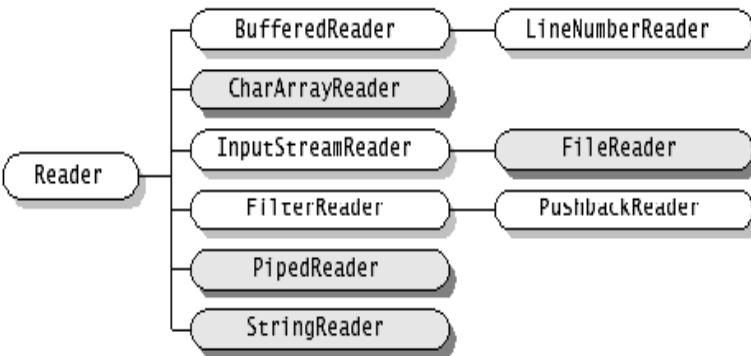
### ◆ Flux binaire, sortie



8

## Hierarchie de flux Java

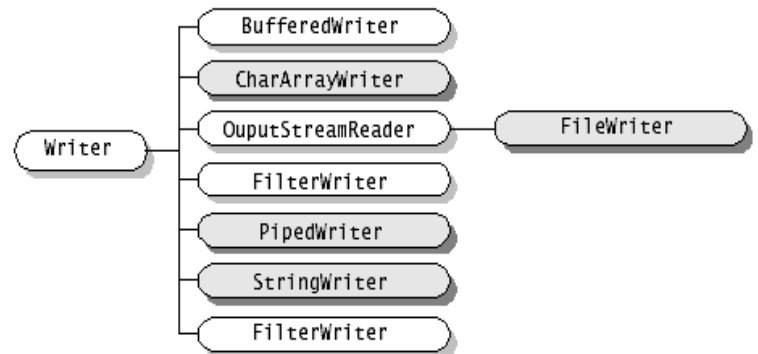
### ◆ Flux caractère, entrée



9

## Hierarchie de flux Java

### ◆ Flux caractère, sortie



10

## Hierarchie de flux Java

### ◆ Autres types de flux

#### ◆ Package java.util.zip

- ◆ Compression données : GZIPInputStream, ZipInputStream ...
- ◆ Vérification intégrité données (CRC) : CheckedInputStream ...

#### ◆ Package javax.crypto

- ◆ Cryptage des données : CipherInputStream ...

#### ◆ Et d'autres ...

### ◆ Les flux peuvent être dépendants les uns des autres

- ◆ Un flux est créé à partir d'un autre (par « wrapping ») : il traite les mêmes données mais avec un traitement supplémentaire

- ◆ Codage des données dans un autre type
- ◆ Filtrage des données, mise en tapon ...

- ◆ Un flux est chaîné à un autre par un pipe

11

## Méthodes des classes Stream

### ◆ Méthodes générales d'accès aux données du flux

#### ◆ Flux en entrée (InputStream)

- ◆ `int read()`
  - ◆ Lecture d'un octet (sous forme de int) dans le flux
- ◆ `int read(byte[] tab)`
  - ◆ Lit une suite d'octets en les plaçant dans tab
  - ◆ Lit au plus la longueur de tab
  - ◆ Retourne le nombre d'octets lu
- ◆ Autres méthodes pour se placer à un endroit donné du flux ...
- ◆ `int available()`
  - ◆ Retourne le nombre d'octets disponibles en lecture dans le flux
- ◆ `void close()`
  - ◆ Ferme le flux

12

## Méthodes des classes Stream

### ◆ Méthodes générales d'accès aux données du flux

#### ◆ Flux en sortie (OutputStream)

- ◆ `void write(int b)`
  - ◆ Écrit un octet (via un int) dans le flux
- ◆ `void write(byte[])`
  - ◆ Écrit le contenu d'un tableau d'octets dans le flux
- ◆ `void flush()`
  - ◆ Force l'écriture dans le flux de toutes les données à écrire
    - ◆ Vide le tampon associé au flux en écrivant son contenu
- ◆ `void close()`
  - ◆ Ferme le flux

#### ◆ Flux en entrées ou sorties

- ◆ Méthodes générales : accès niveau octet

13

## Méthodes des classes Stream

### ◆ Data[Input/Output]Stream (suite)

- ◆ Exemple : écriture d'un objet de la classe `Personne` (classe programmée n'appartenant pas à la hiérarchie Java)

```
◆ Personne pers = new Personne ("toto", 24);  
ObjectOutputStream output = ... ;  
output.writeObject(pers);
```

#### ◆ Pour pouvoir envoyer un objet dans un flux

- ◆ Sa classe doit implémenter l'interface `java.io.Serializable`
- ◆ Interface vide qui sert juste à préciser qu'on autorise les objets de cette classe à être sérialisés
  - ◆ C'est-à-dire pouvant être transformés en série de byte et donc transmissibles via des flux

#### ◆ Exceptions niveau flux : problème d'entrée/sortie

- ◆ La plupart des méthodes d'accès aux flux peuvent lever l'exception `java.io.IOException`

15

## Exemple utilisation de flux Java

### ◆ Exemple d'utilisation de flux

#### ◆ Écriture d'entiers dans un fichier

```
// ouverture d'un flux en sortie sur le fichier entiers.bin  
FileOutputStream ficOut =  
    new FileOutputStream("entiers.bin");
```

```
// ouverture d'un flux de données en sortie à partir de ce flux  
DataOutputStream dataOut =  
    new DataOutputStream(ficOut);
```

```
// écriture des entiers de 10 à 15 dans le fichier  
for(int i=10;i<16;i++)  
    dataOut.writeInt(i);
```

```
// fermeture des flux  
dataOut.close();  
ficOut.close();
```

17

## Méthodes des classes Stream

### ◆ Classes de flux spécialisées

- ◆ Offrent des méthodes d'accès plus évoluées que niveau octet

#### ◆ Deux types de flux intéressants de ce point de vue

- ◆ `Data[Input/Output]Stream`
  - ◆ Lecture/écriture de types primitifs Java
    - ◆ `int`, `char`, `boolean`, `double`, `long`, `byte`, `float`, `short`
  - ◆ Exemple pour `double`
    - ◆ `DataOutputStream: void writeDouble(double b)`
    - ◆ `DataInputStream: double readDouble()`
- ◆ `Object[Input/Output]Stream`
  - ◆ Lecture/écriture d'objets de toute nature
    - ◆ Très puissant et confortable
  - ◆ `ObjectOutputStream: void writeObject(Object o)`
  - ◆ `ObjectInputStream: Object readObject()`

14

## Méthodes des classes Stream

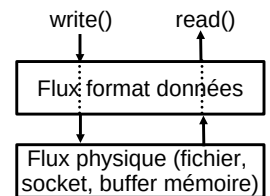
### ◆ Constructions des flux : fonctionnement classique

- ◆ Créer/récupérer un flux associé à une ressource physique (fichier, zone mémoire, socket ...)

- ◆ Passer ce flux en paramètre du constructeur d'un flux de type données (`Data/Object`) pour gérer le type des données transitant par la ressource physique

- ◆ Exemple : créer un `ObjectOutputStream` à partir d'un `FileOutputStream` associé au fichier `test.bin`

```
◆ FileOutputStream fileOut=  
    new FileOutputStream("test.bin");  
ObjectOutputStream objOut =  
    new ObjectOutputStream(fileOut);  
// peut maintenant enregistrer tout objet dans test.bin via objOut 16
```



## Exemple utilisation de flux Java

### ◆ Exemple d'utilisation de flux

#### ◆ Lecture d'entiers à partir d'un fichier

```
// ouverture d'un flux en entrée sur le fichier entiers.bin  
FileInputStream ficIn =  
    new FileInputStream("entiers.bin");
```

```
// ouverture d'un flux de données en entrée à partir de ce flux  
DataInputStream dataIn = new DataInputStream(ficIn);
```

```
// tant que des données sont disponibles, on lit des entiers  
while(dataIn.available() > 0)  
    System.out.println(dataIn.readInt());
```

```
// fermeture des flux  
dataIn.close();  
ficIn.close();
```

18

## Concurrence dans une application

### Threads Java

19

#### Threads en Java

- ◆ Pour créer et lancer un nouveau thread, 2 modes
  - ◆ Étendre la classe `java.lang.Thread`
    - ◆ Redéfinir la méthode `public void run()`
      - ◆ Qui contient la séquence de code qu'exécutera le thread
    - ◆ Pour lancer le thread
      - ◆ Instancier normalement la classe définie
      - ◆ Appeler ensuite la méthode `start()` sur l'objet créé
  - ◆ Implémenter l'interface `java.lang.Runnable`
    - ◆ Définir la méthode `public void run()` de cette interface
      - ◆ Qui contient la séquence de code qu'exécutera le thread
    - ◆ Pour lancer le thread
      - ◆ Instancier normalement la classe définie
      - ◆ Créer une instance de la classe `Thread` en passant cet objet en paramètre
      - ◆ Lancer la méthode `start()` du thread instancié

21

#### Thread Java – exemple

- ◆ Lancement du calcul des factoriels de 1 à 10 en parallèle
  - ◆ ...

```
CalculFactoriel cf;
for (int i=10; i >= 1; i--) {
    cf = new CalculFactoriel(i);
    cf.start();
}
...
```
- ◆ Deux phases pour lancer un calcul
  - ◆ On instancie normalement la classe `CalculFactoriel`
  - ◆ On appelle la méthode `start()` sur l'objet créé
    - ◆ La séquence d'instructions de la méthode `run()` de la classe `CalculFactoriel` est exécutée dans un nouveau thread

## Concurrence

- ◆ Java offre nativement un mécanisme permettant de gérer des flux d'exécution parallèle
  - ◆ Les threads
- ◆ Rappel différence processus/thread
  - ◆ Le processus est créé comme une copie d'un processus existant
    - ◆ Deux processus distincts avec leur zone mémoire propre
  - ◆ Le thread s'exécute au sein d'un processus existant
    - ◆ Nouveau flux d'exécution interne
    - ◆ Partage des données du processus

20

#### Thread Java – exemple

- ◆ Classe `CalculFactoriel` calcule un factoriel et affiche le résultat à l'écran
  - ◆ Via un thread à part

```
public class CalculFactoriel extends Thread {

    protected int nb;

    public void run(){
        int res = 1;
        for (int i=1; i<=nb; i++)
            res = res * i;
        System.out.println("factoriel de "+nb+"="+res);
    }

    public CalculFactoriel(int nb) {
        this.nb = nb;
    }
}
```

22

#### Thread Java – variante exemple

- ◆ Même exemple mais sans spécialiser la classe `Thread`
- ◆ Implémentation de l'interface `Runnable`

```
public class CalculFactoriel implements Runnable {

    protected int nb;

    public void run(){
        int res = 1;
        for (int i=1; i<=nb; i++)
            res = res * i;
        System.out.println("factoriel de "+nb+"="+res);
    }

    public CalculFactoriel(int nb) {
        this.nb = nb;
    }
}
```

24

## Thread Java – variante exemple

### ◆ Lancement des threads

```
◆ ...
CalculFactoriel cf;
for (int i=10; i >= 1; i--) {
    cf = new CalculFactoriel(i);
    (new Thread(cf)).start();
}
...

```

- ◆ On lance un Thread générique qui exécutera la méthode `run()` de l'objet de type `Runnable` passé en paramètre du constructeur

25

## Thread Java – création

### ◆ 2 méthodes pour créer et exécuter un thread

- ◆ Laquelle choisir ?
- ◆ A priori peu de différence
- ◆ Sauf dans le cas où la classe doit hériter d'une autre classe
  - ◆ Doit alors forcément utiliser la méthode avec l'interface `Runnable`

26

## Thread Java – résultat exemple

### ◆ (un) résultat de l'exécution du programme

```
◆ factoriel de 10=3628800
factoriel de 9=362880
factoriel de 8=40320
factoriel de 7=5040
factoriel de 6=720
factoriel de 5=120
factoriel de 4=24
factoriel de 3=6
factoriel de 2=2
factoriel de 1=1

```

- ◆ Les résultats des calculs sont affichés dans l'ordre de leur lancement
  - ◆ Pourtant les calculs de petites valeurs sont normalement plus courts car moins de passages dans la boucle ... 27

27

## Ordonnancement des threads

### ◆ Ordonnancement des processus/threads

- ◆ Sur une machine, nombre de flots d'exécution en réel parallélisme = nombre de processeurs ou cores
- ◆ Les processus/threads doivent partager les supports d'exécution pour s'exécuter
- ◆ Pour simuler un parallélisme d'exécution avec un seul processeur
  - ◆ Un processus/thread n'est pas exécuté du début à la fin en une seule étape
  - ◆ Un processus/thread exécute une partie de ses instructions pendant un temps donné avant de passer la main à un autre processus
  - ◆ Plus tard, il retrouvera la main et continuera son exécution

28

## Ordonnancement des threads

### ◆ Dépendance thread/processus

- ◆ Un thread est créé par et dans un processus
- ◆ Selon le système d'exploitation, l'ordonnancement se fait
  - ◆ Uniquement au niveau processus
    - ◆ Le système s'occupe de gérer uniquement le parallélisme des processus
    - ◆ Un processus gère en interne l'ordonnancement de ses propres threads
  - ◆ Au niveau de tous les threads et processus
    - ◆ Les threads des processus et les processus sont ordonnancés par le système
- ◆ Approche mixte
  - ◆ L'ordonnancement se fait au niveau processus mais certains threads particuliers peuvent être ordonnancés par le système au même niveau que les processus

29

## Ordonnancement des threads

### ◆ Deux types d'ordonnancement par le système (ou par le processus pour ordonnancer ses threads)

- ◆ Préemptif
  - ◆ Le système interrompt l'exécution des processus/threads pour partager l'accès au processeur
  - ◆ Le système décide quel est le prochain processus/thread qui continuera son exécution
- ◆ Coopératif
  - ◆ Un processus/thread ne libère le processeur que
    - ◆ Quand il est bloqué momentanément (entrée/sortie ...)
    - ◆ De sa propre initiative
  - ◆ Le système décide alors quel est le prochain processus/thread qui continuera son exécution

30

## Ordonnancement des threads Java

- ◆ Ordonnancement des threads en Java
  - ◆ Exécution d'une machine virtuelle Java
    - ◆ Via un processus du système d'exploitation
    - ◆ Qui exécute plusieurs threads Java
      - ◆ Le thread principal
        - ◆ Correspondant au `static void main(String argv[])`
      - ◆ Les threads créés par le programme
      - ◆ Les threads gérant l'interface graphique
      - ◆ Garbage collector ...
  - ◆ Particularité de Java
    - ◆ Langage multi-plateformes (Windows, Linux, MacOS, Solaris, ...)
    - ◆ L'ordonnancement des processus/threads dépend du système d'exploitation

31

## Ordonnancement des threads

- ◆ Les threads peuvent avoir des priorités différentes
  - ◆ Un thread plus prioritaire a la main en priorité
    - ◆ Si un thread de plus haute priorité que le thread courant actif veut la main, il la récupère alors de suite via un ordonnancement préemptif
- ◆ Accès aux priorités, méthodes de la classe Thread
  - ◆ `public int getPriority()` : retourne le niveau de priorité du thread
  - ◆ `public void setPriority(int priority)` : change le niveau de priorité du thread
- ◆ Trois constantes de la classe Thread pour définir les priorités
  - ◆ `MAX_PRIORITY` : niveau de priorité maximal possible (10)
  - ◆ `MIN_PRIORITY` : niveau de priorité minimal possible (1)
  - ◆ `NORM_PRIORITY` : niveau de priorité par défaut (5)

33

## Thread Java – nouveau résultat exemple

- ◆ (un) résultat d'exécution de l'exemple après la modification
  - ◆ 

```
factoriel de 1=1
factoriel de 2=2
factoriel de 3=6
factoriel de 4=24
factoriel de 5=120
factoriel de 6=720
factoriel de 7=5040
factoriel de 8=40320
factoriel de 9=362880
factoriel de 10=3628800
```
- ◆ Bien que lancés en dernier, les calculs les plus courts se terminent en premier
  - ◆ Ordonnancement plus « naturel » que le précédent
    - ◆ Correspond à ce que l'on aurait avec un parallélisme physique complet
  - ◆ Mais aurait pu avoir un ordre moins « parfait »
    - ◆ 1, 2, 4, 3, 5, 7, 6, 8, 9, 10 par exemple

35

## Ordonnancement des threads Java

- ◆ Principe fondamental
  - ◆ On ne doit pas se baser sur un modèle d'ordonnancement particulier pour développer une application multi-threadée en Java
- ◆ Par principe, on considérera le modèle le plus contraignant
  - ◆ Généralement c'est l'ordonnancement coopératif des threads Java
    - ◆ Mais le préemptif peut aussi poser des problèmes
  - ◆ Si on veut un parallélisme « correct », tout thread doit relâcher la main de temps en temps
    - ◆ Sans oublier qu'en cas de synchronisation/communication obligatoire entre threads, il faut que tout thread ait la main régulièrement

32

## Ordonnancement des threads Java

- ◆ Retour sur l'exemple du calcul de factoriel
  - ◆ Une fois que la méthode `run()` d'un thread est commencée, on doit donc supposer que ce thread garde au pire le processeur jusqu'à la fin de sa méthode `run()`
  - ◆ Pour avoir un meilleur parallélisme, il faut qu'un thread passe la main à un autre thread de temps en temps
  - ◆ Dans la classe `java.lang.Thread`
    - ◆ `public static void yield()`
    - ◆ Le thread s'interrompt et passe la main à un autre thread
  - ◆ Modification de l'exemple
    - ◆ Ajout d'un `yield()` après chaque calcul dans `run()`

```
for (int i=1; i<=nb; i++) {
    res = res * i;
    Thread.yield();
}
```

34

## Ordonnancement des threads

- ◆ Un thread passe la main à un autre dès qu'il est bloqué ou en attente, c'est-à-dire dans les cas suivants
  - ◆ Il est bloqué en attente sur une entrée/sortie (flux)
  - ◆ Il est bloqué sur l'accès à un objet synchronisé
  - ◆ Il se met en attente avec un `wait()`
  - ◆ Il fait une pause pendant une certaine durée avec un `sleep()`
  - ◆ Il a exécuté un `yield()` pour céder explicitement la main
  - ◆ Il se met en attente de la terminaison d'un autre thread avec un `join()`
  - ◆ Il se termine
  - ◆ Un thread de plus haute priorité demande la main
- ◆ Une application Java se termine quand
  - ◆ Le `main()` et tous les `run()` de tous les threads créés sont terminés

36

## Interactions entre threads

- ◆ Les threads sont des objets comme les autres
  - ◆ Ils possèdent des références sur d'autres objets
  - ◆ Un thread peut appeler des méthodes sur ces objets
  - ◆ On peut appeler des méthodes sur l'objet du thread
  - ◆ Communication/interaction possible via ces objets ou les méthodes du thread
    - ◆ Avec mécanisme possible d'accès en exclusion mutuelle
- ◆ Relations entre les cycles de vie des threads
  - ◆ Un thread peut lancer un autre thread
  - ◆ Un thread peut attendre qu'un ou plusieurs threads se terminent
  - ◆ Un thread peut se bloquer et attendre d'être réveillé par un autre thread

37

## Interactions entre threads

- ◆ Modification du constructeur pour passer le tableau partagé en paramètre

```
public CalculFactoriel(int nb, int[] tab) {
    this.nb = nb;
    this.tab = tab;
}
```

- ◆ Nouveau lancement des threads dans le thread principal

```
int[] resultats = new int[10];
CalculFactoriel cf;
for (int i=10; i >= 1; i--) {
    cf = new CalculFactoriel(i, resultats);
    (new Thread(cf)).start();
}
```

- ◆ Avant d'afficher les résultats : doit attendre que tous les threads soient terminés

39

## Interactions entre threads

- ◆ Modification exemple (suite)
  - ◆ Une fois la boucle avec les `join()` passée, on est certain que tous les threads de calcul sont finis
    - ◆ Peut alors afficher les résultats

```
for (int i=1; i<=10; i++)
    System.out.println(" factoriel de "
        +i+"="+resultats[i-1]);
```

- ◆ Trois méthodes de la classe `Thread` pour attendre la terminaison d'un thread

- ◆ `public void join()` : attend la fin du thread ou retourne immédiatement si le thread est déjà terminé
- ◆ `public void join(int milli)` : attend au plus `milli` millisecondes
- ◆ `public void join(int milli, int nano)` : attend au plus `milli` millisecondes et `nano` nanosecondes

41

## Interactions entre threads

- ◆ Communication par objet partagé
  - ◆ Les threads s'exécutent dans la même machine virtuelle, dans le même espace mémoire
    - ◆ Accès possible aux mêmes objets
  - ◆ Modification de l'exemple précédent pour ne plus afficher les résultats mais les stocker dans un tableau auquel tous les threads ont accès

```
public class CalculFactoriel
{
    protected int[] tab;
    protected int nb;

    public void run() {
        int res = 1;
        for (int i=1; i<=nb; i++)
            res = res * i;
        //enregistre le résultat dans tableau
        tab[nb - 1] = res;
    }
}
```

38

## Interactions entre threads

- ◆ Un thread attend qu'un thread se termine via la méthode `join()` appelée sur le thread dont on attend la fin
- ◆ Pour l'exemple, le thread principal doit attendre que tous les threads lancés soient terminés

```
int[] resultats = new int[10];
CalculFactoriel[] tabCF = new CalculFactoriel[10];
CalculFactoriel cf;
// lance les threads
for (int i=10; i>=1; i--) {
    cf = new CalculFactoriel(i, resultats);
    tabCF[i-1] = cf;
    cf.start();
}
// attend la fin de chaque thread
for (int i=0; i < 10; i++) {
    try { tabCF[i].join(); }
    catch (InterruptedException e) {System.err.println(e);}
}
```

40

## Interactions entre threads

- ◆ Méthodes `join()` (suite)
  - ◆ Les 3 méthodes `join()` peuvent lever l'exception `java.lang.InterruptedException`
    - ◆ Si exception levée : signifie que l'attente du thread a été interrompue et qu'il reprend son activité
      - ◆ Pour arrêter l'attente d'un thread : appel de la méthode `public void interrupt()` sur le thread
  - ◆ Interrogation sur l'état d'un thread
    - ◆ `public boolean isInterrupted()` : retourne vrai si le thread a été interrompu dans son attente
    - ◆ `public boolean isAlive()` : retourne vrai si le thread est en vie (démarré mais pas encore terminé)

42

## Synchronisation sur objets

- ◆ Tableau partagé de l'exemple
  - ◆ Chaque thread écrit dans sa case du tableau
  - ◆ Pas de risque de conflit dans ce cas
- ◆ Mais attention aux accès concurrents à des objets partagés
  - ◆ Peut conduire à des incohérences
    - ◆ Si 2 threads modifient en même temps le même objet par ex.
    - ◆ En pratique, sur une machine mono-processeur, un seul thread est actif en même temps
    - ◆ Mais un thread peut commencer une méthode, passer la main à un autre thread qui modifiera l'état de l'objet
      - ◆ Le premier thread reprend alors l'exécution de la méthode avec un état différent et incohérent

43

## Synchronisation sur objets

- ◆ Exemple (suite)
  - ◆ Lancement d'un calcul de puissance

```
CalculPuissance cp = new CalculPuissance();
... // passage de la référence de cp à d'autres threads
cp.setPuissance(3);
int resultat = cp.calculPuissance(2);
System.out.println("puissance 3 de 2 = "+resultat);
```
  - ◆ Problème
    - ◆ Si pendant l'exécution de `calculPuissance()`, un autre thread appelle `setPuissance()`, le calcul sera faux !
    - ◆ Exemple avec un autre thread appelant `setPuissance()` avec la valeur 4 pendant l'exécution de `calculPuissance()`:

```
puissance 3 de 2 = 16
```
    - ◆ Valeur 16 renvoyée au lieu de 8 ...
      - ◆ Car l'attribut `puissance` est passé à la valeur 4 au milieu de la boucle

45

## Synchronisation sur objets

- ◆ Retour sur l'exemple
  - ◆ Suppression de l'erreur potentielle de calcul
  - ◆ On rajoute `synchronized` dans la définition des méthodes

```
public synchronized int calculPuissance(int val) {
    int res = val;
    for (int i=1; i<puissance; i++)
        res = res * val;
    return res;
}
public synchronized setPuissance(int p) {
    puissance = p;
}
```
  - ◆ Il est alors impossible qu'un thread modifie la valeur de `puissance` lorsqu'un calcul est en cours
    - ◆ Car `synchronized` interdit que `setPuissance()` soit exécutée tant que l'exécution d'un `calculPuissance()` n'est pas finie

47

## Synchronisation sur objets

- ◆ Exemple de code pouvant poser problème

```
public class CalculPuissance {

    protected int puissance = 1;

    public int calculPuissance(int val) {
        int res = val;
        for (int i=1; i<puissance; i++)
            res = res * val;
        return res;
    }

    public setPuissance(int p) {
        puissance = p;
    }
}
```

- ◆ Doit prendre en compte le cas d'ordonnement le plus mauvais
  - ◆ Ordonnement préemptif des threads dans ce cas précis<sup>44</sup>

## Synchronisation sur objets

- ◆ Primitive `synchronized`
  - ◆ Elle s'applique par rapport à un objet (n'importe lequel)
  - ◆ Exclusion mutuelle sur une séquence de code
    - ◆ Il est impossible que 2 threads exécutent en même temps une section de code marquée `synchronized` sur un même objet
    - ◆ Sauf si un thread demande explicitement à se bloquer avec un `wait()`
- ◆ Deux utilisations de `synchronized`
  - ◆ Sur la méthode d'une classe (s'applique à tout son code pour un objet de cette classe)

```
public synchronized int calculPuissance(int val){...}
```
  - ◆ Sur un objet quelconque

```
synchronized(cp) {
    // zone de code protégée sur l'objet cp
}
```

46

## Synchronisation sur objets

- ◆ Exemple du calcul de puissance (suite)
  - ◆ Il reste un problème potentiel de cohérence, pour la séquence de lancement du calcul

```
CalculPuissance cp = new CalculPuissance();
... // passage de la référence de cp à d'autres threads
cp.setPuissance(3);
// un autre thread peut appeler ici setPuissance
cp.setPuissance(4); // exécuté dans un autre thread
int resultat = cp.calculPuissance(2);
System.out.println("puissance 3 de 2 = "+resultat);
```
  - ◆ Le résultat affiché sera là encore 16 au lieu de 8
    - ◆ Calcul effectué correctement cette fois mais ce n'est pas celui qui était voulu par le thread !

48



## Synchronisation sur objets

- ◆ Exemple du calcul de puissance (suite)
- ◆ Pour éviter ce problème, il faut protéger la séquence de positionnement de la puissance puis du calcul

```
CalculPuissance cp = new CalculPuissance();
int resultat;

... // passage de la référence de cp à d'autres threads

synchronized(cp) {
    cp.setPuissance(3);
    resultat = cp.calculPuissance(2);
}
System.out.println("puissance 3 de 2 = "+resultat);
```

- ◆ Avec ce code, il est impossible qu'un autre thread exécute sur l'objet `cp` la méthode `setPuissance()` entre le `setPuissance()` et le `calculPuissance()`

49

## Synchronisation sur objets

- ◆ Pour des variables de types primitifs (`int ...`) en accès concurrent, on utilise `volatile`

- ◆ Le problème n'est pas forcément dans la possible incohérence en lecture/écriture
- ◆ Mais vient du fonctionnement des threads
  - ◆ Localement, un thread gère une copie d'une variable partagée
  - ◆ La déclarer comme `volatile` force à garder la cohérence entre la copie locale et la variable partagée

- ◆ Exemple

```
protected volatile int nb;

public int incNb() { return nb++; }

◆ Assure que si un thread exécute incNb() il utilise la valeur de nb la plus à jour
```

51

## Synchronisation entre threads

- ◆ Du côté du thread attendant le résultat

- ◆ Solution basique (et erronée)
  - ◆ Vérifier en permanence la valeur de `available`

```
ThreadCalcul calcul;
// calcul lancé avec référence sur bon thread
...
// boucle attendant que le résultat soit disponible
while (!calcul.getAvailable()) {
    // ne fait rien, juste attendre que available change
}
int res = calcul.getResult();
```

- ◆ Problème

- ◆ Attente active
  - ◆ Le thread qui fait la boucle peut ne jamais lâcher la main
  - ◆ L'autre thread ne peut donc pas faire le calcul !

53

## Synchronisation sur objets

- ◆ Exemple du calcul de puissance (fin)
- ◆ Avec ce nouveau code, il y a trois sections de code protégées sur l'objet `cp`, avec un accès en exécution en exclusion mutuelle

- ◆ Tout le code de la méthode `setPuissance()`
- ◆ Tout le code de la méthode `calculPuissance()`
- ◆ La séquence

```
synchronized(cp) {
    cp.setPuissance(3);
    resultat = cp.calculPuissance(2);
}
```

- ◆ Si un thread est en train d'exécuter une de ces 3 sections protégées sur l'objet `cp`

- ◆ Aucun autre thread ne peut exécuter une des 3 sections protégées tant que le premier thread n'a pas fini d'exécuter sa section protégée

- ◆ Note

- ◆ La séquence de code incluse dans le `synchronized(cp) {...}` ne contient que des références à `cp` mais ce n'est pas une obligation

50

## Synchronisation entre threads

- ◆ Problème courant

- ◆ Besoin d'un point de synchronisation entre threads
  - ◆ Un thread fait un calcul et un autre thread attend que le résultat de ce calcul soit disponible pour continuer son exécution
  - ◆ Solution basique : déclarer un booléen `available` qui sera mis à vrai quand le résultat est disponible

```
public class ThreadCalcul extends Thread {

    protected boolean available;
    protected int result;

    public boolean getAvailable() { return available; }

    public int getResult() { return result; }

    public void run() {
        // faire calcul, mettre result à jour et
        // préciser que le résultat est disponible
        available = true;
        // continuer l'exécution du thread
    }
}
```

52

## Synchronisation entre threads

- ◆ Pour éviter le problème de l'attente active

- ◆ Soit le thread passe la main avec un `yield()` dans la boucle
  - ◆ Mais reste très actif pour pas grand chose ...
- ◆ Dans la boucle, le thread peut faire des pauses

- ◆ Pause d'un thread : `sleep` dans la classe `Thread`

- ◆ `public static void sleep(long millis[, int nanos]) throws InterruptedException`
  - ◆ Le thread courant fait une pause de `millis` millisecondes [et `nanos` nanosecondes]
  - ◆ Pendant cette pause, un autre thread peut alors prendre la main
  - ◆ L'exception `InterruptedException` est levée si le thread a été interrompu pendant sa pause

54

## Synchronisation entre threads

- ◆ Modification de la boucle d'attente avec un sleep

```
while (!calcul.getAvailable()) {
    try {
        Thread.sleep(100);
    }
    catch (java.lang.InterruptedException e) { ... }
}
int res = calcul.getResult();
```

- ◆ Problèmes

- ◆ Combien de temps doit durer la pause ?
- ◆ On est pas averti dès que le calcul est fini

- ◆ Solution idéale

- ◆ Se mettre en pause et être réveillé dès que le résultat est disponible
- ◆ Programmation en mode « réactif » : réaction/réveil sur événements, jamais d'attente ou de vérification active

55

## Synchronisation entre threads

- ◆ Synchronisation par moniteur

- ◆ `wait()`, `notify()` et `notifyAll()` sont des méthodes de la classe `java.lang.Object`

- ◆ On peut utiliser n'importe quel objet Java comme moniteur de synchronisation

- ◆ Application à l'exemple précédent

- ◆ Thread faisant le calcul

```
public class ThreadCalcul extends Thread {
    protected boolean available;
    protected int result;

    public synchronized boolean getAvailable() {
        return available;
    }
}
```

57

## Cycle de vie d'un thread

- ◆ Thread

- ◆ Objet ayant son propre flot d'exécution

- ◆ États, opérations associés aux threads

- ◆ Créé : instanciation standard d'un objet Java
- ◆ Démarré et actif : après appel de la méthode `start()`
  - ◆ Son propre flot d'exécution est créé et lancé
- ◆ En pause : méthode `sleep()`
- ◆ Bloqué sur un objet synchronisé : méthode `wait()`
  - ◆ Réveillé par un `notify()` sur le même objet
- ◆ Attente de la terminaison d'un autre thread : méthode `join()`
- ◆ Interrompue pendant une pause sur un `wait()`, `sleep()` ou un `join()` par l'appel de `interrupt()`
  - ◆ L'exception `InterruptedException` est levée
- ◆ Terminé : arrivé à la fin de sa méthode `run()`

59

## Synchronisation entre threads

- ◆ Synchronisation par moniteur

- ◆ Dans une section de code protégée par un `synchronized`, trois primitives de synchronisation sur un objet

- ◆ `public void wait() throws InterruptedException`
  - ◆ Le thread se bloque
  - ◆ Il permet alors à un autre thread d'exécuter une séquence de code protégée sur l'objet
    - ◆ C'est le seul cas où un thread peut exécuter une séquence protégée alors qu'un autre thread n'a pas terminé son exécution d'une séquence protégée
  - ◆ Il existe 2 variantes permettant de rester bloqué au plus un certain temps
- ◆ `public void notify()`
  - ◆ Débloquent un thread bloqué (pris au hasard si plusieurs threads bloqués) sur un `wait()` sur cet objet
- ◆ `public void notifyAll()`
  - ◆ Débloquent tous les threads bloqués sur un `wait()` sur cet objet

56

## Synchronisation entre threads

- ◆ Thread faisant le calcul (suite)

```
public void run() {
    // faire le calcul, mettre result à jour
    synchronized(this) {
        // résultat disponible : on débloquent tous les threads
        available = true;
        this.notifyAll();
    }
    // continuer l'exécution du thread

    // méthode de récupération du résultat : si pas encore
    // disponible, on attend qu'il le soit en se bloquant
    public synchronized int getResult() {
        while (!this.getAvailable()) {
            try { this.wait(); }
            catch (InterruptedException e) { ... }
        }
        return result;
    }
}
```

- ◆ Thread attendant le résultat :

```
int res = calcul.getResult();
```

- ◆ Si le résultat n'est pas disponible, on sera bloqué en attendant que `notify` soit exécuté par le thread de calcul

58

## Usage des threads

- ◆ Pourquoi et quand utiliser des threads ?

- ◆ Certains cas imposent des threads

- ◆ Pour entrées/sorties, les lectures sont généralement bloquantes
  - ◆ On dédie un ou plusieurs threads aux réceptions pour gérer des réceptions multiples, ainsi que d'éviter de bloquer le programme
  - ◆ Les threads de lecture communiquent par synchronisation/objets communs avec les autres threads
- ◆ Les interfaces graphiques utilisent également des threads, généralement de hautes priorités
  - ◆ Peut gérer des événements graphiques (clics ...) venant de l'utilisateur alors que le programme effectue d'autres traitements

- ◆ Généralisation des processeurs multi-cores, multi-threads

- ◆ Gain de performances en parallélisant le code de l'application, notamment pour les calculs longs
- ◆ Nécessite de revoir la façon de programmer un calcul

60

## Usage des threads

- ◆ Points délicats en programmation multi-threadée
  - ◆ Éviter les famines : un thread n'a jamais la main
    - ◆ Ne jamais utiliser d'attente active
    - ◆ S'assurer qu'un thread passe la main suffisamment
    - ◆ Ne pas avoir des threads de haute priorité beaucoup trop actifs
  - ◆ Éviter les interblocages
    - ◆ Un thread T1 attend le résultat d'un thread T2 qui attend lui le résultat du thread T1
      - ◆ Si chacun se bloque sur un `wait()`, aucun ne pourra faire le `notify()` réveillant l'autre
    - ◆ Interblocage peut se passer via une chaîne de plusieurs threads interdépendants
      - ◆ Pas toujours simple à détecter si dépendances complexes

61

## Parallélisation

- ◆ Dans le cas d'un calcul/traitement pouvant être découpé en sous-parties indépendantes
  - ◆ Java propose le framework « fork and join »
  - ◆ Des threads sont créés pour chaque sous-partie (fork)
    - ◆ De manière récursive au besoin
  - ◆ Un thread ayant créé des threads attend la fin de ses sous-threads (join)
- ◆ Illustration par l'exemple
  - ◆ On dispose d'un grand tableau de nombres réels
  - ◆ On veut appliquer le même calcul sur chaque case et créer un tableau de même taille avec les résultats
  - ◆ On définit un seuil en termes de nombre de cases
    - ◆ Si le tableau à traiter à une taille supérieure au seuil, on crée deux sous-threads à qui on passe une moitié du tableau à traiter
    - ◆ Si la taille est inférieure au seuil, on fait les calculs sur toutes les cases du (sous-)tableau

63

## Parallélisation

```
// Méthode qui fait le calcul sur la partie du tableau que le thread doit traiter
private void calculSequentiel() {
    for (int i = debut; i < fin; i++)
        // un calcul quelconque ...
        cible[i] = Math.pow(Math.Log10(Math.sqrt(source[i])), 1000);
}

// La méthode compute() est équivalente à run() pour la classe Thread,
// c'est elle qui est appelée dans le thread créé et lancé
@Override
protected void compute() {
    // Calcul de la taille de la partie du tableau à traiter
    int taille = fin - debut;

    // Si la taille est inférieure au seuil, on réalise le calcul puis on se termine
    if (taille < seuil) {
        calculSequentiel();
        return;
    }
    // Sinon on crée deux sous-calculs avec chaque demi-partie de la partie du tableau
    // à traiter. Le invokeAll lance un thread pour chacun de ces calculs
    // et attend qu'ils se terminent
    int milieu = taille / 2;
    invokeAll(new CalculFork(debut, debut + milieu, seuil, source, cible),
              new CalculFork(debut + milieu, fin, seuil, source, cible));
}
```

65

## Usage des threads

- ◆ Problèmes de performances en cas d'utilisation massive de threads
  - ◆ Relativement coûteux de créer un thread à chaque requête
    - ◆ En temps de création et de destruction par le garbage collector
    - ◆ Pool de threads : ensemble de threads déjà créés et qui peuvent être réutilisés pour traiter de nouvelles requêtes
      - ◆ La méthode `run()` est une boucle qui traite une requête à chaque passage
      - ◆ Avec la synchronisation (`wait/notify`) on peut relancer un passage dans la boucle pour traiter une nouvelle requête
      - ◆ Attention à la taille du pool selon le nombre de requêtes simultanées à traiter
  - ◆ Éviter de définir des méthodes en `synchronized` prenant un temps relativement long à s'exécuter et étant souvent appelées

62

## Parallélisation

```
// La classe doit hériter de ResursiveAction, équivalent de Thread pour un
// thread classique
public class CalculFork extends RecursiveAction {
    // Le tableau source
    private double[] source;
    // Le tableau cible qui contiendra les résultats
    private double[] cible;

    // L'index de début de la partie du tableau à traiter
    private int debut;
    // L'index de fin de la partie du tableau à traiter
    private int fin;

    // Le seuil pour savoir si on dédouble ou on fait le calcul
    private int seuil;

    public CalculFork(int debut, int fin, int seuil, double[] source, double[] cible) {
        this.source = source;
        this.cible = cible;
        this.debut = debut;
        this.fin = fin;
        this.seuil = seuil;
    }
}
```

64

## Parallélisation

```
public static void main(String argv[]) {
    // La taille du tableau
    int taille = 50000000;
    // Le seuil pour débouler dans deux sous-threads
    int seuil = 2000000;

    // On génère/obtient les deux tableaux d'une manière quelconque
    double source[] = generationSource(taille);
    double cible[] = generationCible(taille);

    long debut = System.currentTimeMillis();

    // On crée le calcul principal sur l'intégralité du tableau
    CalculFork calcul = new CalculFork(0, taille, seuil, source, cible);

    // On crée le gestionnaire du pool de threads ...
    ForkJoinPool pool = new ForkJoinPool();
    // ... et on lance le calcul
    pool.invokeAll(calcul);

    long temps = System.currentTimeMillis() - debut;
    System.out.println(" Temps d'exécution : " + temps);
}
```

66

## *Parallélisation*

- ◆ Test de performances
  - ◆ CPU : Intel i7-3540M, 3,0–3,7Ghz, dual-core, deux threads par core
    - ◆ 4 threads matériellement en exécution parallèle
  - ◆ Pour 50 millions de cases et un seuil de 2 millions
    - ◆ En pur séquentiel : autour de 4,1 secondes
    - ◆ Avec parallélisme : autour de 1,3 secondes
      - ◆ Pour 32 threads de calculs + 30 threads de fork/join
    - ◆ Plus de 3 fois plus rapide en profitant des 4 supports matériels d'exécution de threads du processeur
- ◆ Autres fonctionnalités autour des threads Java
  - ◆ Accès concurrents à des listes ou autres collections, avec gestion de la concurrence en écriture
  - ◆ File d'attente bloquante producteur/consommateur
  - ◆ Verrous sur des ressources partagées
  - ◆ Tris parallélisés
  - ◆ ...