JAVA – THREAD – PROGRAMMATION PARALLELE

Vidéo youtube de : Abdelwahab Naji
lien : https://www.youtube.com/watch?v=cM7CBYzd8hs

- Introduction du concept de Thrad
- Process vs Thread
- Comment créer des Threads avec JAVA (classes – interfaces - méthodes)
- Application multithread en JAVA
- Problème de priorité
- Problème de synchronisation

Le concept de Thread (JAVA)

Un processus est une instance d'un programme qui s'exécute

Un processus peut avoir plusieurs Threads

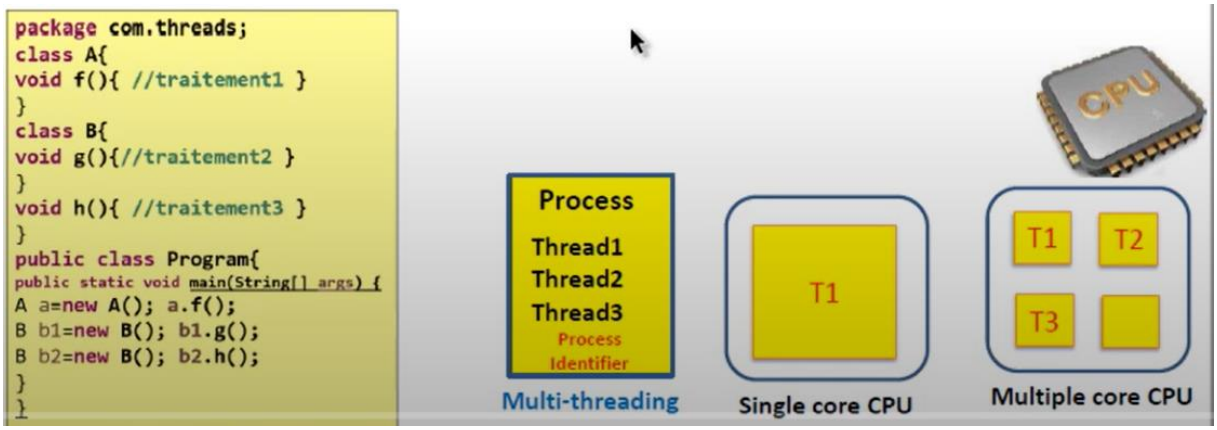
• Un Thread est une unité (ou sous ensemble) d'un processus

• Un thread fonctionne en parallèle avec autres threads d'un même processus (multiThreading)

• Sur une machine monoprocesseur ou single core: la JVM alloue le temps d'utilisation du CPU pour accomplir les traitements

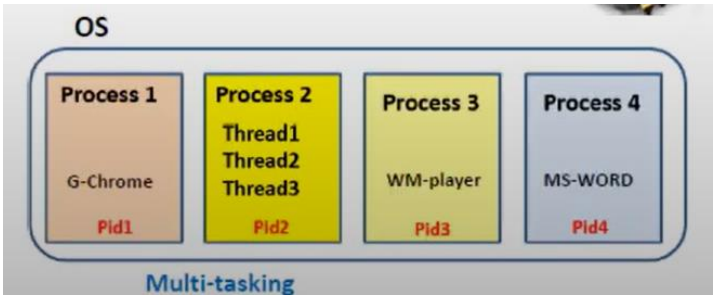
• Sur une machine multiprocesseur (multicore): la JVM répartit l'exécution sur plusieurs cores

Les Threads sont aussi considérés comme tâche ou processus léger(light weight process)



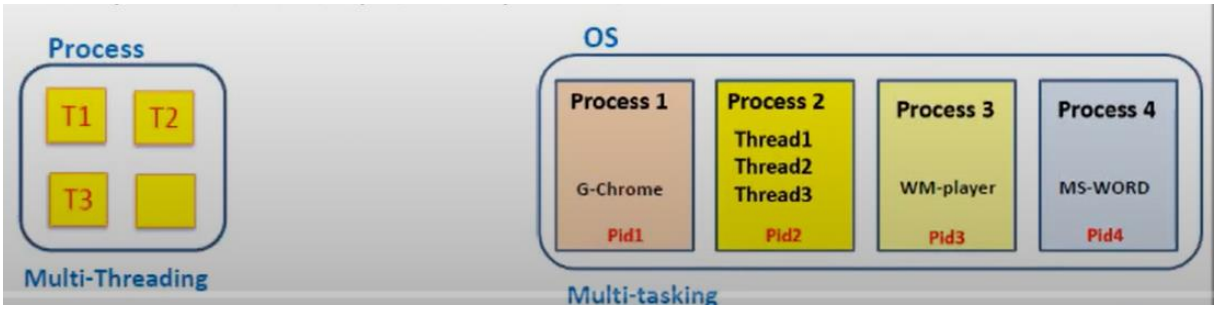
Le concept de Processus

- Un processus est une instance en exécution d'un programme
- La gestion de l'exécution des processus est assurée par le système d'exploitation (OS)
- Un OS capable d'exécuter plusieurs processus en même temps : Multi-Tasking
- Un processus peut contenir des threads



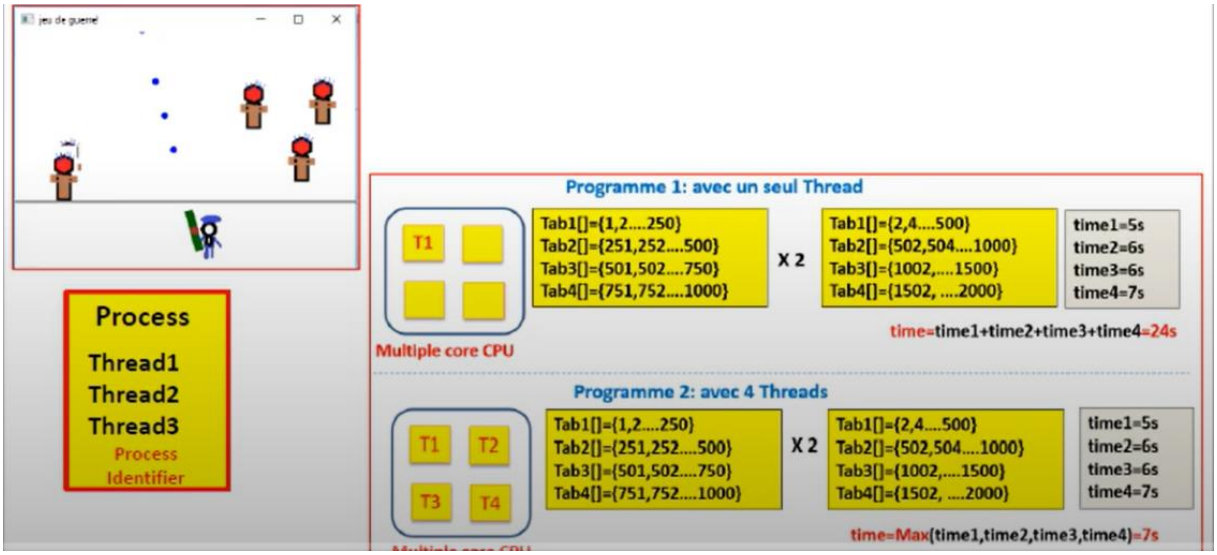
Processus vs Thread

- Les processus sont indépendants l'un de l'autre, tandis que les Threads appartiennent au même processus
- Les processus ont des adresses mémoires différentes, tandis que les Threads partagent la même zone mémoire
- Les processus peuvent communiquer entre eux à travers leur capacité d'échanges de données, tandis que les Threads peuvent avoir accès direct aux ressources occupées par leur processus
- Changement de contexte entre processus est plus lent que celui entre threads
    - Mémorisation d'état
    - Reprise d'exécution à partir d'un point donnée



Pourquoi les Threads

- Pour maintenir la réactivité d'une application durant une longue tâche d'exécution
- Pour donner la possibilité d'annulation de tâches séparables
- Pour exécuter des traitements en parallèle (gagner du temps)



Threads et JAVA

- En java, un thread est un Objet
- On le crée et on lui assigne des traitements à faire => qu'est ce qu'il va exécuter en parallèle
- La JVM prend en charge la responsabilité d'exécuter le Thread
- Un thread a plusieurs états
- Les programmes que nous avons réalisés jusqu'à maintenant contiennent un seul Thread (le main)
- Ce Thread existe tout seul dans l'espace mémoire réservé au programme. Il n'a pas de concurrent.
- Il exécute ses traitements de manière séquentielle

Il y a thread qui est exécuté par défaut :

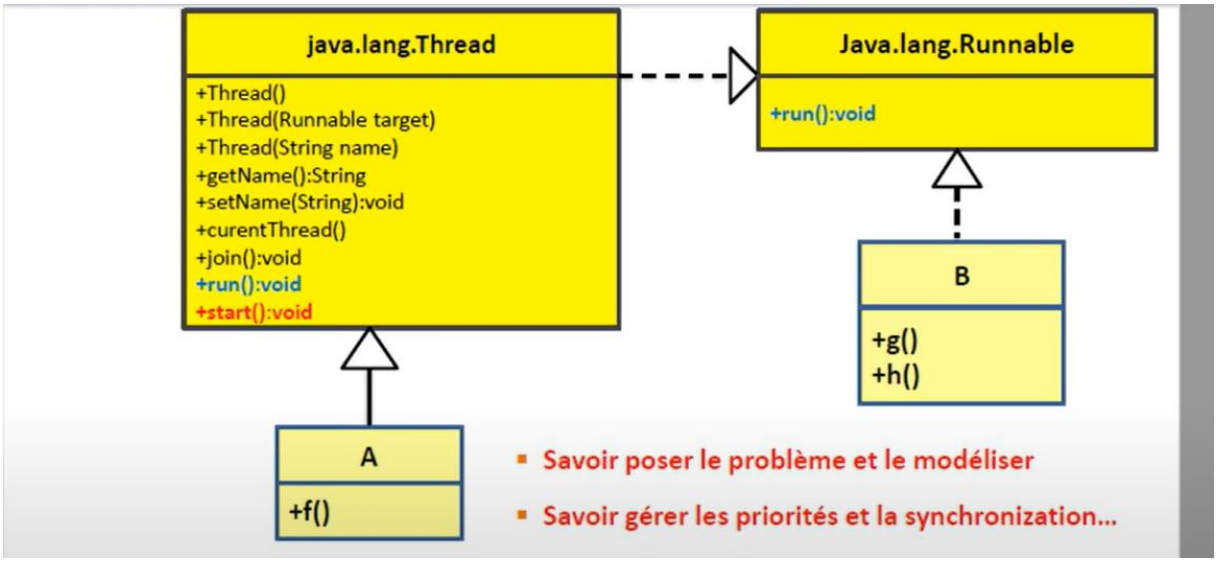
Main.java

```
1  package thread.setp1;
2
3
4  public class Main {  new *
5      public static void main(String[] args) {  new *
6
7          //double a =Double.parseDouble("tt");
8
9          System.out.println(Thread.currentThread().getName());
10         System.out.println(Thread.currentThread().getState());
11         System.out.println(Thread.currentThread().getId());
12
13         }
14     }
15
```

Run Main

```
C:\Users\zotri\.jdks\openjdk-19.0.2\bin\java.exe "-javaagent:C:\Users\zotri\AppData\Local\Programs\
main
RUNNABLE
1

Process finished with exit code 0
```

Comment créer des Threads en JAVA ?



- Deux méthodes:
  - Créer un thread en utilisant la classe java.lang.Thread
  - Créer un thread en utilisant l'interface java.lang.Runnable

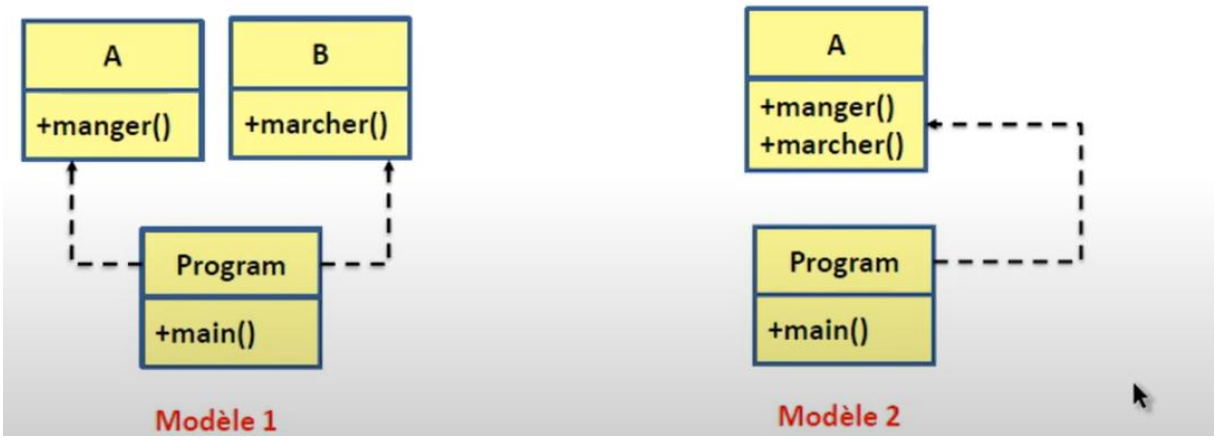
• run(): void When an Object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

• start(): void, Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

• La méthode start() est appelée une seule fois sur un thread.

Simulation de comportement (Séquentielle) : sans Thread

- Simuler le comportement d'une personne
  - manger() (10 secondes pour manger)
  - marcher() (15 secondes pour marcher)
  - Les deux traitements se font de manière séquentielle (25 secondes pour manger et marcher)

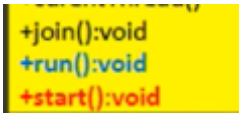


Exemple séquentielle (sans thread):

```java
class A {
    void manger() {
        for (int i = 0; i < 10; i++)
            System.out.println("je mange" + i);
    }
}

class B{
    void marcher() {
        for (int i = 0; i < 10; i++)
            System.out.println("je marche" + i);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("je me suis reveillé tard !");
        // manger et marcher pour aller à l'école

        //séquentielle
        A a = new A();
        B b = new B();
        a.manger();
        b.marcher();
    }
}
```

```
C:\Users\zotri\.jdks\openjdk-19.0.2\bin\java.exe "-javaag
je me suis reveillé tard !
je mange0
je mange1
je mange2
je mange3
je mange4
je mange5
je mange6
je mange7
je mange8
je mange9
je marche0
je marche1
je marche2
je marche3
je marche4
je marche5
je marche6
je marche7
je marche8
je marche9

Process finished with exit code 0
```

Pour le parallèle il ne suffit pas d'ajouter « extends Thread » à une classe. Il est obligatoire d'ajouter la méthode run dans chaque classe



Ici j'ai 3 threads :

1. Main
2. A
3. B

```
class B extends Thread{  2 usages  new *
    void marcher() {  1 usage  new *
        for (int i = 0; i < 10; i++)
            System.out.println("je marche" + i);
    }

    @Override  new *
    public void run() {
        System.out.println("run....");

    }
}

public class Main {  new *
    public static void main(String[] args) {  new *
        System.out.println("je me suis reveillé tard !");
        // manger et marcher pour aller à l'école

        A a = new A();
        B b = new B();
        a.manger();
        b.marcher();
        a.start();
        b.start();
    }
}
```

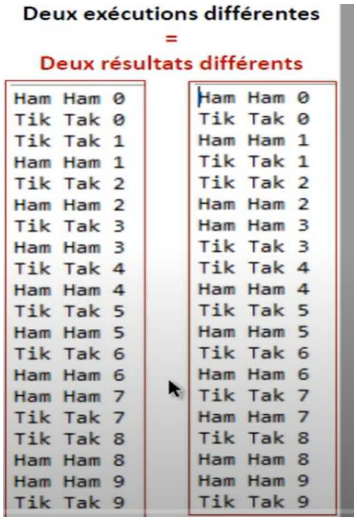
Une fois le Run mis avec la méthode dans la classe il faut appeler la méthode par exemple a.start appelle la méthode run qui elle appelle la méthode manger =>

Main.java

```
3   class A extends Thread {  2 usages  new *
11          manger();
12      }
13  }
14
15  class B extends Thread {  2 usages  new *
16      void marcher() {  2 usages  new *
17          for (int i = 0; i < 10; i++)
18              System.out.println("je marche" + i);
19      }
20
21      @Override  new *
22      public void run() {
23          marcher();
24      }
25  }
26
27  public class Main {  new *
28      public static void main(String[] args) {  new *
29          System.out.println("je me suis reveillé tard !");
30          // manger et marcher pour aller à l'école
31
32          A a = new A();
33          B b = new B();
34          a.manger(); //appel direct de la méthode pas de traitement en parallèle
35          b.marcher(); //appel direct de la méthode pas de traitement en parallèle
36          a.start(); //traitement en parallèle
37          b.start(); //traitement en parallèle
38      }
39  }
```

Thread & JVM

/ ! \ Chaque exécution en parallèle est aléatoire



- Le Thread a besoin de la JVM pour s'exécuter
- À une unité de temps donnée, un seul thread qui s'exécute
- Quand il s'agit de plusieurs Threads qui s'exécutent en parallèle, la JVM décide quel thread exécuter pour une unité de temps (Threads scheduler algorithm)
    - L'implémentation de cet algorithme dépend de l l'implémentation de la JVM et on peut pas la changer

    • mais on peut influencer sur le temps alloué à chaque thread à l'aide des méthodes disponibles dans la classe Thread et la classe Object. ***Avec la méthode sleep par exemple comme illustré ci-dessous***

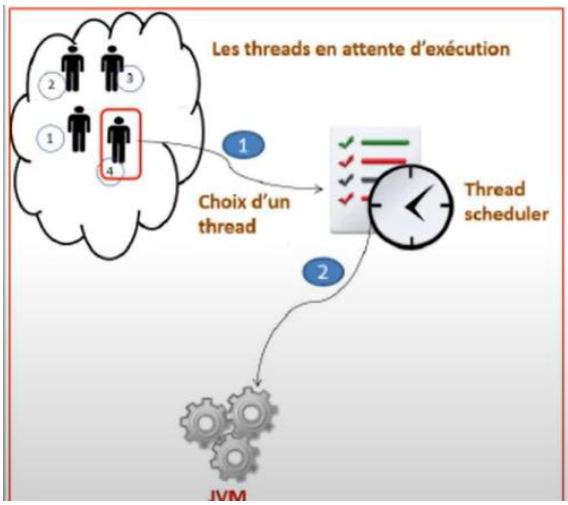
• Le thread a une pile d'exécution

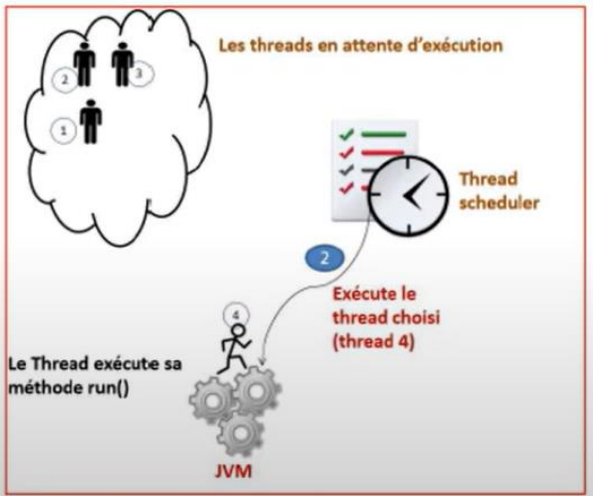
- Le développeur ne peut pas prévoir l'ordre d'exécution: se fait de manière aléatoire par la JVM

Avec un état : sleep => la JVM n'exécute aucune tache à ce moment-là elle va faire appel un autre thread

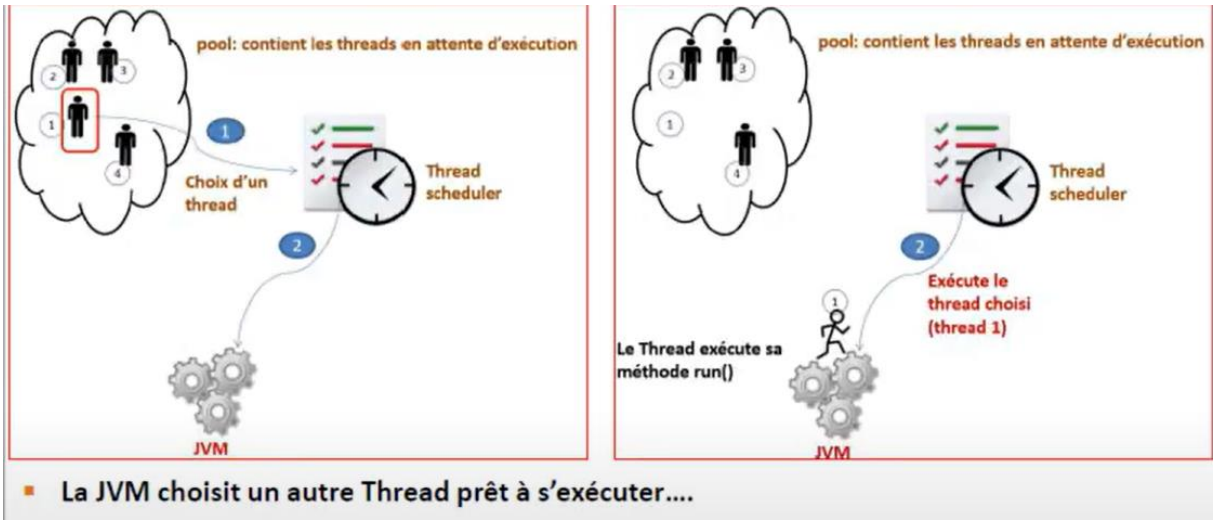
```java
class B extends Thread {  2 usages  new *
    void marcher() {  1 usage  new *
        for (int i = 0; i < 10; i++) {
            System.out.println("je marche" + i);
              try {
                Thread.sleep( millis: 1000);
            } catch
            (Exception e) {
            }

        }
    }

    @Override  new *
    public void run() {
        marcher();
    }
}
```

- Thread scheduler choisit un Thread à exécuter
- Exécution d'un Thread parmi plusieurs Threads
- Chaque thread change d'état entre running et en attente jusqu'à terminer son traitement

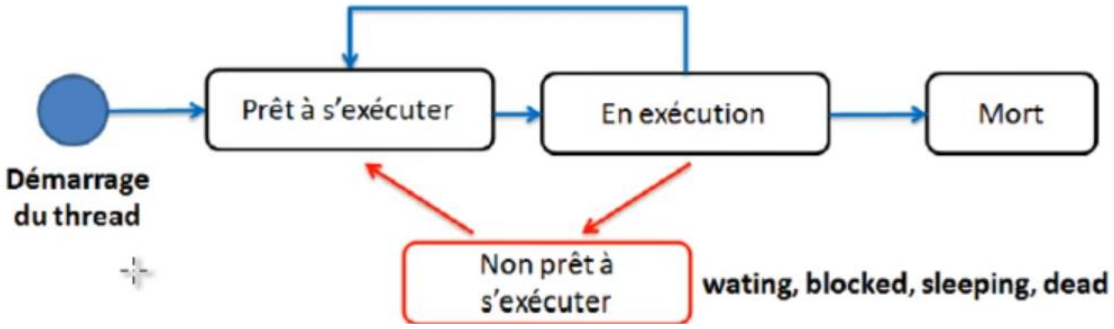




- Thread N04 est prêt à s'exécuter — choisi par la JVM — exécuté pour un temps défini
- La JVM le rend à l'ensemble des threads en attente



Cycle de vie et les états d'un Thread



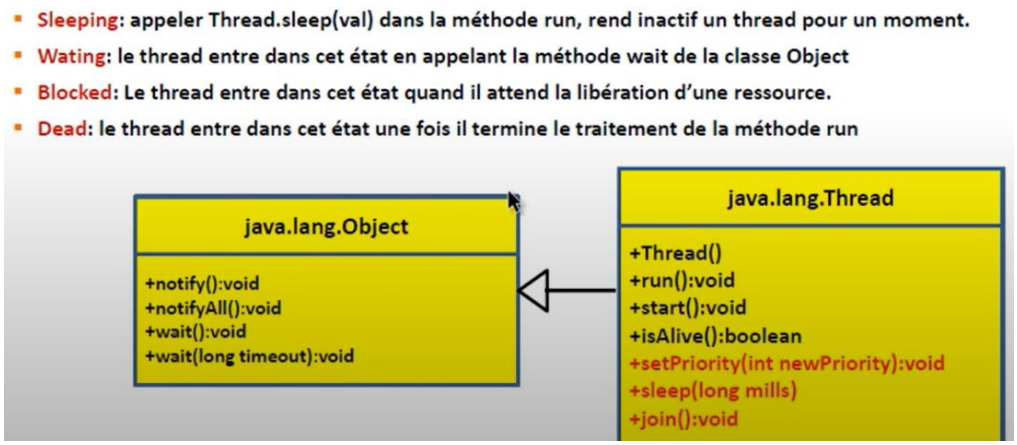
• Non Prêt à s'exécuter: ne sera jamais choisi par la JVM tant qu'il est dans cet état.

• Prêt à s'exécuter (runnable): se trouve dans le pool d'exécution et éligible à s'exécuter mais le scheduler ne l'a pas encore piqué.

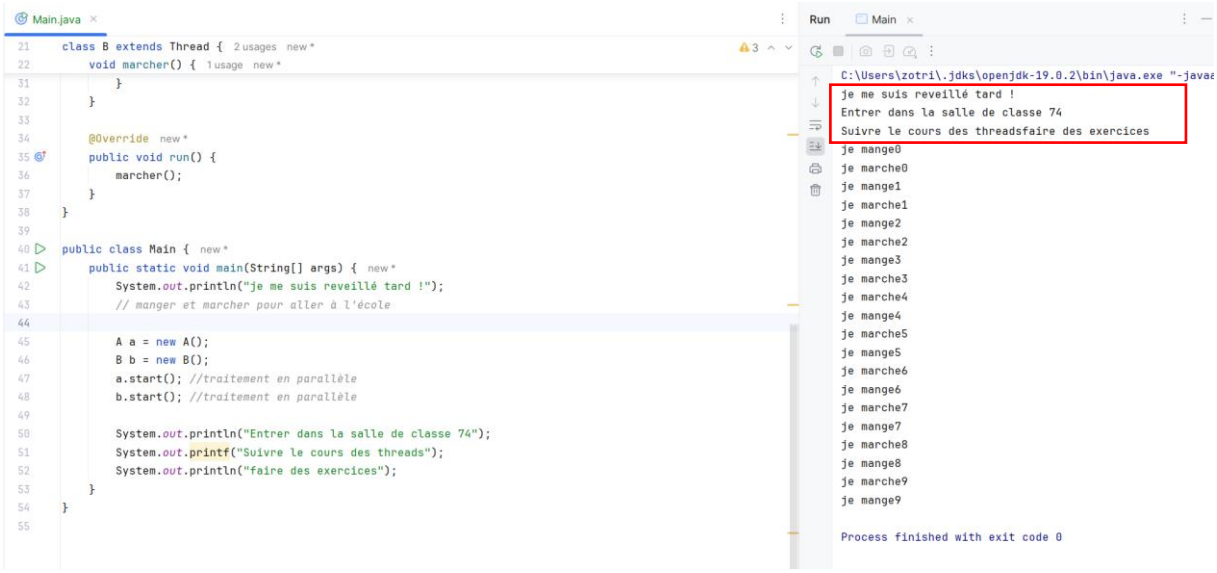
• Le thread ne peut entrer à cet état seulement si la méthode start a été appelée sur lui.

• En exécution (running): quand le thread entre dans cet état la JVM prend en charge l'exécution du code se trouvant dans la méthode run().

• Le thread reste dans cet état jusqu'à ce que le scheduler le rend au pool.



Ce n’est pas possible, il faut synchroniser les traitements



```java
class B extends Thread {
    void marcher() {
    }
    @Override
    public void run() {
        marcher();
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println("je me suis reveillé tard !");
        // manger et marcher pour aller à l'école
        A a = new A();
        B b = new B();
        a.start(); //traitement en parallèle
        b.start(); //traitement en parallèle
        System.out.println("Entrer dans la salle de classe 74");
        System.out.printf("Suivre le cours des threads");
        System.out.println("faire des exercices");
    }
}
```

```
C:\Users\zotri\.jdks\openjdk-19.0.2\bin\java.exe "-javaa
je me suis reveillé tard !
Entrer dans la salle de classe 74
Suivre le cours des threadsfaire des exercices
je mange0
je marche0
je mange1
je marche1
je mange2
je marche2
je mange3
je marche3
je marche4
je mange4
je marche5
je mange5
je marche6
je mange6
je marche7
je mange7
je marche8
je mange8
je marche9
je mange9

Process finished with exit code 0
```

Vu que A et B sont en sleep, le thread main va profiter d’exécuter toutes Tses tâches

Influencer le cycle de vie et les états d’un Thread

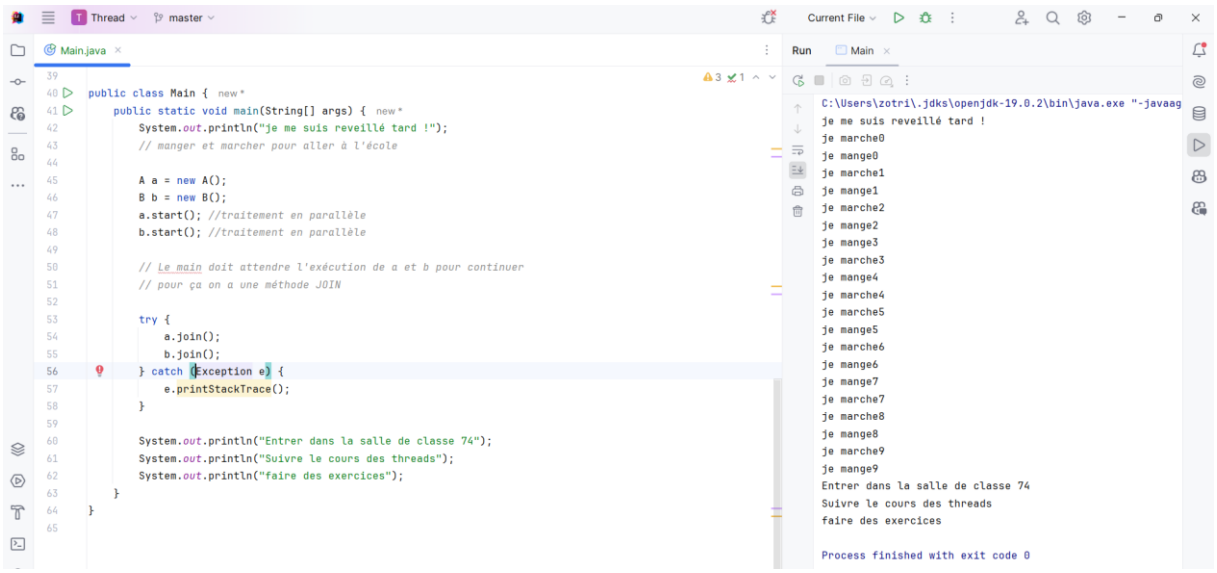
• join()

- Utilisé si on souhaite que le thread soit exécuté aprés l'exécution d'un autre thread
- Utile quand l'exécution d'un thread dépend de l'exécution d'un autre thread

• L'instruction t1.join() est exécutée par un autre tread t2. cela signifie que t2 doit

attendre la fin de t1 et reprend son exécution.

Vu que le main est en cours d’exécution, il va attendre la fin de l’exécution de a et b avant de passer à la suite :



```java
public class Main {
    public static void main(String[] args) {
        System.out.println("je me suis reveillé tard !");
        // manger et marcher pour aller à l'école
        A a = new A();
        B b = new B();
        a.start(); //traitement en parallèle
        b.start(); //traitement en parallèle
        // Le main doit attendre l'exécution de a et b pour continuer
        // pour ça on a une méthode JOIN
        try {
            a.join();
            b.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Entrer dans la salle de classe 74");
        System.out.println("Suivre le cours des threads");
        System.out.println("faire des exercices");
    }
}
```

```
C:\Users\zotri\.jdks\openjdk-19.0.2\bin\java.exe "-javaag
je me suis reveillé tard !
je marche0
je mange0
je marche1
je mange1
je marche2
je mange2
je mange3
je marche3
je mange4
je marche4
je marche5
je mange5
je marche6
je mange6
je mange7
je marche7
je marche8
je mange8
je marche9
je mange9
Entrer dans la salle de classe 74
Suivre le cours des threads
faire des exercices

Process finished with exit code 0
```

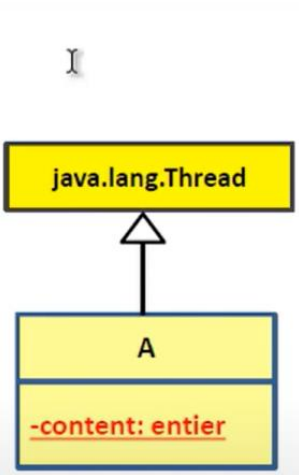
On peut donner une priorité =>

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("je me suis reveillé tard !");
        // manger et marcher pour aller à l'école

        A a = new A();
        B b = new B();
       // a.setPriority(1);  priorité de a de 1 à 10

        a.setPriority(Thread.MAX_PRIORITY);
        b.setPriority(Thread.MIN_PRIORITY);

        a.start(); //traitement en parallèle
        b.start(); //traitement en parallèle

        // Le main doit attendre l'exécution de a et b pour continuer
        // pour ça on a une méthode JOIN

        try {
            a.join();
            b.join();
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Entrer dans la salle de classe 74");
        System.out.println("Suivre le cours des threads");
        System.out.println("faire des exercices");
    }
}
```

```
C:\Users\zotri\.jdks\openjdk-19.0.2\bin\java.exe "-javaag
je me suis reveillé tard !
je marche0
je mange0
je mange1
je mange2
je mange3
je mange4
je mange5
je mange6
je marche1
je marche2
je mange7
je mange8
je marche3
je mange9
je marche4
je marche5
je marche6
je marche7
je marche8
je marche9
Entrer dans la salle de classe 74
Suivre le cours des threads
faire des exercices

Process finished with exit code 0
```

Problème de ressources partagée

```java
package com.threads;
class A extends Thread{
public static int content=0;
@Override
public void run() {
for(int i=0;i<1000;i++){
System.out.println(" work "+i +" of "+this.getName());
content++;
}
}
}
public class Program{
public static void main(String[] args) {
A a1=new A();
A a2=new A();
a1.setName("Ali");
a2.setName("Baba");
a1.start();
a2.start();
System.out.println("******************** content: "+A.content);
}
}
```



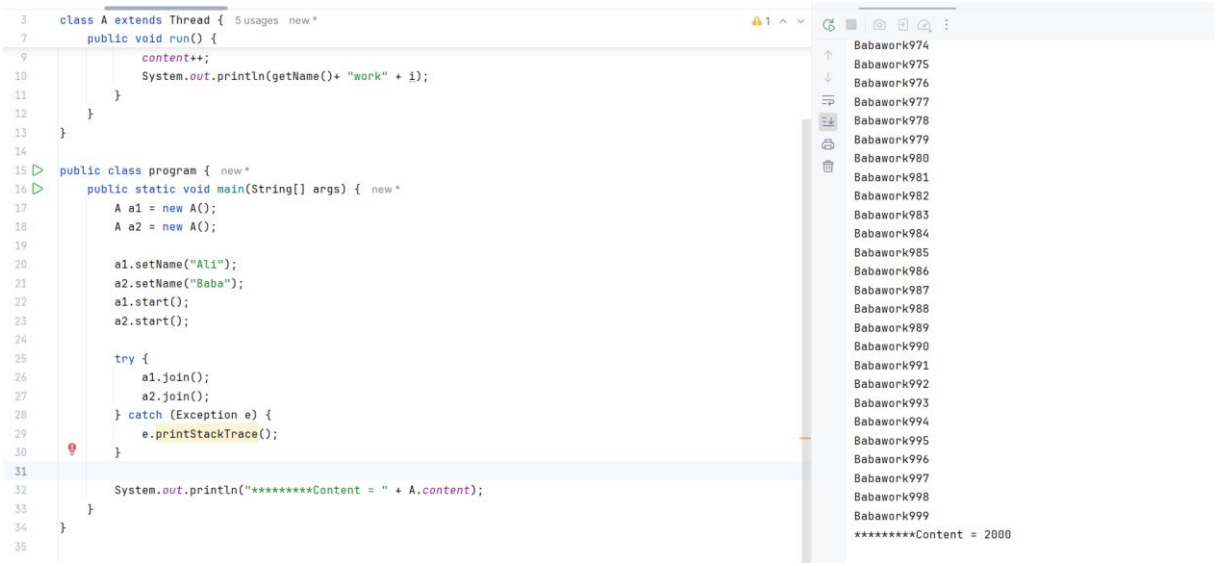
- Dans le cas où il s'agit d'un accès concurrent à une ressource, des problèmes de synchronisation peuvent se poser
- Java offer la possibilité de gérer ce type de problèmes

Ce résultat donnera 0 car pas de join :

```java
package thread.setp1.shared;

class A extends Thread {  5 usages  new *
    public static int content = 0;  2 usages

    @Override  new *
    public void run() {
        for (int i = 0; i < 1000; i++)
        content++;
    }
}

public class program {  new *
    public static void main(String[] args) {  new *
        A a1 = new A();
        A a2 = new A();

        a1.start();
        a2.start();

        System.out.println("*********Content = " + A.content);
    }
}
```

```
C:\Users\zotri\.jdks\openjdk-19.0.2
*********Content = 0
```

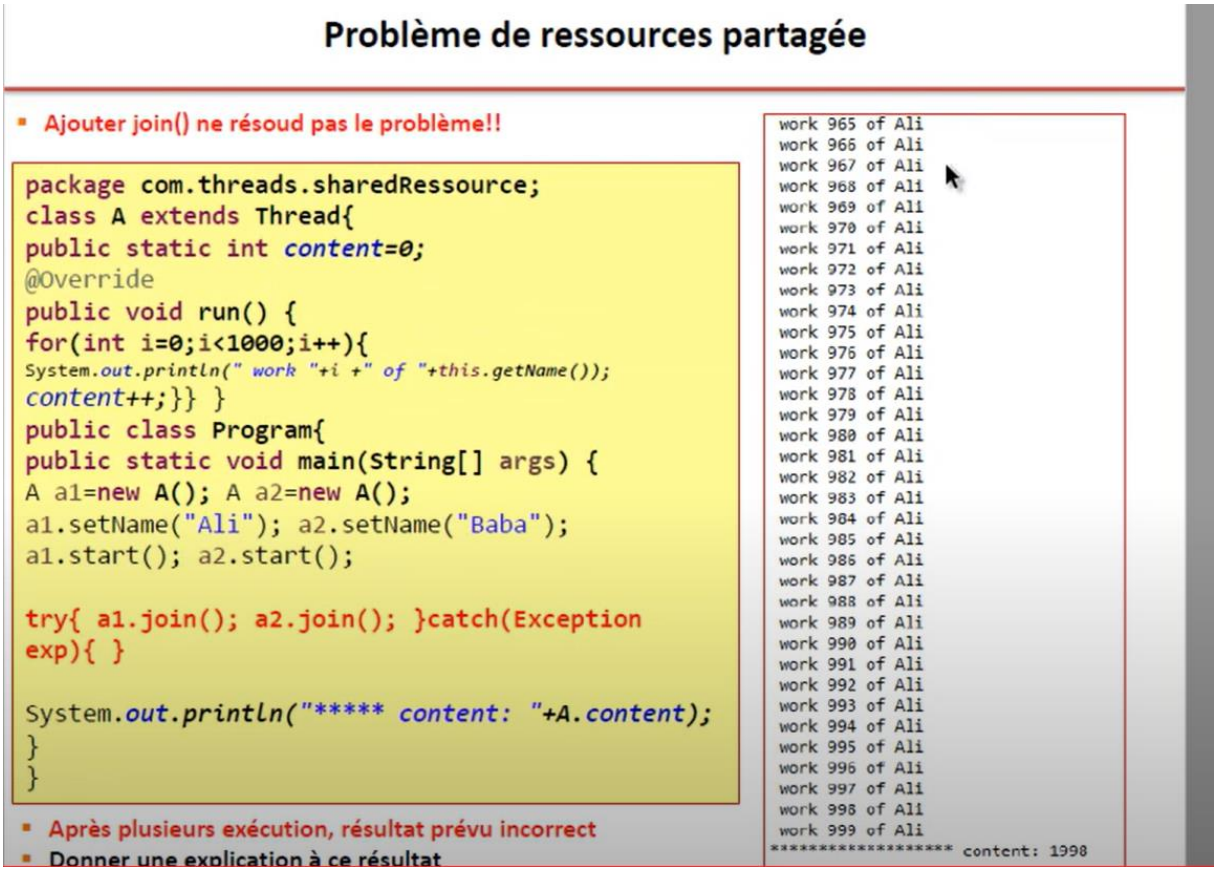
Il y a un accès en concurrence, en même temps A1 et A2, il y a une probabilité d'avoir une erreur



```
class A extends Thread {  5 usages  new *
    public void run() {
            content++;
            System.out.println(getName()+ "work" + i);
        }
    }
}

public class program {  new *
    public static void main(String[] args) {  new *
        A a1 = new A();
        A a2 = new A();

        a1.setName("Ali");
        a2.setName("Baba");
        a1.start();
        a2.start();

        try {
            a1.join();
            a2.join();
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("*********Content = " + A.content);
    }
}
```

```
Babawork974
Babawork975
Babawork976
Babawork977
Babawork978
Babawork979
Babawork980
Babawork981
Babawork982
Babawork983
Babawork984
Babawork985
Babawork986
Babawork987
Babawork988
Babawork989
Babawork990
Babawork991
Babawork992
Babawork993
Babawork994
Babawork995
Babawork996
Babawork997
Babawork998
Babawork999
*********Content = 2000
```

Il faut autoriser qu'un seul accès à une ressource partagé



Problème de ressources partagée

- Ajouter join() ne résoud pas le problème!!

```
package com.threads.sharedRessource;
class A extends Thread{
public static int content=0;
@Override
public void run() {
for(int i=0;i<1000;i++){
System.out.println(" work "+i +" of "+this.getName());
content++;}} }
public class Program{
public static void main(String[] args) {
A a1=new A(); A a2=new A();
a1.setName("Ali"); a2.setName("Baba");
a1.start(); a2.start();

try{ a1.join(); a2.join(); }catch(Exception
exp){ }

System.out.println("***** content: "+A.content);
}
}
```

```
work 965 of Ali
work 966 of Ali
work 967 of Ali
work 968 of Ali
work 969 of Ali
work 970 of Ali
work 971 of Ali
work 972 of Ali
work 973 of Ali
work 974 of Ali
work 975 of Ali
work 976 of Ali
work 977 of Ali
work 978 of Ali
work 979 of Ali
work 980 of Ali
work 981 of Ali
work 982 of Ali
work 983 of Ali
work 984 of Ali
work 985 of Ali
work 986 of Ali
work 987 of Ali
work 988 of Ali
work 989 of Ali
work 990 of Ali
work 991 of Ali
work 992 of Ali
work 993 of Ali
work 994 of Ali
work 995 of Ali
work 996 of Ali
work 997 of Ali
work 998 of Ali
work 999 of Ali
****************** content: 1998
```

- Après plusieurs exécution, résultat prévu incorrect
- Donner une explication à ce résultat

Le multi threading et l'accès concurrent

- Identifier les parties critiques de code
- Un seul thread à la fois qui doit accéder à ces parties
- Java utilise le mécanisme de verrouillage
- Si un thread arrive à accéder à un code critique (non verrouillé par un autre thread) il le verrouille. Ainsi il possède le verrou (lock)
- Un seul thread à la fois qui peut avoir le verrou.
- Tant que un thread ne relâche pas le verrou (lock) sur un objet, les autres threads ne peuvent pas accéder à ces parties critiques.

T1
T2

```
public void run() {
for(int
i=0;i<1000;i++){
content++;
}
}
```

Le multi threading et l'accès concurrent

- Le concept de synchronisation est introduit
- La synchronisation est un mécanisme qui coordonne l'accès à des données commune et à du code critique par des threads
- Seulement la méthode toute entière ou l'une de ces parties qui peuvent être synchronisée
- On ne peut pas synchroniser une classe ou des attribut
- La synchronisation est assurée par le mot clé synchronized

T1
T2

```
public void run() {
for(int
i=0;i<1000;i++){
content++;
}
}
```

Le multi threading et l'accès concurrent

- **Comment synchroniser?**
- **Synchroniser une méthode en sa totalité est le moyen le plus simple pour assurer qu'un seul thread accède à un code critique**
- **Parfois, seulement une partie du code qui a besoin d'être protègée. Dans ce cas, la partie du code à protéger est synchronisée avec un objet**

```
void run () {
  synchronized(this) {
   content++    ...
 }
}
```

```
private synchronized void  incrementer() {
content++    ...

}
```

Main.java program.java

```
1  package thread.setp1.shared;
2
3  class A extends Thread {  5 usages  new *
4      public static int content = 0;  3 usages
5      static synchronized void increment() {  1 usage  new *
6          content = content + 1;
7      }
8
9      @Override  new *
10     public void run() {
11         for (int i = 0; i < 10000; i++){
12             increment();
13             System.out.println(getName()+ " work " + i);
14         }
15     }
16 }
```

Exemples: