IUT de Blois

Les threads

R309 Programmation événementielle
R&T 2 – IUT de Blois
sameh.kchaou@univ-tours.fr

Fiche Matière

Objectif général

- Utiliser les techniques de programmation en réaction à des événements abordés du point de vue interface homme machine, réseau et système.

Objectifs spécifiques

- Se familiariser avec la programmation événementielle.
- Connaître la panoplie des outils de développement.
- Programmer des applications informatiques interactives.
- Concevoir des Interfaces Homme Machine Professionnelle

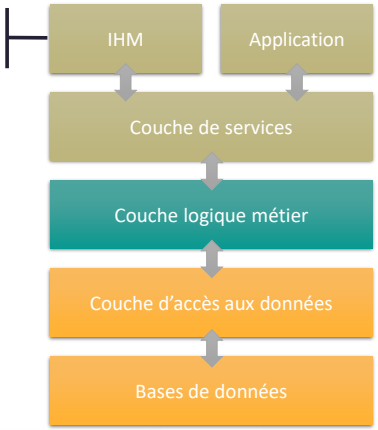
Plan

Introduction

Introduction

Architecture orientée services (SOA)

Evénements en parallèle : chargements, animations, mouvements,…



Application serveur communique avec plusieurs clients

Programmation concurrente

Programmation concurrente

Définition | Intérêt | Fonctionnement

- La programmation concurrente est l'activité d'effectuer plusieurs traitements, spécifiés distinctement les uns des autres, en même temps.
- En général, dans la spécification d'un traitement, beaucoup de temps est passé à attendre
  - ✓ Idée: exploiter ces temps d'attente pour réaliser d'autres traitements, en exécutant en concurrence plusieurs traitements
  - ✓ Sur monoprocesseur, simulation de parallélisme
  - ✓ Multitâche préemptif, partage de façon équilibrée le temps processeur entre différents processus.

Programmation concurrente

Définition | **Intérêt** | Fonctionnement

- Optimiser l'utilisation du/des processeurs
- Tirer avantage des architectures multi-cores
  - ✓ Les ordinateurs modernes possèdent au moins 2 cœurs.
- Simplifier la structure d'un programme
  - ✓ Écrire plusieurs programmes simples, plutôt qu'un seul programme capable de tout faire, puis de les faire coopérer pour effectuer les tâches nécessaires.
- Augmenter le parallélisme
  - ✓ Tout programme faisant du calcul devrait être développé de manière concurrente.
- Permettre à plusieurs utilisateurs de travailler sur la même machine
- Satisfaire des contraintes temporelles
  - ✓ Programmation temps réel.

Programmation concurrente

Définition | Intérêt | **Fonctionnement**

- Le multitâche préemptif est assuré par **l'ordonnanceur** (scheduler), un service de l'OS.
- **Une quantité de temps définie** (quantum) est attribuée par l'ordonnanceur à chaque processus : les processus ne sont donc pas autorisés à prendre un temps non-défini pour s'exécuter dans le processeur.
- Dans un ordonnancement (statique à base de priorités) avec préemption, un processus peut être préempté (remplacé) par n'importe quel processus plus prioritaire qui serait devenu prêt.

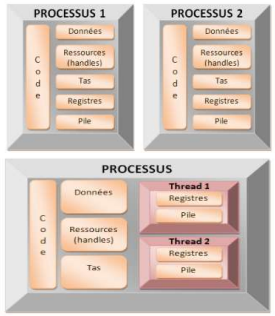
Programme concurrent

Définition | Intérêt | **Fonctionnement**

- L'exécution d'un processus se fait dans son contexte. Quand il y a changement de processus courant, il y a une commutation ou changement de contexte.
- En raison de ce contexte, la plupart des systèmes offrent la distinction entre :

✓« **processus lourd** », qui sont complètement isolés les uns des autres car ayant chacun leur contexte,

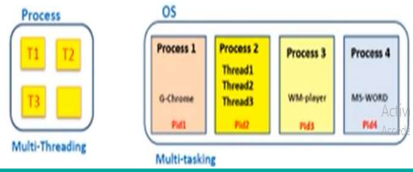
✓« **processus légers** » **(threads)**, qui partagent un contexte commun sauf la pile (les threads possèdent leur propre pile d'appel).



Processus VS thread

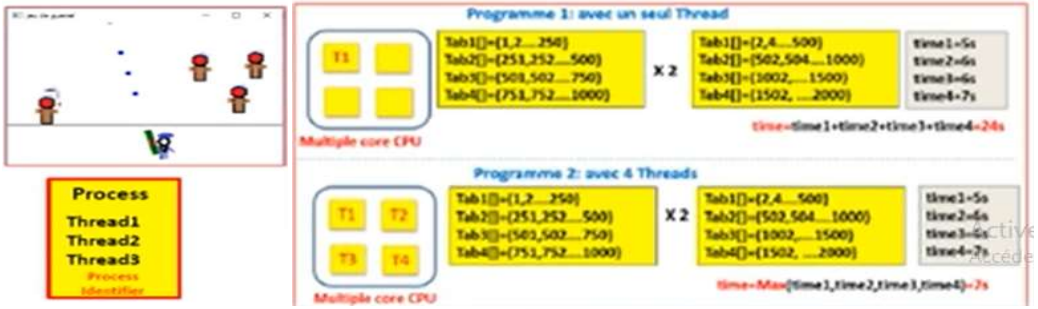
Processus VS thread

- 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



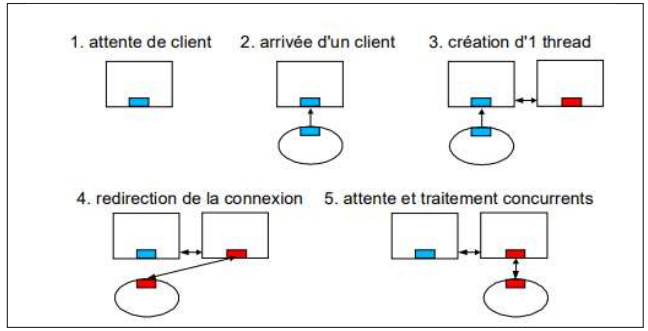
Processus VS thread

- **Pourquoi les threads ?**
  - ✓Pour maintenir la réactivité d'une application durant une longue tache d'exécution
  - ✓Pour donner la possibilité d'annulation de taches séparables
  - ✓Pour exécuter des traitements en parallèle (gagner de temps)



Processus VS thread

- **Autre exemple: Client/Serveur**
  - ✓ L'attente d'un nouveau client peut se faire en même temps que le service au client déjà la.



Les threads en Java

Les threads en Java

- Java supporte l'utilisation des threads de façon native :
  - ✓ Un thread est une instance de la classe Thread
  - ✓ La JVM intègre un moteur multithreads
  - ✓ A l'inverse de la plupart des autres langages, le programmeur n'a pas à utiliser des librairies natives du système d'exploitation pour écrire des programmes multitâches
  - ✓ Deux objets sont associés à un thread :
    - Objet déterminant le code à exécuter par le thread (Amorce du thread)
    - Objet contrôlant le thread et le représentant auprès des objets de l'application(Contrôleur du thread)

Les threads en Java

- La classe de l'objet qui définit l'amorce de code à exécuter doit implémenter l'interface Runnable qui force l'implémentation de la méthode run()

```
class MonThread implements Runnable {
    ...
    public void run() {
        ... code de l'activité ...
    }
}
class Principale {
    public static void main(String[] argv) {
        MonThread p = new MonThread(...);
        Thread t = new Thread(p);
        t.start();
        ...
}
```

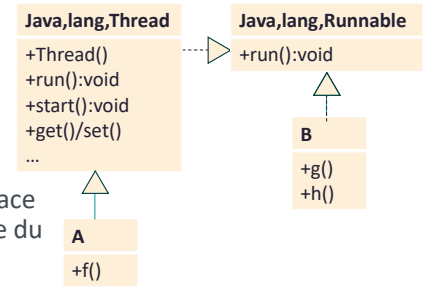
Classe Thread implement interface Runnable

La méthode run() est l'amorce du code sera exécuté par le Thread lorsque celui ci sera créé.

Lorsqu'une instance de Thread est créée, il faut lui indiquer l'amorce de code du thread qui sera contrôlé par cette instance

Les threads en Java

- **Comment créer un thread ?**
- Deux méthodes :
  - ✓ En héritant la classe Thread et en définissant sa méthode run(), ou
  - ✓ En instanciant le thread avec un objet Runnable
- La méthode run(): lorsqu'un objet implémentant interface Runnable est utilisé pour créer un thread, le démarrage du thread entraîne l'appel de la méthode d'exécution de l'objet dans ce thread existant séparément
- La méthode start() est appelée une seule fois sur un thread



Les threads en Java

- Exemple 1 :

```
public class DemoThread  implements  Runnable{
    public void run() {
        for(int i=0;i<1000;i++) {
        System.out.println(" hey thread1 started");
        }
    }
    public static void main(String[] args) {
        DemoThread d=new DemoThread();
        Thread t1=new Thread(d);
        t1.start();

        DownloadThread down =new DownloadThread();
        Thread t2=new Thread(down);
        t2.start();
    }
}
```

Implements Runnable interface

Creating instance of the class

Creating instance of thread class and starting the thread

Le Thread t va contrôler l'amorce DemoThread

Les threads en Java

- Exemple 2 :

```
public class PlayMusic extends Thread {

    public void run() {
        for(int i=0;i<1000;i++) {
            System.out.println("Music Playing ..... ");
        }
    }


    public static void main(String Args[])
    {
        PlayMusic p=new PlayMusic();
        p.start();

for(int i=0;i<1000;i++) {
        System.out.println("coding");
    }
}
}
```

Extends Thread Class

Creating instance of the class

Starting the thread

Le contrôleur et l'amorce sont un seul objet PlayMusic

Les threads en Java

❑ La classe java.lang.Thread modélise un thread :

- ✓currentThread() : donne le thread actuellement en cours d'exécution
- ✓setName() : fixe le nom du thread
- ✓getName() : nom du thread
- ✓isAlive() : indique si le thread est actif (true) ou non(false)
- ✓start() : lance l'exécution d'un thread et invoque la méthode run()
- ✓run() : méthode exécutée automatiquement après que la méthode start ait été exécutée
- ✓sleep(n) : arrête l'exécution d'un thread pendant n millisecondes
- ✓join() : opération bloquante - attend la fin du thread pour passer à l'instruction suivante

Les threads en Java

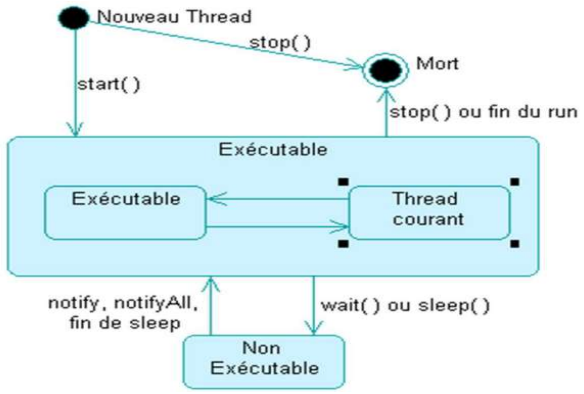
❑ Les constructeurs sont :

- Thread() : crée une référence sur une tâche asynchrone. Celle-ci est encore inactive. La tâche créée doit posséder la méthode run : ce sera le plus souvent une classe dérivée de Thread qui sera utilisée
- Thread(Runnable object) : l'objet Runnable passé en paramètre qui implémente la méthode run().

Req: si vous invoquez la méthode run (au lieu de start) le code s'exécute bien, mais aucun nouveau thread n'est lancé dans le système. Inversement, la méthode start, lance un nouveau thread dans le système dont le code à exécuter démarre par le run.

Les threads en Java

❑ les différents états ainsi que les différentes transitions possibles d'un cycle de vie d'un Thread:



Les threads en Java

❑ **Démarrage d'un thread**

```
// premier test
public class TestThread extends Thread {
  ...
 public TestThread() { }

public void run() {

public static void main(String[] args) {
        TestThread t1 = new TestThread();
        t1.start();
} }
```

Lorsque le thread a fini d'exécuter la méthode ***run()*** il entre dans un état **dead** et ne peut pas être relancé par la méthode ***start()***. Cependant, l'instance est toujours présente ce qui permet d'avoir des informations sur l'état du thread.

Les threads en Java

❑ Suspension d'un thread

❑ La méthode statique *sleep(long):*

- ✓ Elle veut dire patienter ou **endormir pendant un certain temps**, va permettre de **suspendre l'exécution d'un thread**
- ✓ Elle prend comme paramètre un temps en milliseconde (un **long**) qui correspond au temps d'attente avant de poursuivre le déroulement du thread
- ✓ Dans le cas d'une animation graphique, pour éviter un affichage trop rapide, on utilise cette méthode
- ✓ Elle lance l'exception **InterruptedException** si le Thread est stoppé pendant son sommeil (par un appel de la méthode ***interrupt()*** par exemple)

Les threads en Java

❑ Suspension d'un thread

❑ La méthode wait*():*

- ✓ La méthode ***wait()*** provoque la suspension de l'exécution du thread dans lequel s'exécute le code courant
- ✓ ***wait(long):*** spécifier une durée
- ✓ Un appel aux méthodes ***notify()*** ou ***notifyAll()*** permet au **Thread** de sortir de cet état d'attente. S'il y a plusieurs threads en attente, c'est celui qui a été suspendu le plus longtemps qui est réveillé
- ✓ La méthode wait suspend l'exécution d'un thread, en attendant qu'une certaine condition soit réalisée. La réalisation de cette condition est signalée par un autre thread par les méthodes notify ou notifyAll
- ✓ Wait, notify et notifyAll sont définies dans la classe java.lang.Object et sont donc héritées par toute classe

Les threads en Java

❑ Suspension d'un thread

```
// premier test avec la classe thread
public class FirstThread extends Thread {
  private String threadName;
  public FirstThread(String threadName) {
      this.threadName = threadName;
      this.start();    }
   public void run() {
       try {  for(int i=0;i<5;i++) {
                  System.out.println( "Thread: " + this.threadName + " -  " + i );
                  Thread.sleep(30); // sleep peut déclencher une exception
      } } catch (InterruptedException exc) {
                    exc.printStackTrace();       }   }
 public static void main(String[] args) {
            FirstThread thr1  = new FirstThread("Toto");
} }
```

L'exécution :
Thread: toto 0
Thread: toto 1
Thread: toto 2
Thread: toto 3
Thread: toto 4

Les threads en Java

❑ Attendre la fin d'un autre Thread

❑ La méthode ***join()*** fait attendre la fin d'un thread par le thread en cours d'exécution

- ✓ Elle permet de s'assurer que le traitement du thread est terminé avant de poursuivre le déroulement du programme

```
Thread {
              run() {
               System.out.println("Début du long traitement");
             sleep(3000);
             System.out.println("Fin du long traitement");
             } (InterruptedException e) { e.printStackTrace(); } }
main(String[] args) {
             {
              Thread t = MyThread();
               t.start();
               System.out.println("On attend la fin du thread");
                t.join();
                System.out.println("Le thread a fini son exécution");
            }(InterruptedException e) { e.printStackTrace(); } }
```

InterruptedException est levée dans les cas suivants:

1. Un Thread t1 n'a pas fini son exécution
2. Un Thread t2 appelle t1.join()
3. t2.interrupt() est appelée

Les threads en Java

❑ Tester l'exécution d'un Thread

- ❑ La méthode ***isAlive()*** permet de savoir si un thread est toujours en vie
- ❑ On considère qu'un thread est vivant s'il a démarré (la méthode ***start()*** a été appelée) et qu'il n'est encore terminé (il n'est pas sorti de sa méthode ***run()***)
- ❑ Cette méthode retourne un booléen : **true** si le thread est dans sa méthode ***run()***, **false** dans les autres cas

```
Boolean etat;
Thread t1 = Thread();
etat = t1.isAlive
```


Les threads en Java

❑ Arrêt d'un Thread

- ❑ Pour provoquer l'arrêt d'un thread, il est conseillé de faire une vérification périodique d'une variable de celui-ci : il peut s'agir d'un **boolean** que vous mettrez à **true** lorsqu'il est nécessaire de stopper le thread. Ainsi, en testant et prévoyant proprement cette condition, vous pouvez libérer proprement les ressources utilisées par le thread ainsi que les verrous posés et terminer l'exécution de la méthode ***run()***
- ❑ Si votre thread procède à des attentes (à l'aide de ***wait()***, ***join()*** ou ***sleep()***), il ne sera pas toujours possible de procéder de cette manière. Dans ces cas-là, vous pouvez avoir recours à la méthode **interrupt**() qui provoque la levée d'une exception **InterruptedException** ce qui vous permettra de libérer proprement les ressources dans un bloc **catch** avant de terminer la méthode **run**()

Les threads en Java

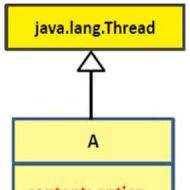
❑ Arrêt d'un Thread

- ❑ De même, si votre thread est bloqué sur une opération d'entrées/sorties sur un canal interruptible (**java.nio.channel.InterruptibleChannel**), il est aussi possible d'utiliser la méthode ***interrupt()*** qui fermera celui-ci et lèvera une exception **ClosedByInterruptException**
- ❑ Il existe une méthode ***stop()*** qui n'a été conservée que pour des questions de compatibilité mais celle-ci est fortement **déconseillée** (***deprecated***) : en effet celle-ci provoque l'arrêt brutal du thread et ne permet pas de vérifier que toutes les opérations de libération des ressources ou verrous s'effectuent correctement

Synchronisation des threads

Synchronisation des threads


```
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);
}
}
```




```
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 975 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 incorrecte

Synchronisation des threads

- Dans le cas ou il s'agit d'un accès concurrent à une ressource, des **problèmes de synchronisation** peuvent se poser
- Java utilise le **mécanisme de verrouillage**
- Identifier les **parties critiques** de code
- Un seul thread à la fois qui doit accéder à ces parties
- 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 qui s'appelle **moniteur**
- Tant que un thread ne relâche pas le verrou (lock) sur un objet, les autres ne peuvent pas accéder à ces parties critiques

T1
T2

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


Synchronisation des threads

- La synchronisation est un mécanisme qui coordonne l'accès aux données communes et au code critique par des threads
- Seulement la méthode toute entière ou l'une de ces parties être synchronisée
- La synchronisation est assurée par le mot clé « synchronized »
- Synchroniser une méthode en sa totalité est le moyen le plus simple pour assurer qu'un seule thread accède à un code critique
- Parfois, seulement une partie du code qui a besoin d'être protégée. Dans ce cas, la partie du code à protéger est synchroniser avec un objet
- Le thread se met en attente sur un objet par la méthode wait()

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

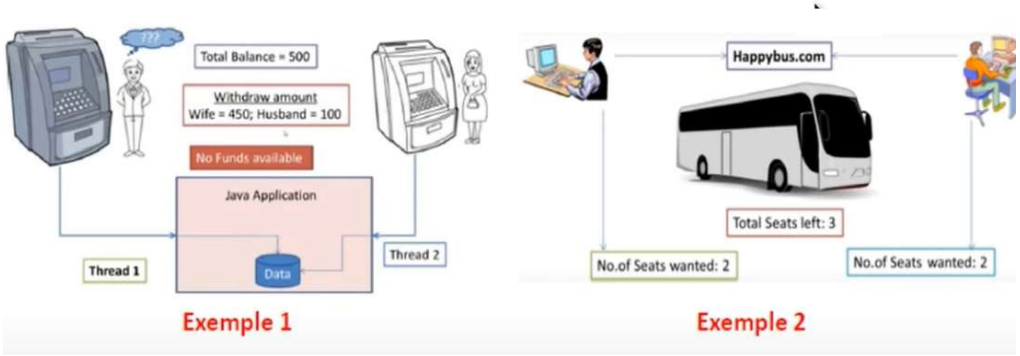
```
private synchronized void  incrementer() {
content++    ...

}
```


Synchronisation des threads

- Une mauvaise gestion de la synchronisation entre blocs peut mener à une situation de blocage total d'une application appelée deadlock
- Une deadlock intervient lorsqu'un premier thread "T1" se trouve dans un bloc synchronisé "B1", et est en attente à l'entrée d'un autre bloc synchronisé "B2". Dans ce bloc "B2", se trouve déjà un thread "T2", en attente de pouvoir entrer dans "B1"
- Ces deux threads s'attendent mutuellement, et ne peuvent exécuter le moindre code.
- La situation est bloquée, la seule façon de la débloquer est d'interrompre un des deux threads

Synchronisation des threads



Gestion des threads

Gestion des threads

- Pour le système, gérer des threads, c'est …

Gérer le partage du temps
+
Gérer des priorités

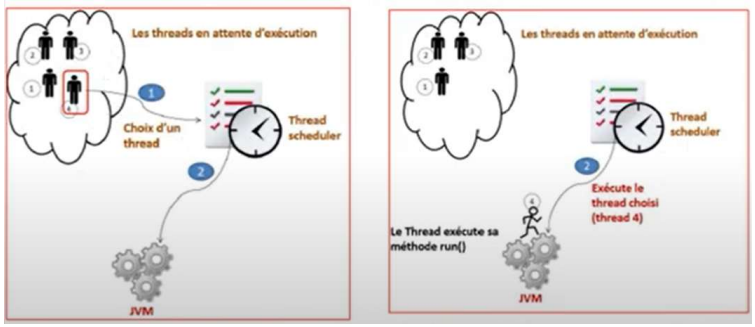
- A une unité de temps donnée, un seul thread qui s'exécute
- Quand il s'agit de plusieurs Threads qui s' exécutent en parallèle, le JVM décide quel thread exécuter pour une unité de temps (Threads scheduler algorithm)
  - ✓ L'implémentation de l'algorithme dépend de l'implémentation de la JVM et on ne peut pas la changer
  - ✓ Mais on peut influencer sur le temps alloué à chaque thread à l'aide des méthodes

Gestion des threads

- Le développeur ne peut pas prévoir l'ordre d'exécution: se fait de maniéré aléatoire par le JVM
- Thread scheduler choisit un thread à exécuter parmi plusieurs Threads
- Chaque thread change d'état entre running et en attente jusqu'à terminer son traitement
- A tout moment, quand plusieurs threads sont «runnables», le thread de plus haute priorité est choisi
  - ✓Égale à NORM_PRIORITY (5)
  - ✓Définie entre MIN_PRIORITY (1) et MAX_PRIORITY (10)
- **Si ce thread stoppe ou est suspendu pour une raison quelconque, alors, un thread de priorité inférieure peut s'exécuter**

Gestion des threads

- Le thread N°4 est prêt à exécuter – choisi par la JVM – exécuté pour un temps défini
- La JVM le rend à l'ensemble des threads en attente
- Le JVM choisit un autre Thread prêt à s'exécuter…



Quizz!

Quizz!

- Qu'est ce qu'un Thread ?

Quizz!

- On peut changer le nom d'un thread principal. Vrai ou faux?

Quizz!

- Deux threads peuvent avoir le même nom. Vrai ou faux?

Quizz!

- Quel sera le résultat du programme suivant ?

```
public class JavaThreadsQuiz
{
    public static void main(String[] args)
    {
        String name = Thread.currentThread().getName();

        System.out.println(name);
    }
}
```


Quizz!

- Quel sera le résultat du programme suivant ?

```
    public static void main(String[] args)
    {
        Thread thread = new Thread();

        thread.setPriority(12);

        thread.start();
    }
}
```

Quizz!

- Qu'est-ce qu'une condition de concurrence?
  - a. Un problème lié à l'utilisation excessive de la mémoire.
  - b. Une situation où le résultat d'une opération dépend de l'ordre d'exécution des threads.
  - c. Un type d'erreur de syntaxe dans le code.
  - d. Un état où un thread est en attente indéfinie d'une ressource.

Quizz!

- Quel mécanisme est utilisé pour éviter les conditions de concurrence entre threads?
  - a. Sémaphore
  - b. Verrou (lock)
  - c. File d'attente (queue)
  - d. Variable conditionnelle

Quizz!

- Quel est le risque lorsque plusieurs Thread accèdent et modifient les mêmes données ?

Quizz!

- Quel est le risque lorsque plusieurs Thread accèdent et modifient les mêmes données ?

- Que faire ?