

# Concurrence

Processus simultanés





# Objectifs du cours

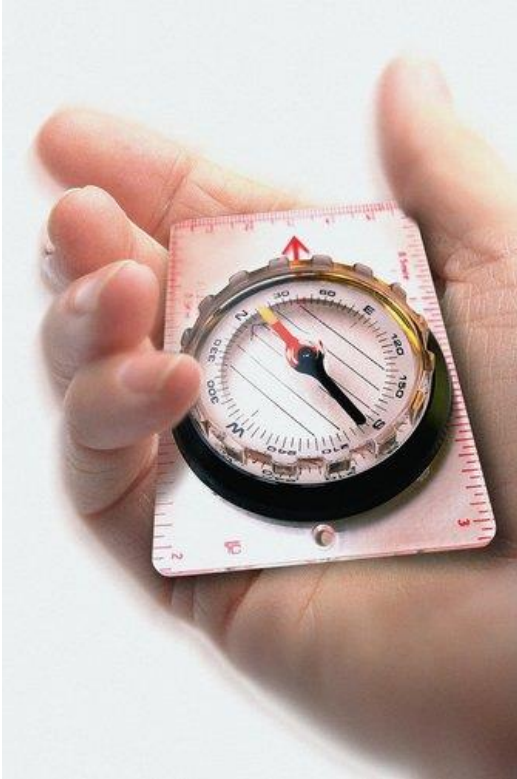
En complétant ce cours, vous serez en mesure de:

- Expliquer ce qu'est un thread
- Expliquer ce que sont les impasses (deadlock) et la pénurie de ressources
- Créer des Threads
- Les Gérer



Concurrence

# Plan de cours



- Introduction
- Utilisation de base
- Mettre en pause et arrêter les threads
- Synchronisation et serrures

Concurrence

# INTRODUCTION



# Programmation concurrente

- Les utilisateurs d'ordinateurs peuvent faire plus d'une chose à la fois
- Ils peuvent continuer à travailler dans un traitement de texte, tandis que d'autres applications téléchargent des fichiers et diffusent de l'audio
- Même le traitement de texte doit toujours être prêt pour les événements clavier et souris
  - Peu importe à quel point il est occupé à reformater le texte ou à mettre à niveau l'affichage
- Logiciel qui peut faire de telles choses :
  - Programmation concurrente !



# Processus et threads

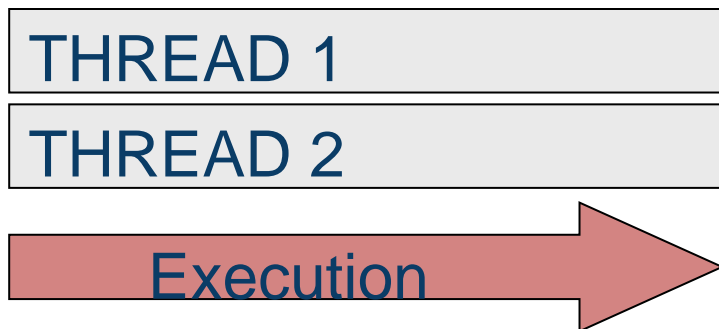
- Deux unités d'exécution de base : les processus et les threads
- Un processus a un environnement d'exécution autonome
  - Possède son propre espace mémoire privé
- Threads ou processus légers
  - Créer un nouveau thread nécessite moins de ressources que de créer un nouveau processus
  - Existe dans un processus
  - Partager les ressources du processus
- Durant ce cours, nous allons nous concentrer sur les Threads !



# Ordonnancement du processeur

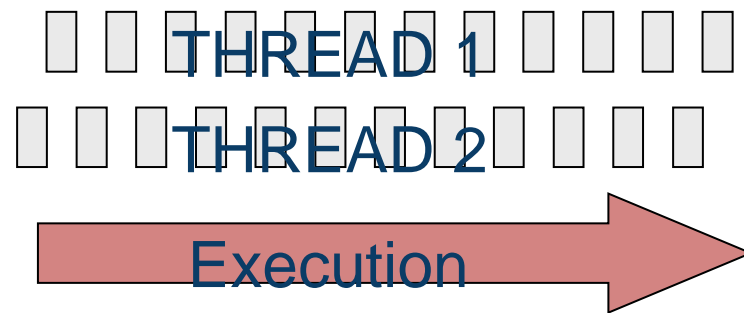
- Comment sont exécutés les threads par un ordinateur à processeur unique :

## Application



Illusion :  
Exécution simultanée

## Processor



En vrai :  
Exécution alternative



# Avantages

- Permet d'exécuter des tâches « simultanées »
- Les actions ne sont pas exécutées les unes après les autres
- Faire une boucle sans fin
- Utilisez-le pour créer des sockets
- Crée des processus enfants
- Appartient à un groupe de threads
- ...





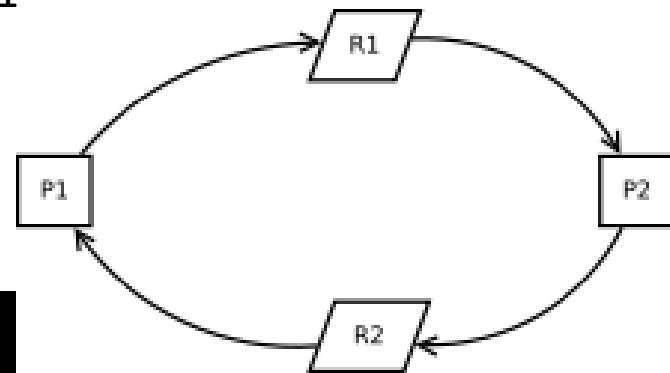
# Contraintes

- Le développeur doit :
  - Concevoir correctement son programme
  - Savoir si le comportement de l'application est garanti
  - Prévenir les bogues
- L'ordre d'exécution n'est jamais prévisible
- Les problèmes courants avec Concurrency sont :
  - Impasse (deadlock)
  - Famine de ressources



# Definition: **Deadlock**

- Situation dans laquelle deux ou plusieurs actions concurrentes attendent chacune la fin de l'autre...
  - ... et donc ni l'un ni l'autre ne le fait jamais
- Exemple :
  - Les deux processus ont besoin des deux ressources
    - P1 nécessite des ressources supplémentaires R2
    - P2 nécessite une ressource supplémentaire R1
  - Aucun processus ne peut continuer



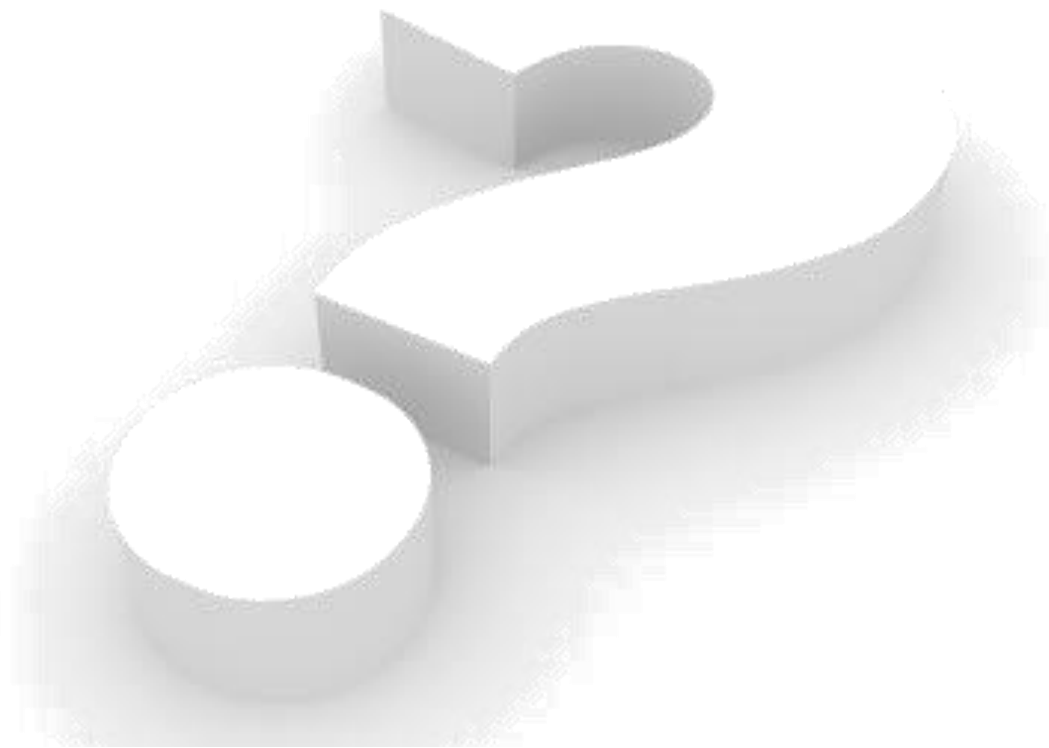


# Definition: **Famine** de ressources

- Situation où un processus est perpétuellement privé des ressources nécessaires sans lesquelles le programme ne peut jamais terminer sa tâche
- Exemple :
  - Un thread veut une ressource et vérifie à intervalles réguliers si elle a été libérée
  - D'autres threads veulent aussi la même ressource
  - Avec un peu de chance, le fil n'attendra pas trop longtemps
  - Sans chance, d'autres threads prendront la ressource avant chaque vérification et cela attendra...
  - ... peut-être pour toujours !



# Questions ?



Concurrence

# UTILISATION DE BASE



# Classes Java

- La plate-forme Java fournit :
  - Classe de Thread
  - Interface exécutable (**Runnable**)
- Les objets Thread représentent des threads que vous pouvez exécuter
- Les objets exécutables (**Runnable**) représentent ce qu'un thread exécutera

**MyRunnable**

```
// Do something
```

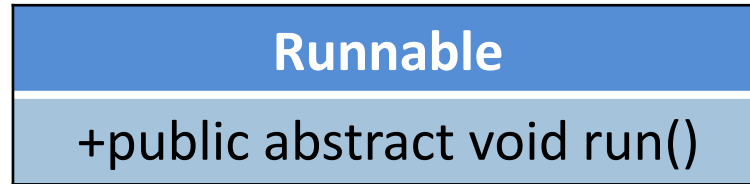
**Thread**





# Interface Runnable

- **Runnable** est une interface simple qui définit une seule méthode



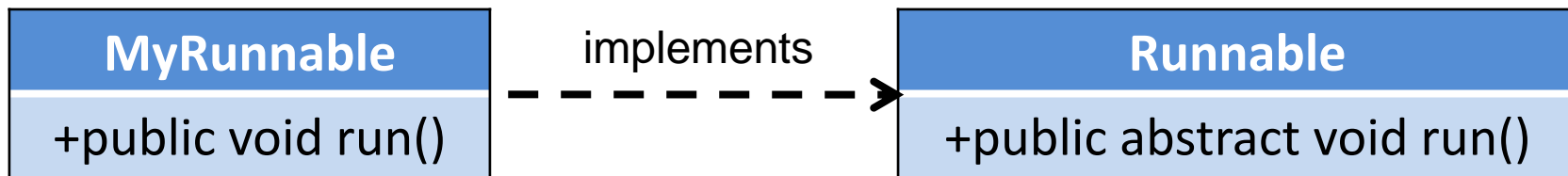
- Pour obtenir des objets **Runnable** vous devez :
  - Créer une nouvelle classe implémentant **Runnable**
  - Implémentez la méthode run() avec ce que vous voulez que le **thread** fasse
  - Créez des instances de votre nouvelle classe



# Interface Runnable

- Exemple de classe Runnable :

```
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        // Do something  
    }  
}
```







# Interface Runnable

- Exemple de création d'un Thread :

```
MyRunnable myRunnable = new MyRunnable();  
Thread thread = new Thread(myRunnable);  
thread.start();
```

- Simulation :

**MyRunnable myRunnable**

```
run() {...}
```

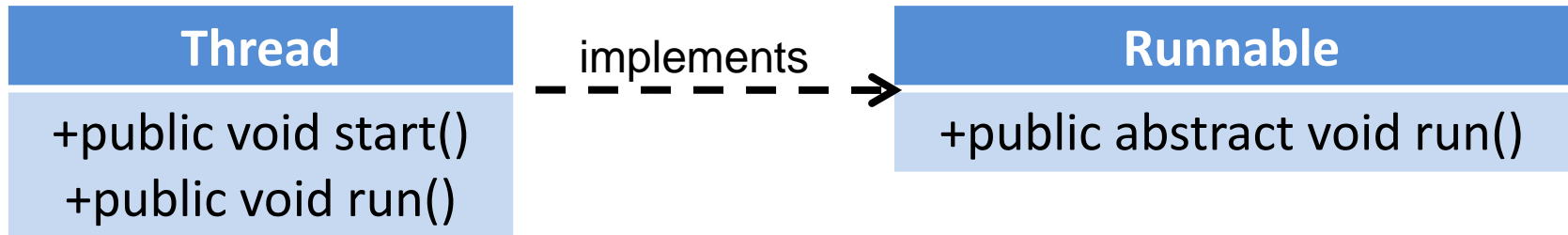
**Thread thread**

```
start() {...}
```



# Classe Thread

- Comme nous l'avons vu, les objets Thread représentent des threads exécutables potentiels et ont besoin d'un objet **Runnable**
- Mais un objet Thread peut aussi être exécuté sans **Runnable** !
- En effet, la classe Thread implémente l'interface **Runnable**
  - Ainsi, vous pouvez sous-classer Thread et remplacer la méthode run ()

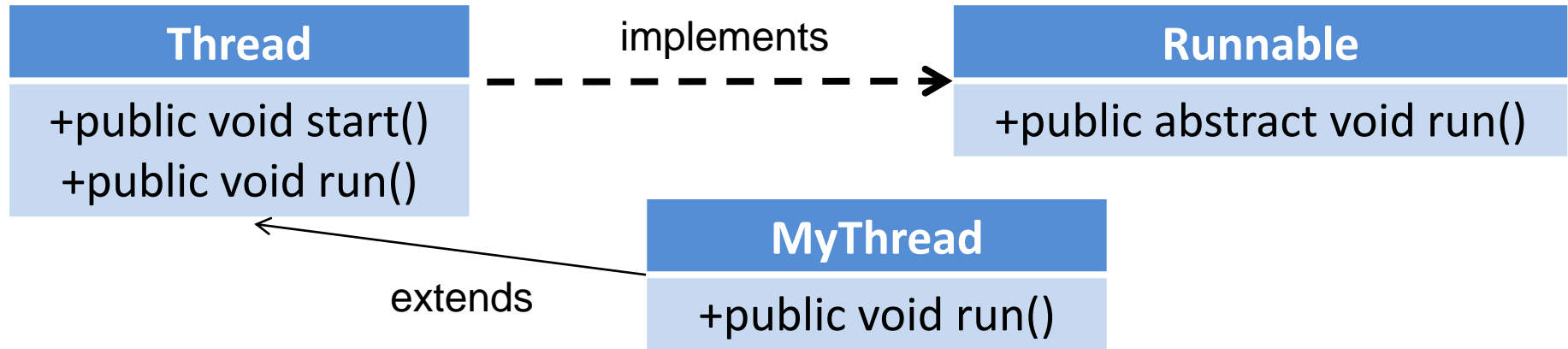




# Classe Thread

- Exemple de Thread personnalisé sans Runnable:

```
public class MyThread extends Thread {  
    @Override  
    public void run () {  
        // Instructions  
    }  
}
```





# Classe Thread

- Exemple de création d'un Thread :

```
MyThread thread = new MyThread();  
thread.start();
```

- Simulation :

**MyThread thread**

```
run() {...}
```

**Thread**

```
start() {...}
```



# Utiliser un Thread

- Pour démarrer un Thread :
  - Utilisez toujours et uniquement la méthode `start()`

```
MyThread thread = new MyThread();  
thread.start();
```

- La méthode `start ()` invoquera la méthode `run ()` dans un thread différent
- Si vous appelez directement la méthode `run()`, elle sera exécutée dans le même thread et non dans un nouveau



# Utiliser un Thread

- Pour démarrer un Thread :

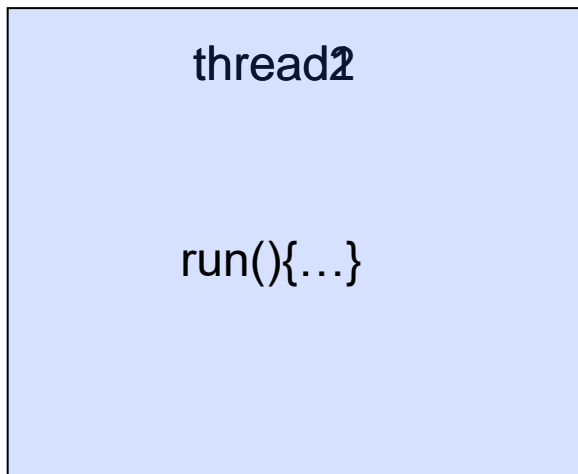
```
MyThread thread1 = new MyThread();  
MyThread thread2 = new MyThread();  
thread1.start();  
thread2.start();
```



# Utiliser un Thread

- Simulation :

MyThread



`run(){...}`



`run(){...}`



# Avec ou sans Runnable ?

- Vous avez ici deux responsabilités :
  - **Le Runnable** qui est responsable d'un comportement à exécuter
  - **Le Thread** qui est chargé de gérer le thread qui exécutera le Runnable





# Avec ou sans Runnable ?

- Bon... Et alors ?
  - La séparation des préoccupations permet :
    - Une meilleure encapsulation
    - Une meilleure factorisation du code
    - Une amélioration de la maintenabilité
- Des exemples simples ?
  - Un **Runnable** personnalisé peut étendre une classe
    - Un Thread personnalisé ne peut pas
  - Un **Runnable** personnalisé peut être utilisé par différents types d'objets Thread



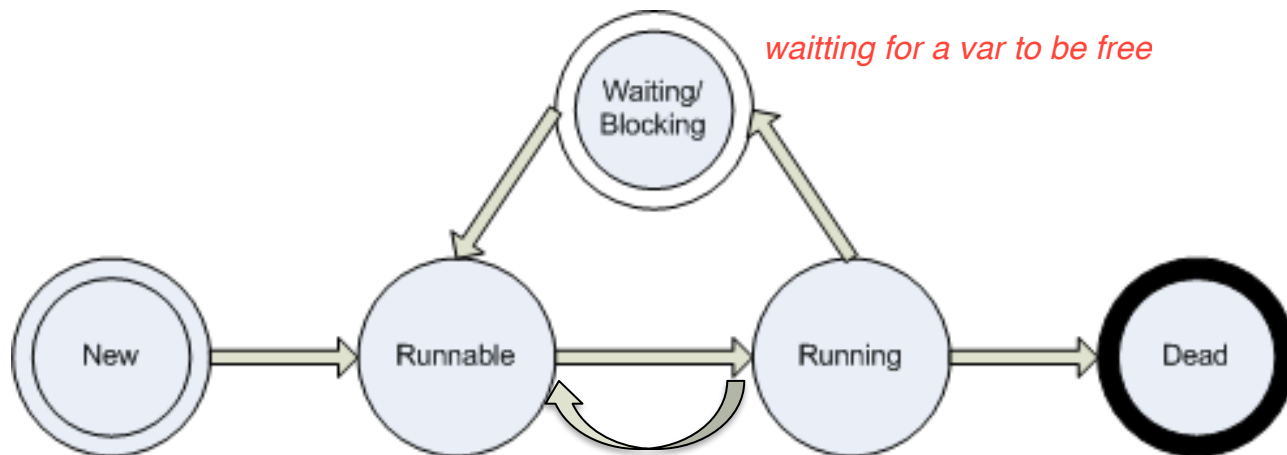
# États des threads

- Un thread peut être dans l'un des états suivants :
  - **NEW** : Pas encore démarré mais déjà instancié
  - **RUNNABLE** : le thread est prêt à s'exécuter
  - **RUNNING** : le thread est en cours d'exécution
  - **BLOCKED** : En attente d'un bloc moniteur (bloc synchronisé)
  - **WAITING** ou **TIMED\_WAITING** : Attente d'une action après un `join()`, `wait()` ou `sleep()`
  - **TERMINATED** : le fil de discussion est terminé
- Obtenir l'état du thread :
  - `getState()`



# États des threads

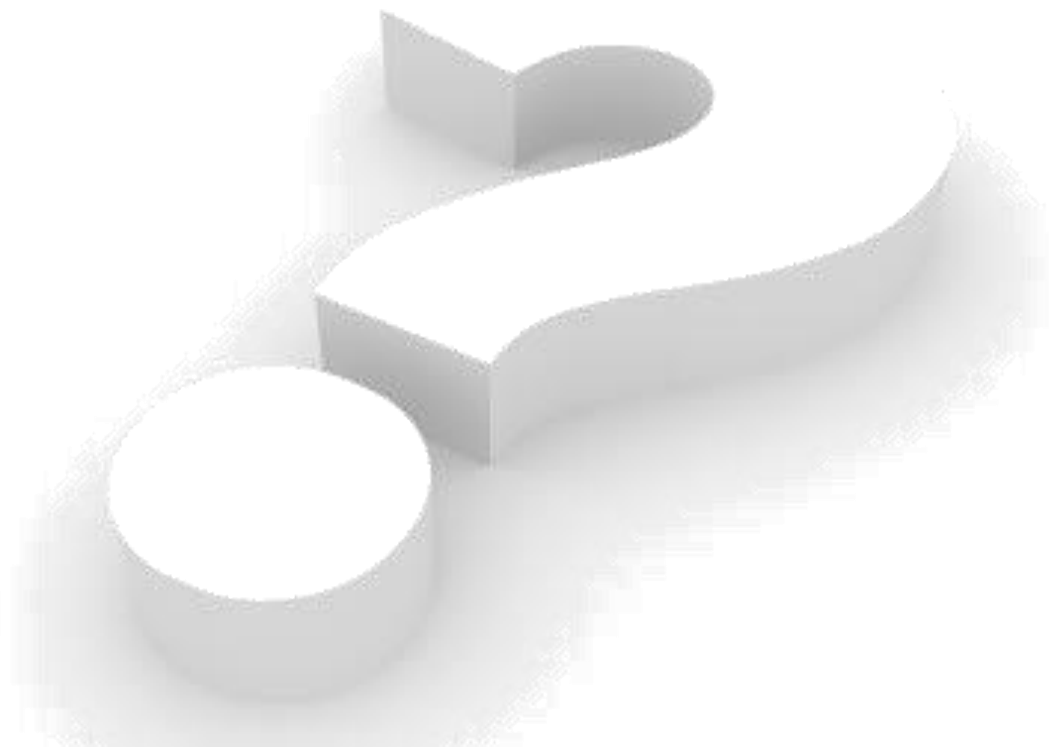
- Transition entre les états de thread :



- Exécutable et en cours d'exécution -> **RUNNING**
- Attente/blocage -> **BLOCKED, WAITING** et **TIMED\_WAITING**
- Mort -> **TERMINATED**



# Questions ?



Concurrence

# **METTRE EN PAUSE ET ARRÊTER LES THREADS**



METTRE EN PAUSE ET ARRÊTER LES THREADS

## Mettre un Thread en pause

- Il est possible de mettre un thread en pause
- Pour des raisons différentes :
  - Attendre qu'une opération sur un compte bancaire soit terminée pour en faire une autre
  - Attendre qu'un autre thread s'arrête pour écrire dans un fichier
- 4 façons de le faire : **sleep()**, **yield()**, **join()**, **wait()**



## METTRE EN PAUSE ET ARRÊTER LES THREADS

# Mettre un Thread en pause

- Thread.sleep(long milliseconds) :
  - Provoque la suspension de l'exécution du thread actuel pendant une période spécifiée
- Exemple :

```
public class RunningThread extends Thread {  
    @Override  
    public void run() {  
        try {  
            for (int i = 0; i < 3; i++) {  
                Thread.sleep(1000);  
                System.out.println(getName()+" is running");  
            }  
        } catch (InterruptedException e) { ... }  
    }  
}
```



METTRE EN PAUSE ET ARRÊTER LES THREADS

# Mettre un Thread en pause : yield

- Thread.yield() :
  - Provoque une pause temporaire de l'objet thread en cours d'exécution et permet à d'autres threads de s'exécuter
- Exemple :

```
@Override
public void run() {
    while (!Thread.isInterrupted()) {
        System.out.println(getName() + " is running");
        Thread.yield();
    }
}
```





METTRE EN PAUSE ET ARRÊTER LES THREADS

## Mettre un Thread en pause : join

- join() :
  - Arrête le thread en cours jusqu'à la fin du thread invoquant la méthode join
- join(long millisecond) :
  - Arrête le thread en cours pendant une durée maximale spécifiée par le thread invoquant la méthode de jointure, puis le thread en cours continue



# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Utiliser un Thread: join

- Example:

```
public static void main(String[] args) {  
    RunningThread rt = new RunningThread ();  
    rt.start();  
    try {  
        System.out.println("Waiting for Thread-0");  
        rt.join();  
    } catch (InterruptedException e) {  
        System.out.println("Has been interrupted");  
    }  
    System.out.println("Continues");  
}
```



## METTRE EN PAUSE ET ARRÊTER LES THREADS

# Interruptions

- Une interruption indique à un thread qu'il doit arrêter ce qu'il est en train de faire et faire autre chose
- Pour envoyer une interruption, vous pouvez invoquer la méthode **interrupt()** sur l'objet thread
  - Cette méthode n'arrête pas un thread mais lui ajoute un indicateur d'état d'interruption
- En fonction de ce que vous avez défini dans la méthode `run()`, le Thread peut gérer cet indicateur et l'arrêter ou simplement l'ignorer



# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Interruptions

- Comment à l'intérieur du thread savoir quel est l'état de ce flag ?
  - Vous avez deux façons :
    - Utilisez la méthode **Thread.interrupted()**
- Exemple:

```
for (int i = 0; i < inputs.length; i++) {  
    doSomething(inputs[i]);  
    if (Thread.interrupted()) {  
        // We've been interrupted: no more crunching.  
        return;  
    }  
}
```



METTRE EN PAUSE ET ARRÊTER LES THREADS

# Interruptions

- Comment à l'intérieur du thread savoir quel est l'état de ce flag ?
  - Vous avez deux façons :
    - **Catch InterruptedException**
      - Cette exception peut être levée par la plupart des méthodes de pause comme **sleep()**
      - Il interrompt la pause et passe directement à la clause catch



# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Interruptions

- Example :

```
for (int i = 0; i < importantInfo.length; i++) {  
    // Pause for 4 seconds  
    try {  
        Thread.sleep(4000);  
    } catch (InterruptedException e) {  
        // We've been interrupted: no more messages.  
        return;  
    }  
    System.out.println(importantInfo[i]);  
}
```



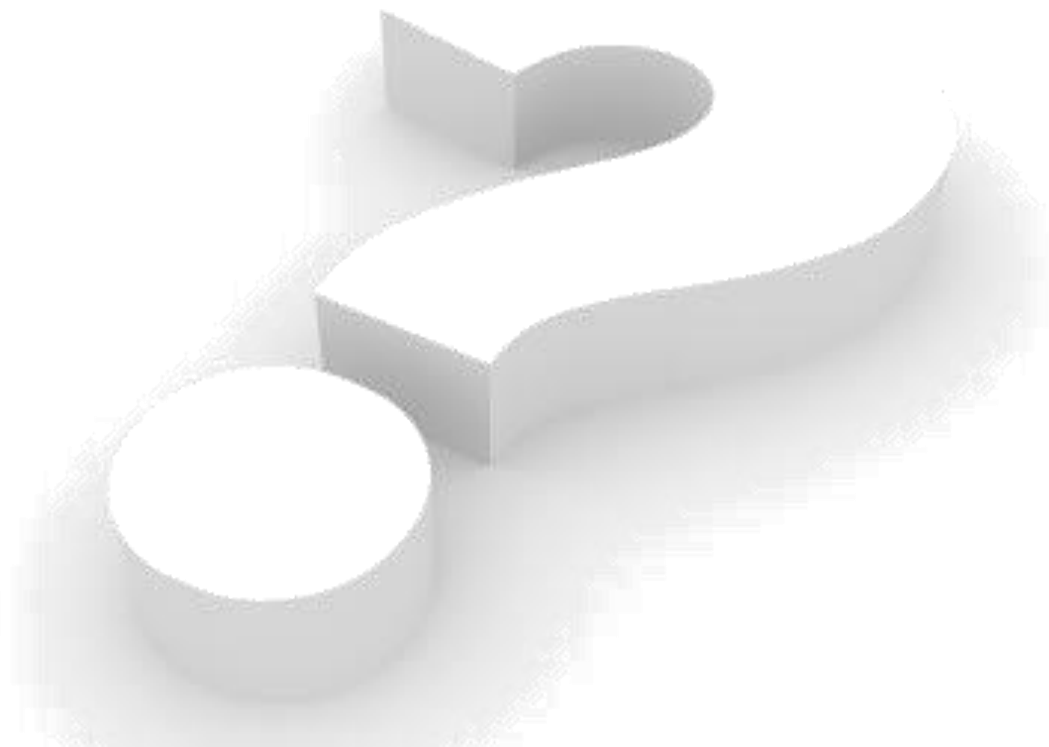
# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Interruptions

- Mais pourquoi utiliser ce flag d'interruption ? Pourquoi un thread ne peut pas simplement demander à un autre de s'arrêter ?
  - Parce que vous ne savez pas ce que fait le Thread !
  - Arrêter directement un thread peut avoir des effets dramatiques
    - Et si vous l'arrêtiez quand il écrit dans un fichier ?
- Il est plus sûr de demander au thread de s'interrompre (s'il le souhaite)
- Sinon, le moyen le plus simple d'arrêter un thread est simplement d'attendre qu'il se termine pour exécuter sa méthode `run()`



# Questions ?







# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Exercise (1/6)

- Connaissez-vous le problème des philosophes de la restauration ?
  - Rassurez-vous, cela n'a rien à voir avec la philosophie 😊
  - C'est juste un exemple illustratif d'un problème informatique courant dans la programmation concurrente
    - Et nous allons le résoudre en Java !





# METTRE EN PAUSE ET ARRÊTER LES THREADS

## Exercise (2/6)

- Explications du problème :
  - Cinq philosophes silencieux assis à une table circulaire faisant l'une des deux choses suivantes :
    - Manger ou penser
  - Parce qu'ils mangent des spaghettis, ils ont besoin de deux fourchettes pour manger
    - Mais il n'y a qu'une seule fourchette par siège...
    - Donc chacun doit utiliser la fourchette de son voisin !
  - Chaque philosophe est représenté par un thread !



# METTRE EN PAUSE ET ARRÊTER LES THREADS

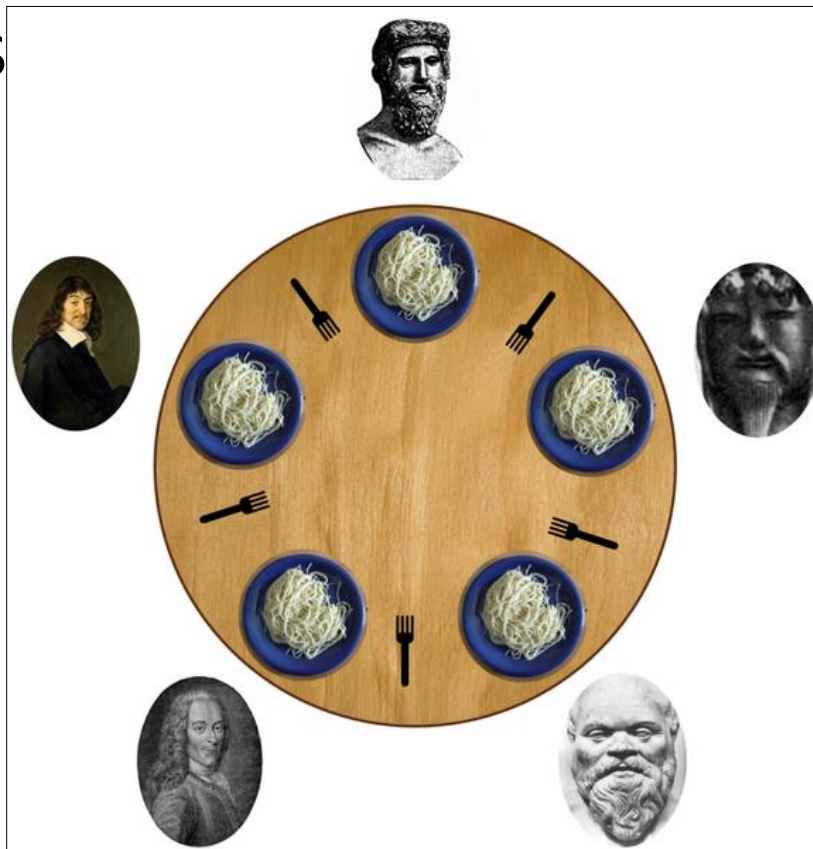
## Exercise (3/6)

- Quand un philosophe a faim, il essaie de prendre les fourchettes à côté de lui immédiatement
  - Si l'une d'elles est déjà utilisée, il attend que son voisin la libère
- Quand un philosophe affamé a deux fourchettes en mains et en même temps, il commence à manger
- Quand un philosophe finit de manger, il lâche ses fourchettes et recommence à réfléchir
- Les philosophes mangent et pensent pendant différentes durées aléatoires



# Exercise (4/6)

METTRE EN PAUS





## METTRE EN PAUSE ET ARRÊTER LES THREADS

### Exercise (5/6)

- Pour commencer, nous allons développer une version simple du programme sans gestion des fourchettes
- Créer un nouveau projet nommé Philosophes
  - Créer un package `com.cci.philosophers`
  - Créez quatre classes :
    - **Philosophe** : représenter un philosophe, doit implémenter `Runnable`
    - **Launcher** : une classe simple avec la méthode principale de l'application qui lance un thread par philosophes



## METTRE EN PAUSE ET ARRÊTER LES THREADS

### **Exercise (6/6)**

- Dans la méthode `run()` de la classe `Philosopher` :
  - Tant que le thread n'est pas interrompu, le philosophe doit:
    - Penser pendant un temps aléatoire
    - Manger à un moment aléatoire
    - Et encore...
- Afficher un message dans la console pour montrer l'état des philosophes

Concurrence

# **SYNCHRONISATION ET SERRURES**



Synchronisation et serrures

# Synchronisation

- Qu'est-ce que la synchronisation ?
  - Un système de serrure à code
  - Contrôler l'accès aux méthodes et aux attributs
  - Préserver la cohérence des données partagées
  - Éviter les comportements inattendus





# Attributs et méthodes synchronisés

- Imaginer...
  - Franck et Lucy ont chacun une carte de crédit pour le même compte bancaire
  - Ils vérifient toujours le compte avant de faire un retrait
  - Mais :
    - Franck fait un retrait de 30\$
    - Le compte n'est pas mis à jour immédiatement
    - Lucy fait un retrait de 40\$ mais avec un mauvais montant du compte en banque...
- Résultat :
- La banque perd une opération monétaire...

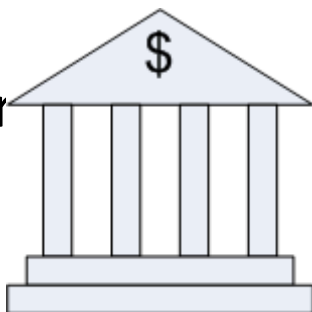


# Attributs et méthodes synchronisés

Synchronisation et s



Franck



Bank

Java  
Application

~~50€~~

~~20€~~

10€

Lucy à la mauvaise valeur du compte



Lucy

50€

20€

10€

30€

40€



# Attributs et méthodes synchronisés

- Solutions :
  - La deuxième opération attend que la première soit faite
  - Utiliser un mécanisme de synchronisation
  - Synchroniser le bloc d'instructions qui effectue l'opération
  - Verrouiller le compte
- En Java :
  - Utiliser l'objet **ReentrantLock** qui implémente l'interface Lock
  - Utiliser le mot-clé **synchronized**



# Lock et ReentrantLock

## – Lock interface :

- Contrôler l'accès aux ressources partagées
- Méthodes fournies :
  - **void lock()** : acquiert le verrou
  - **boolean tryLock()** : acquiert le verrou uniquement s'il est libre au moment de l'invocation
  - **void unlock()** : libère le verrou



# Lock et ReentrantLock

- Exemple :

– Le thread actuel possède le verrou :

```
ReentrantLock lock = new ReentrantLock ();  
// Return the total amount after the  
// withdrawal  
public void withdrawal (float cash) {  
    lock.lock(); Ne gère pas les prb de famille : le T1 peut locker cette fonction en boucle  
    credit -= cash;  
    debit += cash;  
    lock.unlock();  
}
```

```
Thread thread1 = new Thread(() -> {  
    for (int i = 0; i < 5; i++) {  
        account.withdrawal(100.0f);  
    }  
});
```



Synchronisation et serrures

# Synchronisation

- Java offre un mécanisme de synchronisation de bas niveau qui permet d'établir des points de rendez-vous de threads.
- Ce mécanisme repose sur les méthodes **wait()**, **notify()** de la classe `Object` et le mot-clé **synchronized** qui peut-être attaché à une méthode.



# Synchronisation

- Quand un premier thread entre dans une méthode **synchronized**, il attache le service *moniteur* à l'instance considérée ou à la classe si la méthode est **static**.

Si un second thread arrive pour exécuter une **methode synchronized**, il est bloqué jusqu'à la libération du *moniteur*. Ceci arrive quand le premier thread sort de la méthode ou pendant un **wait()** qui ne peut figurer que dans une méthode **synchronized**.



# Mot-clé **synchronized**

- Le mot-clé **synchronized** permet de définir un verrou implicite
- Peut être utilisé sur :
  - méthodes

```
public synchronized float withdrawal (int cash) {  
    credit -= cash;  
    debit += cash;  
    return credit + debit;  
}
```

- Objets (dans ce cas le thread obtient le lock sur l'objet account)

```
synchronized (account) {  
    // The credit value still > 0  
    if (account.getCredit() > 0) {  
        amount = account.withdrawal(cash);  
    }  
}
```





Synchronisation et serrures

# Les méthodes `wait` et `notify`

- La méthode `wait()` :
  - Méthode de la classe `Object`
  - Doit être invoqué dans un code synchronisé
  - Suspendre l'exécution du thread en cours
  - Libérer le verrou
  - Possibilité de spécifier la latence maximale en millisecondes



# Les méthodes wait et notify

- **wait()** :

```
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        try { Toutes les instance de type my runnable vont se synchroniser  
            synchronized(this) {  
                wait();  
            }  
        } catch (InterruptedException e) {  
            ...  
        }  
    }  
}
```



# Les méthodes `wait` et `notify`

- La méthode `notify()` :
  - Méthode de la classe `Object`
  - Réveille un thread en attente
- La méthode `notifyAll()` :
  - Méthode de la classe `Object`
  - Réveille tous les threads en attente



# Les méthodes wait et notify

- Exemple :

```
public class RunningThread extends Thread {  
    public void run() {  
        synchronized (this) {  
            try {  
                System.out.println("Child thread is waiting");  
                wait(); // Le thread attend jusqu'à ce qu'il soit notifié  
                System.out.println("Child thread is awake");  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



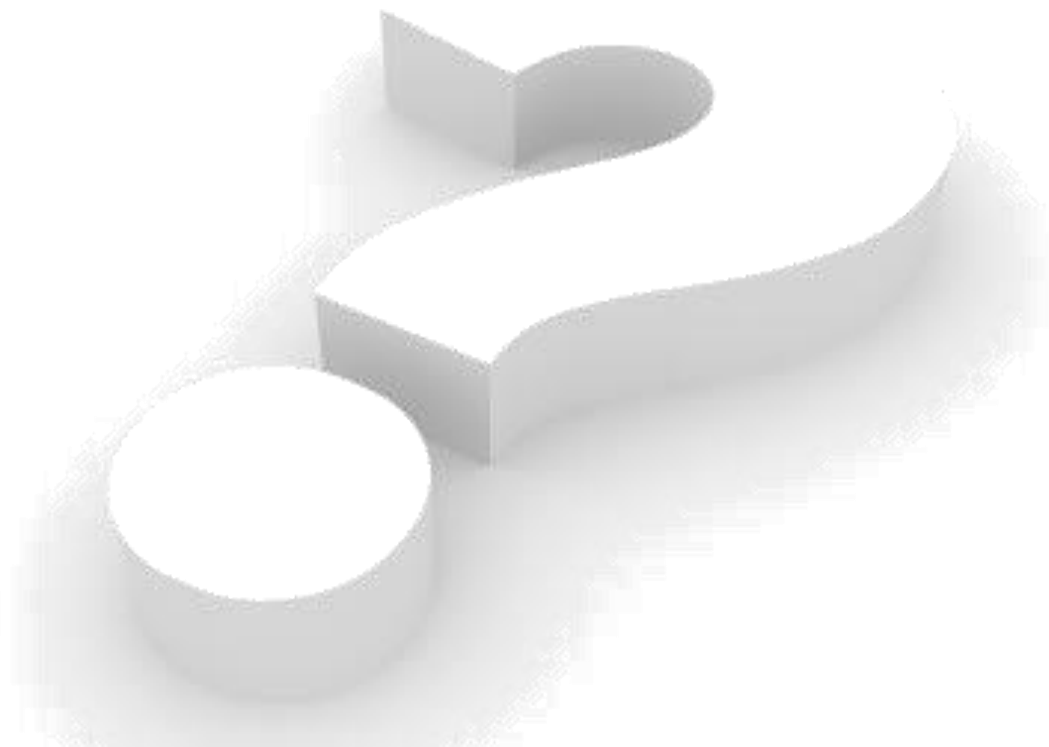
# Les méthodes wait et notify

- Exemple :

```
public static void main(String[] args) {  
    RunningThread rt = new RunningThread ();  
    rt.start();  
    try {  
        System.out.println("Main thread is waiting 2s");  
        Thread.sleep(2000);  
        synchronized (rt) { Sync with main thread  
            rt.notify(); réveille le thread rt  
        }  
    } catch (InterruptedException e) { ... }  
    System.out.println("The main thread continues");  
}
```



# Questions ?





# Quizz

Quel mot-clé utilise-t-on pour synchroniser un bloc d'instruction ?

**Synchronized**

Comment libérer un verrou dans un bloc synchronisé ?

**Avec la méthode wait()**

Comment réveiller un thread arrêté par la méthode wait() ?

**Avec la méthode notify()**

A quoi sert la méthode notifyAll() ?

**Pour réveiller tous les threads endormis**

Quelles conditions devons-nous respecter pour utiliser correctement la méthode wait() ?

**Utiliser wait() dans un try-catch et un bloc synchronized**



## Exercise (1/2)

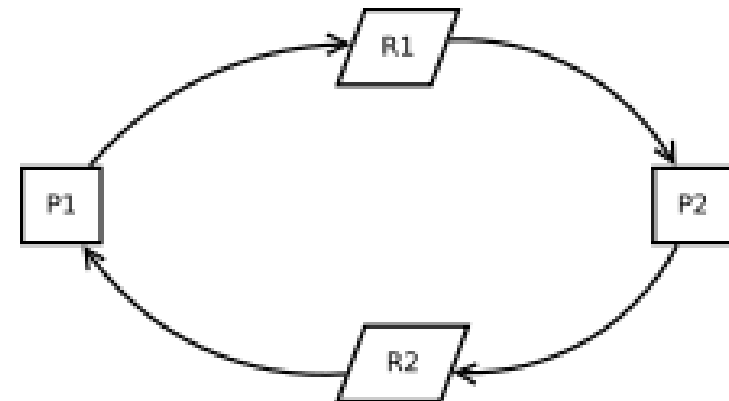
- Vous savez maintenant comment utiliser les verrous :
  - Créez deux nouvelles classes :
    - **DiningTable** : représente la table, composée des philosophes et des fourchettes
    - **Fork** : représente une fourchette





## Exercise (2/2)

- Pour manger, un philosophe doit prendre les deux fourchettes à côté de lui
- Une fourchette ne peut être utilisée que par un seul philosophe à la fois
- Votre solution doit éviter la famine et les impasses





**Fin**

*Merci de votre attention*