

Programmation Avancée et Application

Les threads

Jean-Guy Mailly

`jean-guy.mailly@parisdescartes.fr`

LIPADE - Université de Paris (Paris Descartes)

<http://www.math-info.univ-paris5.fr/~jmailly/>

1. Introduction

2. Les bases sur les threads en Java

Création et manipulation de threads en Java

Arrêter un thread

3. Ordonnancement et Synchronisation

4. Threads et JavaFX

Introduction

Threads et processus : généralités

- Processus : programme en cours d'exécution
- Chaque processus a son espace mémoire
 - pour le code
 - pour les données
 - pour les piles d'exécution (variables locales, adresses de retour des fonctions)
- Un processus = plusieurs threads (au moins un, à la création du processus)
- Les threads sont exécutés à tour de rôle
- Le choix du processus ou thread à exécuter (et la durée d'exécution) est choisi par le système d'exploitation
- Plusieurs threads = calcul « en parallèle »

Pourquoi des threads ?

- Meilleure répartition des tâches au sein d'un programme
- Exemple : un serveur
 - Sans threads : le serveur ne peut traiter qu'une requête à la fois

```
while(serveur.estActif()){  
    requete = serveur.ecouterRequetes()  
    serveur.traiterRequete(requete)  
}
```

Pourquoi des threads ?

- Meilleure répartition des tâches au sein d'un programme
- Exemple : un serveur

- Sans threads : le serveur ne peut traiter qu'une requête à la fois

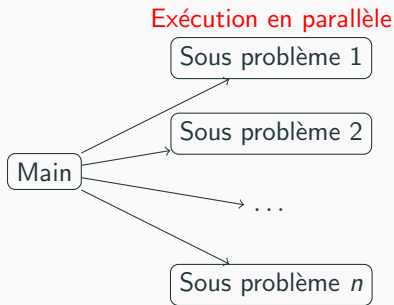
```
while(serveur.estActif()){  
    requete = serveur.ecouterRequetes()  
    serveur.traiterRequete(requete)  
}
```

- Avec threads : un thread est dédié à chaque requête, pas besoin d'attendre la fin d'un premier requête pour en traiter une autre

```
while(serveur.estActif()){  
    requete = serveur.ecouterRequetes()  
    t = creerThread()  
    t.traiterRequete(requete)  
}
```

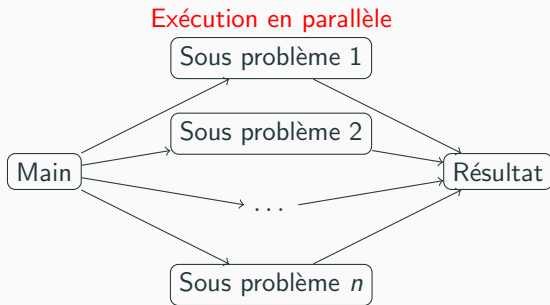
Pourquoi des threads ?

- Meilleure utilisation des ressources (par ex. : processeur multi-cores)
- On peut (parfois) découper un gros problème en sous-problèmes indépendants. Chaque sous problème est traité par un thread dédié, et à la fin le thread principal obtient son résultat en combinant les résultats des sous-problèmes



Pourquoi des threads ?

- Meilleure utilisation des ressources (par ex. : processeur multi-cores)
- On peut (parfois) découper un gros problèmes en sous-problèmes indépendants. Chaque sous problème est traité par un thread dédié, et à la fin le thread principal obtient son résultat en combinant les résultats des sous-problèmes



- Exécution de la JVM : un processus
- Exécution de la méthode main : un thread
- Autres threads :
 - le garbage collector
 - JavaFX Application
- On peut créer ses propres threads avec la classe `java.lang.Thread`

Premier exemple de Thread

- On souhaite créer un chronomètre : un thread dédié implémente un compteur à chaque seconde
- Réutilisation de la classe Compteur (cf cours JavaFX) :

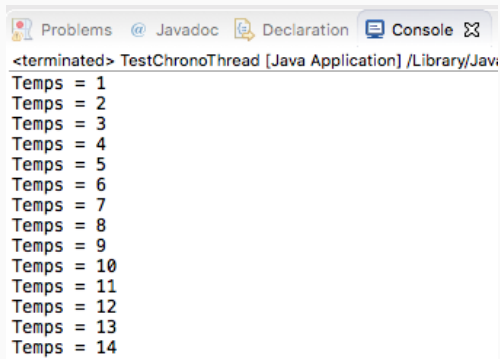
```
public class Compteur {  
    private int valeur ;  
  
    public Compteur() { valeur = 0 ; }  
  
    public void incrementer(int i) { valeur += i ; }  
  
    public void incrementer() { valeur++ ; }  
  
    public int getValeur() { return valeur ; }  
}
```

Premier exemple de Thread

```
public class ChronoThread extends Thread {  
    private Compteur cpt;  
  
    public ChronoThread() { cpt = new Compteur(); }  
  
    public void run() {  
        while (true) {  
            try {  
                sleep(1000);  
                compteur.incrementer();  
                System.out.println("Temps=" + cpt.getValeur());  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Premier exemple de Thread

```
public class TestChronoThread {  
    public static void main(String[] args) {  
        ChronoThread chrono = new ChronoThread();  
        chrono.start();  
    }  
}
```



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the Java application. The output starts with a terminated status message, followed by a series of lines indicating the number of temps (1 through 14).

```
<terminated> TestChronoThread [Java Application] /Library/Jav  
Temps = 1  
Temps = 2  
Temps = 3  
Temps = 4  
Temps = 5  
Temps = 6  
Temps = 7  
Temps = 8  
Temps = 9  
Temps = 10  
Temps = 11  
Temps = 12  
Temps = 13  
Temps = 14
```

Les bases sur les threads en Java

- Utilisation de la méthode `start`
 - Provoque la création d'un thread par le système
 - Appelle la méthode `run` du thread créé
- Utilisation de la méthode `run` :
 - **ne pas appeler directement la méthode `run` !**
 - Dans le cas contraire, elle est exécutée depuis le thread initial, et il n'y a pas de création d'un nouveau thread

Quelques méthodes de la classe Thread

Les constructeurs :

- **public** Thread(){...} permet de créer un Thread
- **public** Thread(Runnable r){...} permet de créer un Thread qui exécutera la méthode run de l'objet r

Quelques méthodes de la classe Thread

- **public void** run () {...} est la méthode exécutée par le thread
- **public void** start () {...} crée le thread, et appelle run
- **public boolean** isAlive () {...} teste si le thread est vivant (= la méthode run est en cours d'exécution)
- **public final** join () {...} permet d'attendre la fin de l'exécution du thread
- **public static void** sleep(**long** t) {...} rend le thread inactif pendant *t* millisecondes (dépend de la précision de l'horloge du système)
- **public void** setDaemon(**boolean** b) {...} permet d'indiquer que le thread est un démon, c'est-à-dire un thread qui n'empêche pas la JVM de s'arrêter. Cette méthode doit être utilisée avant l'appel à start ()

Attention : les méthodes join et sleep peuvent lancer une InterruptedException

Exemple : création de deux threads

```
public class ThreadAffichage extends Thread {  
    private String chaine;  
    private int max;  
  
    public ThreadAffichage(String str, int m) {  
        chaine = str;  
        max = m;  
    }  
  
    public void run() {  
        for (int i = 0; i < max; i++)  
            System.out.print(chaine + " ");  
    }  
}
```

Exemple : création de deux threads

```
public class TestAffichage {  
    public static void main(String[] args) {  
        ThreadAffichage t1 = new ThreadAffichage("1", 1000);  
        ThreadAffichage t2 = new ThreadAffichage("0", 1000);  
        t1.start();  
        t2.start();  
    }  
}
```

Un petit extrait du résultat :



```
<terminated> TestAffichage [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home  
1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0
```

L'interface Runnable

```
public interface Runnable {  
    public void run();  
}
```

- L'interface Runnable déclare uniquement la méthode run
- Au lieu de créer une classe fille de Thread, on peut créer une classe qui implémente Runnable, et utiliser
Thread t = **new** Thread(monRunnable)
pour créer un thread
- La méthode start du thread t lance dans ce cas la méthode run de l'objet monRunnable

Exemple d'utilisation de l'interface Runnable

Création d'une classe qui implémente Runnable

```
public class RunnableAffichage implements Runnable {  
    private String chaine;  
    private int max;  
  
    public RunnableAffichage(String str, int m) {  
        chaine = str;  
        max = m;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < max; i++)  
            System.out.print(chaine + " ");  
    }  
}
```

Exemple d'utilisation de l'interface Runnable

Création du Thread à partir du Runnable

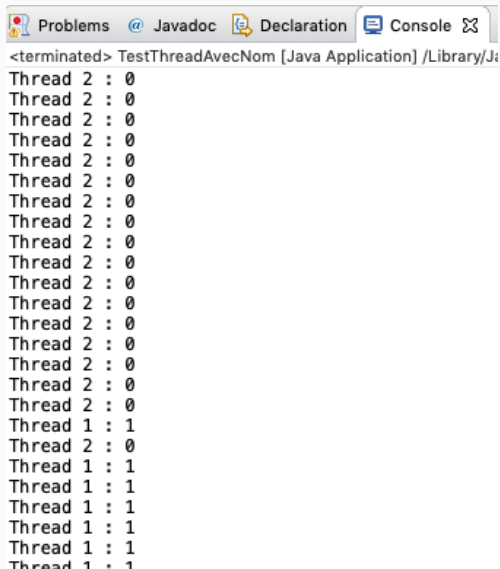
```
public class TestRunnableAffichage {  
    public static void main(String[] args) {  
        Runnable run1 = new RunnableAffichage("1", 1000);  
        Runnable run2 = new RunnableAffichage("0", 1000);  
        Thread t1 = new Thread(run1);  
        Thread t2 = new Thread(run2);  
        t1.start();  
        t2.start();  
    }  
}
```

Nommer des threads

On peut nommer un thread, et utiliser ce nom par la suite :

```
public class TestThreadAvecNom {  
    public static void main(String[] args) {  
        Thread t1 = new Thread("Thread_1") {  
            public void run() {  
                for (int i = 0; i < 1000; i++)  
                    System.out.println(getName() + "_:_1");  
            } };  
        Thread t2 = new Thread("Thread_2") {  
            public void run() {  
                for (int i = 0; i < 1000; i++)  
                    System.out.println(getName() + "_:_0");  
            } };  
        t1.start(); t2.start();  
    }  
}
```

Nommer des threads



The screenshot shows an IDE console window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application named 'TestThreadAvecNom'. The output shows a sequence of thread IDs. The first 17 lines are 'Thread 2 : 0', followed by 'Thread 1 : 1', and then a series of 'Thread 2 : 0' and 'Thread 1 : 1' alternating. The last line is partially cut off.

```
<terminated> TestThreadAvecNom [Java Application] /Library/Ja
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 2 : 0
Thread 1 : 1
Thread 2 : 0
Thread 1 : 1
Thread 1 : 1
Thread 1 : 1
Thread 1 : 1
Thread 1 : 1
Thread 1 : 1
```

Obtenir le thread en cours d'exécution

- La méthode statique `Thread.currentThread()` retourne l'objet de type `Thread` qui correspond au thread en cours d'exécution
- Cela permet, à tout moment, de déterminer (et manipuler) le thread en cours
- Exemple :

```
public class Util {  
    public static void printMessage(String message){  
        String name = Thread.currentThread().getName();  
        System.out.println(name + " : " + message);  
    }  
}
```


Obtenir le thread en cours d'exécution

```
public class TestThreadEnCours {  
    public static void main(String[] args) {  
        Thread t1 = new Thread("Thread_1") {  
            public void run() {  
                for (int i = 0; i < 1000; i++)  
                    Util.printMessage("Bonjour");  
            }  
        };  
        Thread t2 = new Thread("Thread_2") {  
            public void run() {  
                for (int i = 0; i < 1000; i++)  
                    Util.printMessage("Au_revoir");  
            }  
        };  
        t1.start(); t2.start();  
    }  
}
```

Obtenir le thread en cours d'exécution



```
<terminated> TestThreadEnCours [Java Application] /Library/Ja
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 1 : Bonjour
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 1 : Bonjour
Thread 2 : Au revoir
Thread 2 : Au revoir
Thread 1 : Bonjour
```

Arrêter un thread : la méthode stop()

- La méthode stop() de la classe Thread est notée Deprecated dans la Javadoc : **elle ne doit surtout plus être utilisée**
- Explication : le fonctionnement de la méthode stop peut laisser certains objets utilisés par le thread dans un état incohérent (par exemple, au milieu d'une modification incomplète)
- Il faut alors passer par une variable qui indique au thread de s'arrêter. Le thread utilise cette variable dans la méthode run() pour déterminer s'il doit s'arrêter, seulement à un moment où on est sûr qu'aucune donnée n'est compromise

Arrêter un thread « proprement »

```
public class MyRunnable implements Runnable {  
    private boolean arret = false;  
    public synchronized void arreter() {  
        arret = true;  
    }  
    private synchronized boolean continuer() {  
        return !arret;  
    }  
    @Override  
    public void run() {  
        while(continuer()) {  
            // traitement normal du thread...  
        }  
    }  
}
```

- On revient sur le mot-clé **synchronized** dans la suite

Arrêter un thread « proprement »

```
public class MyRunnableMain {  
    public static void main(String[] args) {  
        MyRunnable myRunnable = new MyRunnable();  
        Thread thread = new Thread(myRunnable);  
  
        thread.start();  
        // Instructions ...  
  
        myRunnable.arreter();  
    }  
}
```

Ordonnancement et Synchronisation

Influencer l'ordre des actions

```
Thread t = new MonThread();  
t.start();  
// Autres instructions  
for (int i = 0; i < 1000; i++)  
    System.out.println("Bonjour");
```

En théorie, l'exécution de ce code va alterner les intructions du thread et les autres instructions.

Influencer l'ordre des actions

[illegible]

Influencer l'ordre des actions



```
<terminated> TestJoin [Java Application] /Library/Java/JavaVirt  
Dans mon thread  
Dans mon thread  
Dans mon thread  
Dans mon thread  
Dans mon thread  
Dans mon thread  
Dans mon thread  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour  
Bonjour
```

Influencer l'ordre des actions

[illegible]

Influencer l'ordre des actions : la méthode join

La méthode `join` de la classe `Thread` permet d'attendre la fin de l'exécution du thread avant d'exécuter les instructions suivantes.

Attention : Cette méthode est susceptible de lever une exception.

```
Thread t = new MonThread();
t.start();
try {
    t.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
// Autres instructions
for (int i = 0; i < 1000; i++)
    System.out.println("Bonjour");
```

Cette fois-ci, le programme affiche 1000 fois "Dans mon thread" d'abord, puis 1000 fois "Bonjour".

Influencer l'ordre des actions : la méthode join

La méthode `join` peut aussi prendre un paramètre de type **long** :
public void `join(long millis)` attend au plus `millis` millisecondes la fin de l'exécution du thread.

```
Thread t = new MonThread();  
t.start();  
try {  
    t.join(1000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
// Autres instructions...
```

Si l'exécution du thread `t` prend plus qu'une seconde, alors les autres instructions pourront s'exécuter.

Section critique

- Des problèmes peuvent apparaître lorsqu'une même zone mémoire est utilisée par plusieurs threads
- Exemple : modification d'un compteur par plusieurs threads

```
public class Compteur{  
    private int valeur ;  
  
    public Compteur(){ valeur = 0;}  
  
    public void incrementer(int n){  
        valeur += n ;  
    }  
  
    public int getValeur(){return valeur;}  
}
```

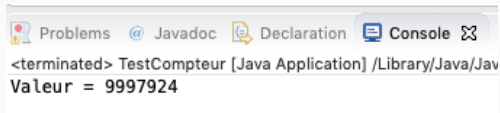
Section critique

```
public class TestCompteur {  
    public static void main(String[] args) {  
        Compteur compteur = new Compteur();  
        for (int i = 0; i < 10000; i++) {  
            Thread t = new Thread() {  
                public void run() {  
                    for (int i = 0; i < 1000; i++)  
                        compteur.incrementer(1);  
                }  
            };  
            t.start();  
        }  
        System.out.println("Valeur_=" +  
                           + compteur.getValeur());  
    }  
}
```

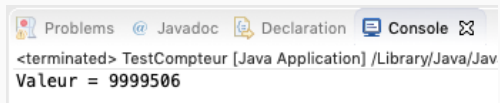
- Quel affichage obtient-on ?

Section critique

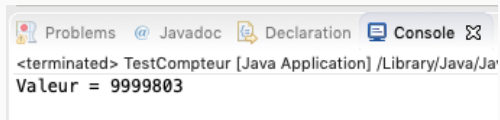
- Trois exécutions différentes :



```
<terminated> TestCompteur [Java Application] /Library/Java/Jav  
Valeur = 9997924
```



```
<terminated> TestCompteur [Java Application] /Library/Java/Jav  
Valeur = 9999506
```



```
<terminated> TestCompteur [Java Application] /Library/Java/Jav  
Valeur = 9999803
```

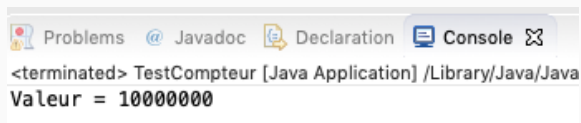
- Lorsqu'un thread incrémente le compteur, plusieurs actions se déroulent :
 1. Accès à la valeur du compteur
 2. Calcul de *valeur* + 1
 3. Affectation de *valeur* + 1 au compteur
- Si deux threads t1 et t2 accèdent au compteur en même temps, ils vont tous les deux récupérer la même valeur (par exemple 0), et donc après l'exécution des deux threads, le compteur vaut 1 au lieu de 2

- Il faut forcer les threads à considérer chaque section critique comme une instruction atomique
- Lorsqu'un thread accède à la zone critique, il faut empêcher les autres threads d'y accéder :
 - Pose d'un verrou sur la section critique lorsque le thread y accède
 - Libération de la zone lorsque le thread quitte la section critique
- Mot-clé **synchronized**

Synchronisation du compteur

```
public class Compteur {  
    private int valeur;  
  
    public Compteur() {  
        valeur = 0;  
    }  
  
    public synchronized void incrementer(int n) {  
        valeur += n;  
    }  
  
    public int getValeur() {  
        return valeur;  
    }  
}
```

Synchronisation du compteur : exécution



The screenshot shows a horizontal toolbar with icons for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. Below the toolbar, the console output reads: '<terminated> TestCompteur [Java Application] /Library/Java/Java' followed by a new line with 'Valeur = 10000000'.

```
<terminated> TestCompteur [Java Application] /Library/Java/Java  
Valeur = 10000000
```

Synchronisation sur un bloc d'instructions

- Il est possible de placer le verrou sur un bloc d'instructions à l'intérieur d'une méthode

```
private Object verrou ;
```

```
public void maMethode(){  
    // Instructions ...  
    synchronized(verrou){  
        // Instructions avec verrou  
    }  
    // Encore des instructions ...  
}
```

```
public synchronized void maMethode(){  
    // Methode avec verrou  
}
```

est équivalent à :

```
public void maMethode(){  
    synchronized(this){  
        // Instructions avec verrou  
    }  
}
```

wait(), notify() et notifyAll()

Trois méthodes de la classe Object :

- **public void** wait () {...} place le thread en cours en attente
- **public void** notify () {...} réveille un thread en attente sur l'objet courant
- **public void** notifyAll () {...} réveille tous les threads en attente sur l'objet courant

Principe :

- On dispose d'une file dans laquelle certains threads écrivent des ressources (les producteurs), et d'autres threads lisent des ressources (les consommateurs)
- Exemple d'application concrète : file d'attente d'une imprimante
- Lorsque la file est vide, les consommateurs doivent attendre qu'un producteur y écrive à nouveau
- Lorsqu'un producteur écrit dans la file, il signale aux (éventuels) consommateurs endormis qu'ils peuvent se réveiller

Classe Producteur

```
public class Producteur extends Thread {  
    private LinkedList<Integer> file;  
    private int val;  
  
    public Producteur(String name, int val,  
                      LinkedList<Integer> file) {  
        super(name);  
        this.val = val;  
        this.file = file;  
    }  
  
    public void produire() {  
        synchronized (file) {  
            file.add(val);  
            file.notifyAll();  
        }  
    }  
}
```



```
@Override
public void run() {
    while (true) {
        try {
            sleep(1000);
            produire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Classe Consommateur

```
public class Consommateur extends Thread {  
    private LinkedList<Integer> file;  
    public Consommateur(String name,  
        LinkedList<Integer> file) {  
        super(name);  
        this.file = file;  
    }  
    @Override  
    public void run() {  
        while (true) {  
            try {  
                sleep(1000); consommer();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
public void consommer() {  
    synchronized (file) {  
        while (file.isEmpty())  
            try {  
                file.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        System.out.println(Thread.currentThread()  
            .getName() + " : " + file.removeFirst());  
    }  
}
```

Méthode main

```
public static void main(String[] args) {  
    LinkedList<Integer> file = new LinkedList<Integer>();  
  
    for (int i = 0; i < 5; i++) {  
        Producteur p = new Producteur("Prod_" + i, i, file);  
        p.start();  
    }  
  
    for (int i = 0; i < 10; i++) {  
        Consommateur c = new Consommateur("Conso_" + i,  
                                           file);  
        c.start();  
    }  
}
```

Threads et JavaFX

- Certaines opérations doivent être exécutées en tâches de fond, via un autre thread, pour ne pas bloquer le fonctionnement de l'interface graphique
- On indique à JavaFX de modifier la GUI via `Platform.runLater(runnable)`

Exemple : incrémentation automatique d'un compteur

```
public class CompteurApp extends Application {  
    private int compteur = 0;  
    private Label label = new Label("" + compteur);  
  
    private void incrementeCompteur() {  
        compteur++;  
        label.setText("" + compteur);  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

- Il reste à définir la méthode `start` de l'application

Exemple : incrémentation automatique d'un compteur

```
@Override
```

```
public void start(Stage stage) {  
    BorderPane pane = new BorderPane();  
    pane.setCenter(label);
```

```
    Scene scene = new Scene(pane, 200, 200);
```

```
    // Initialisation du Thread qui  
    // incremente le compteur  
    // sur le slide suivant
```

```
    stage.setScene(scene);  
    stage.show();
```

```
}
```

```
}
```


Exemple : incrémentation automatique d'un compteur

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        Runnable updater = new Runnable() {
            @Override
            public void run() {
                incrementeCompteur ();
            } }; // Fin new Runnable
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
            Platform.runLater(updater);
        }
    } }); // Fin new Thread()
thread.setDaemon(true);
thread.start();
```

- docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/concurrent/package-summary.html
- Fournit des classes utiles pour la programmation concurrente