ENSAH

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

Les Threads

Langage Java

2ème année Génie Informatique

Ecole Nationale des Sciences Appliquées – Al Hoceima

Prof A. Bahri
abahri@uae.ac.ma

A.U 2020/2021

Le concept de thread

- Un thread (processus léger ou activité) est :
  - Un fil d'instructions (un chemin d'exécution) à l'intérieur d'un processus (programme).
  - une unité d'exécution au sein d'un même processus (ce n'est pas un autre processus).
- Tous les threads d'un même processus partagent la même zone mémoire.
- Un thread possède un nom, une priorité, etc.
- Un thread s'exécute jusqu'au moment où:
  - Un thread de plus grande priorité devient exécutable.
  - Une méthode `wait ()`, `yield ()` ou `sleep ()` est lancée.
  - Son quota de temps a expiré dans un système préemptif.
- La programmation multithreads donne l'illusion de la simultanéité.
  - De nombreux threads peuvent s'exécuter simultanément dans un programme (exécution concurrente de threads).
  - Les ressources allouées à un processus (temps processeur, mémoire) sont partagées entre les threads qui le composent.

Le concept de processus

- Un processus est une instance en exécution d'un programme
- Un processus peut contenir plusieurs threads
  - Un processus possède au moins un thread (qui exécute le programme principal, habituellement la fonction main()).
- La gestion de l'exécution des processus est assurée par le système d'exploitation (OS)
- Un OS est capable d'exécuter plusieurs processus en même temps: multi-tasking (Linux, Windows 7, …)

Différences entre un thread et un processus

- Au niveau de la mémoire :
  - Les sous-processus issus d'un même processus ont leur propre espace d'adressage et doivent, pour communiquer, utiliser des moyens de communication spécifiques (tubes..)
  - Les threads issus d'un même processus partagent la même zone mémoire (segments de code et de données), ce qui rend très facile (et périlleux !) la communication entre threads.

Intérêts des threads en général

- Communication très simple grâce aux données partagées
- Augmenter la "productivité" d'une application par l'exécution concurrente de ces threads
- Exécuter des traitements en parallèle (gagner du temps)
- Maintenir la réactivité d'une application durant une longue tache d'exécution.
- Donner la possibilité d'annulation de taches spéciales

Threads et Java

- En java, un thread est un objet
- On le crée et on lui assigne des traitements à faire
- La JVM prend en charge la responsabilité d'exécuter le thread
- Un thread a plusieurs états

```
public class Main {

   public static void main(String[] args) {
    System.out.println("Nom du thread: "+Thread.currentThread().getName());
    System.out.println(" Etat du thread: "+Thread.currentThread().getState());
    System.out.println(" Id du thread: "+ Thread.currentThread().getId());
  }
}
```

Résultat d'exécution

```
Nom du thread: main
Etat du thread: RUNNABLE
Id du thread: 1
```

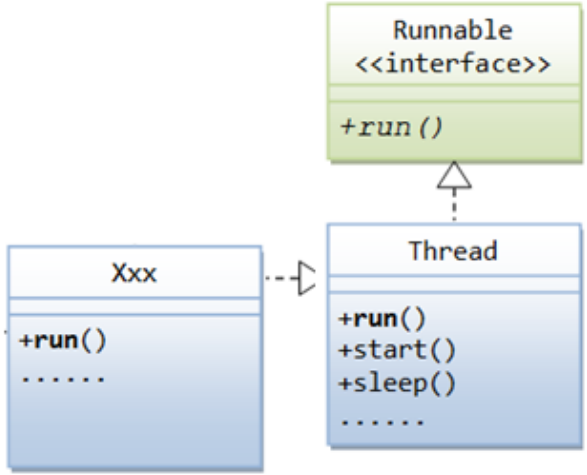
Comment créer des Threads en Java?

- Un thread java est un objet qui control l'exécution d'un programme. C'est une instance de la classe Thread

- Deux façons pour créer un thread java :

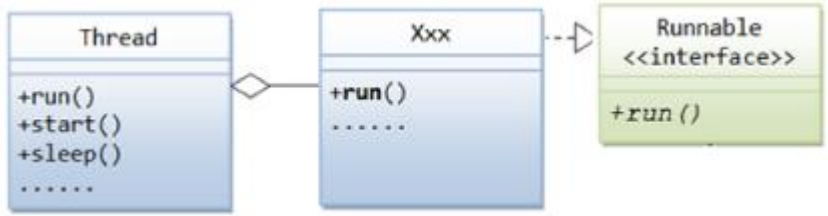
Etendre la classe Thread

```
class Xxx extends Thread{
            …..
}
```



Implanter l'interface Runnable

```
class Xxx implements Runnable{
            …..
}
```



Comment créer des Threads en Java?

Créer Thread avec Thread ou Runnable?

- Créer un Thread avec Runnable.
  - Si Thread doit être une sous-classe d'une autre classe (une classe ne peut pas avoir 2 supe-classes: pas d'héritage multiple en java)

- Créer un Thread avec Runnable.
  - pour cacher les variables et méthodes de la classe Thread

Comment créer des Threads en Java?

Etendre la classe Thread

```
public class MyThread extends Thread {
    public MyThread(String nom) {
        super(nom);
    }
    public void run() {// corps du thread
        for (int i=0; i<3; i++)
          System.out.println(" je suis le thread : "+getName());
    }
    public static void main(String args[ ]) {
        new MyThread("Premier thread").start();
        new MyThread("Deuxieme thread").start();
    }
}
```

Résultat d'exécution

```
je suis le thread : Deuxieme thread
je suis le thread : Premier thread
je suis le thread : Deuxieme thread
je suis le thread : Deuxieme thread
je suis le thread : Premier thread
je suis le thread : Premier thread
```

Comment créer des Threads en Java?

Implanter l'interface Runnable

```
public class MyExecutable implements Runnable {
    String nom;
    public MyExecutable(String nom) {
        this.nom = nom;
    }
    @Override
    public void run() {// corps du thread
        for (int i=0; i<3; i++)
            System.out.println(" je suis l'executable: "+ nom);
    }
    public static void main(String args[ ]) {
        new Thread(new MyExecutable("Premier executable")).start();
        new Thread(new MyExecutable("Deuxieme executable")).start();
    }
}
```

Résultat d'exécution

```
je suis l'executable: Deuxieme executable
je suis l'executable: Premier executable
je suis l'executable: Premier executable
je suis l'executable: Premier executable
je suis l'executable: Deuxieme executable
je suis l'executable: Deuxieme executable
```

Méthodes start() et run()

La différence principale entre ces deux méthode est que:

- lorsque le programme appelle la méthode start(), un nouveau thread est créé et lui associé une pile d'exécution et le code à l'intérieur de la méthode run () est exécuté dans cette pile,

- alors que si on appel la méthode run () directement aucun nouveau thread n'est créé et le code à l'intérieur de run() s'exécutera par le Thread en cours.

Différences avec un appel à run()

```
public class MyThread extends Thread {
    public MyThread(String nom) {
        super(nom);
    }
    public void run() {// corps du thread
        for (int i=0; i<3; i++)
          System.out.println(" je suis le thread : "+getName());
    }
    public static void main(String args[ ]) {
        new MyThread("Premier thread").run(); //start();
        new MyThread("Deuxieme thread").run(); //start();
    }
}
```

Résultat d'exécution

```
je suis le thread : Premier thread
je suis le thread : Premier thread
je suis le thread : Premier thread
je suis le thread : Deuxieme thread
je suis le thread : Deuxieme thread
je suis le thread : Deuxieme thread
```

Exécution séquentiel !!
Voir slide 9 pour comparer

Quelques méthodes de la classe: java.lang.Thread

Méthode	Description
start ()	Rend un thread exécutable en lançant la méthode run ().
sleep (i)	Endort le thread pour i millisecondes.
wait ()*	Suspend le thread.
notify ()*	Place le thread dans un état exécutable.
notifyAll ()*	Réveille tous les threads en attente.
yield ()	Place le thread de l'état « en cours d'exécution » à l'état « exécutable ».
setPriority (i)	Modifie la priorité d'un thread (i est compris entre MIN_PRIORITY et MAX_PRIORITY).
join() ***join (long)***	Pour qu'un deuxième thread attende la fin d'exécution d'un premier thread, il suffit d'appeler la méthode join sur le premier thread. Un paramètre de temps (en millisecondes) peut être spécifié.

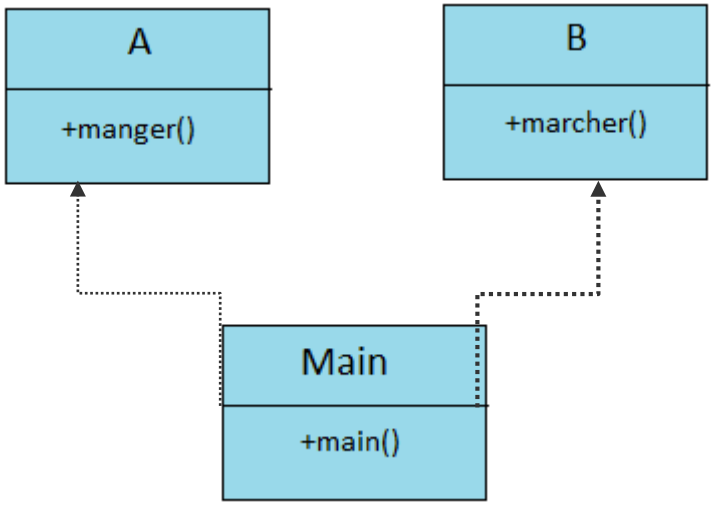
* Méthodes héritées de la classe *java.lang.Object*

Simulation de comportement (séquentielle): sans Thread

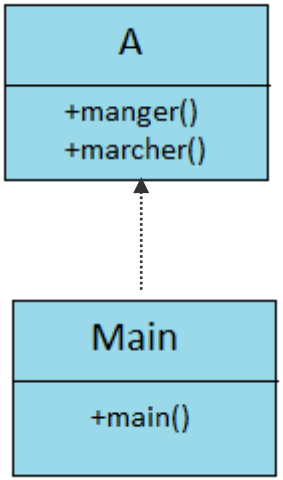
Simuler le comportement d'une personne

- manger() (1 secondes (minutes))
- marcher() (2 secondes (minutes))

- Les deux traitements se font de manière séquentielle (3 secondes(minutes) pour manger et marcher



Modèle 1



Modèle 2

Simulation de comportement (séquentielle): sans Thread

```
class A {
   void manger() {
         for(int i=1;i<=10;i++)
            System.out.println(" Miette"+i);
   }
}

class B {
    void marcher() {
         for(int i=1;i<=10;i++)
              System.out.println(" Pas "+i);
   }
}

public class Main{

    public static void main(String[] args) {
         A a=new A();
         B b=new B();
         a.manger();
         b.marcher();
    }
}
```

Résultat d'exécution

```
Miette 1
Miette 2
Miette 3
Miette 4
Miette 5
Miette 6
Miette 7
Miette 8
Miette 9
Miette 10
Pas 1
Pas 2
Pas 3
Pas 4
Pas 5
Pas 6
Pas 7
Pas 8
Pas 9
Pas 10
```

Simulation de comportement (parallèle??): avecThread

```
class A extends Thread{
   void manger() {
           for(int i=0;i<10;i++)
              System.out.println(" Miette "+i);
   }
   public void run() {
           manger();
   }
}

class B extends Thread{
    void marcher() {
           for(int i=0;i<10;i++)
                System.out.println(" Pas "+i);
   }
   public void run() {
           marcher();
   }
}

public class Main{

    public static void main(String[] args) {
           A a=new A();
           B b=new B();
           a.start();
           b.start();
    }
}
```

Résultat d'exécution

```
Miette 0
Miette 1
Miette 2
Miette 3
Miette 4
Pas 0
Miette 5
Pas 1
Miette 6
Pas 2
Miette 7
Pas 3
Pas 4
Miette 8
Pas 5
Miette 9
Pas 6
Pas 7
Pas 8
Pas 9
```

Simulation de comportement (parallèle): avecThread

```
class A extends Thread{
   void manger() {
           for(int i=0;i<10;i++) {
             System.out.println("Miette "+i);
             try { Thread.sleep(1000); }catch(Exception exp) {}
           }
   }
   public void run() {
           manger();
   }
}
class B extends Thread{
    void marcher() {
           for(int i=0;i<10;i++){
              System.out.println("Pas "+i);
              try { Thread.sleep(1000); }catch(Exception exp) {}
           }

   }
   public void run() { marcher();  }
}
public class Main{
    public static void main(String[] args) {
           A a=new A();
           B b=new B();
           a.start();
           b.start();
    }
}
```

Résultat d'exécution

```
Miette 0
Pas 0
Pas 1
Miette 1
Pas 2
Miette 2
Pas 3
Miette 3
Pas 4
Miette 4
Miette 5
Pas 5
Miette 6
Pas 6
Miette 7
Pas 7
Miette 8
Pas 8
Miette 9
Pas 9
```

Thread et JVM

Deux résultats d'exécution différents

- Le thread a besoin de JVM pour s'exécuter
- À une unité de temps donnée, un seul thread qui s'exécute
- Quant il s'agit de plusieurs threads qui s'exécutent en parallèle, le JVM décide quel thread s'exécuter pour une unité de temps (Threads **scheduler algorithm**)

```
Pas 0
Miette 0
Pas 1
Miette 1
Miette 2
Pas 2
Miette 3
Pas 3
Miette 4
Pas 4
Miette 5
Pas 5
Miette 6
Pas 6
Miette 7
Pas 7
Miette 8
Pas 8
Pas 9
Miette 9
```

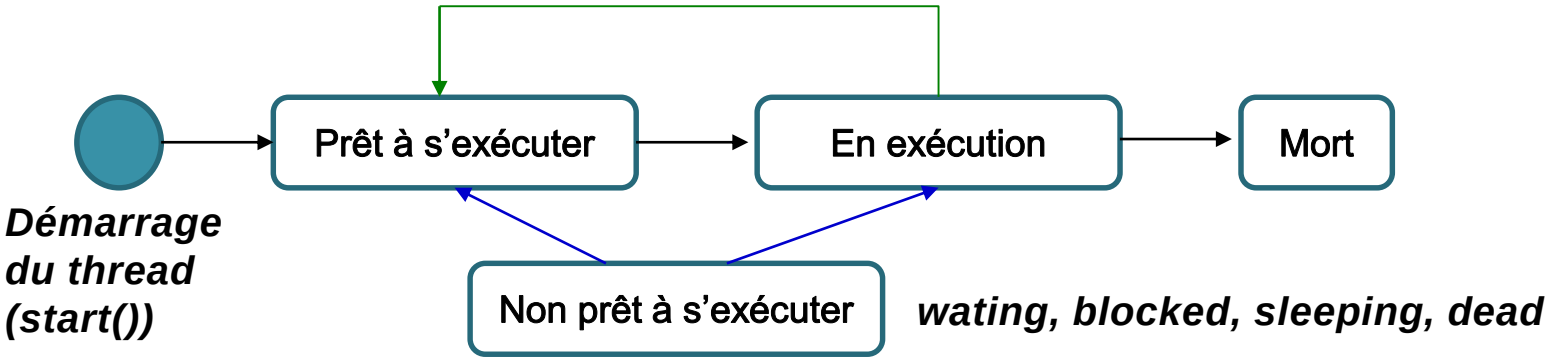
```
Miette 0
Pas 0
Pas 1
Miette 1
Pas 2
Miette 2
Pas 3
Miette 3
Pas 4
Miette 4
Miette 5
Pas 5
Miette 6
Pas 6
Miette 7
Pas 7
Miette 8
Pas 8
Miette 9
Pas 9
```

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

Thread et JVM

- 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

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 thtread 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 cette état jusqu'à ce que le scheduler le rend au pool*

Cycle de vie et les états d'un Thread

```
class A extends Thread{
   void manger() {
           for(int i=0;i<10;i++) {
             System.out.println("Miette "+i);
             try { Thread.sleep(1000); }catch(Exception exp) {}
          }
   }
   public void run() {
           manger();
   }
}
class B extends Thread{
    void marcher() {
           for(int i=0;i<10;i++){
              System.out.println("Pas "+i);
              try { Thread.sleep(1000); }catch(Exception exp) {}
          }

   }
   public void run() { marcher();  }
}
public class Main{
    public static void main(String[] args) {
           System.out.println("je me suis réveillé plus tard!!");
           A a=new A();
           B b=new B();
           a.start();
           b.start();
           System.out.println("Entrer à l'école");
           System.out.println("Commencer le travail");
   }
}
```

Résultat d'exécution

```
je me suis réveillé plus tard!!
Entrer à l'école
Commencer le travail
Miette 0
Pas 0
Pas 1
Miette 1
Miette 2
Pas 2
Pas 3
Miette 3
Miette 4
Pas 4
Pas 5
Miette 5
Pas 6
Miette 6
Pas 7
Miette 7
Pas 8
Miette 8
Pas 9
Miette 9
```

Il faut réglé ce problème !!!!

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

Utilisation de la méthode join() d'un Thread

- join()
  - Utilisé si on souhaite que le thread soit exécuté après l'exécution d'un autre thread
  - Utile quand l'exécution d'un thread dépend de l'exécution d'un autre thread
  - L'instruction t.joint() bloque le thread COURANT jusqu'à la fin de l'exécution du thread contrôlé par t
  - L'instruction t1.joint() est exécutée par un autre thread t2. Cela signifie que t2 doit attendre la fin de t1 et reprend son exécution.

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

Utilisation de la méthode join() d'un Thread

```
public class Main{
    public static void main(String[] args) {
            System.out.println("je me suis réveillé plus tard!!");
            A a=new A();
            B b=new B();
            a.start();
            b.start();
            try {
               a.join();
               b.join();
            }catch(Exception e) {}
            System.out.println("Entrer à l'école");
            System.out.println("Commencer le travail");
    }
}
```

Résultat d'exécution

```
je me suis réveillé plus tard!!
Pas 0
Miette 0
Miette 1
Pas 1
Pas 2
Miette 2
Pas 3
Miette 3
Miette 4
Pas 4
Miette 5
Pas 5
Miette 6
Pas 6
Miette 7
Pas 7
Miette 8
Pas 8
Miette 9
Pas 9
Entrer à l'école
Commencer le travail
```

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

Fixer l'ordre de priorité d'un Thread

```
public class Main{
    public static void main(String[] args) {
           System.out.println("je me suis réveillé plus tard!!");
           A a=new A();
           B b=new B();

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

           a.start();
           b.start();
           try {
              a.join();
              b.join();
           }catch(Exception e) {}
           System.out.println("Entrer à l'école");
           System.out.println("Commencer le travail");
    }
}
```

Enlever la méthode « sleep() »!!!

Résultat d'exécution

```
je me suis réveillé plus tard!!
Miette 0
Pas 0
Miette 1
Miette 2
Pas 1
Miette 3
Miette 4
Miette 5
Pas 2
Miette 6
Miette 7
Pas 3
Miette 8
Miette 9
Pas 4
Pas 5
Pas 6
Pas 7
Pas 8
Pas 9
Entrer à l'école
Commencer le travail
```

Parfois, deux threads peuvent accéder à une donnée en même temps: ⇒ problème!!!!

Problème de ressource partagée

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

Exemple de problème

Initialisation: x=2, création de deux threads T1 et T2
Objectif: incrémenter la valeur de x par deux threads T1 et T2
1.T1 : lit la valeur de x (=2)
2. T2 : lit la valeur de x (=2)
3. T1 : calcule x + 1 (x=3)
4. T2 : calcule x + 1 (x=3)
5. T1 : range la valeur calculée dans x (=3)
6. T2 : range la valeur calculée dans x (=3)

x contient 3 au lieu de 4 !

Synchronisation entre Threads

- En programmation parallèle, on appelle section critique, une partie du code qui ne peut être exécutée en même temps par plusieurs threads sans risquer de provoquer des anomalies de fonctionnement

- Il faut donc éviter l'exécution simultanée de sections critiques par plusieurs threads

- En Java le mot clé synchronized est utilisé pour synchroniser les threads et les empêcher d'exécuter en même temps des portions de code

- Plusieurs threads ne peuvent exécuter en même temps du code synchronisé sur un même objet

- La synchronisation est un mécanisme qui coordonne l'accès à des données commune et à des code critique par des threads

Synchronisation entre Threads

- 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,

Synchronisation entre Threads

- Seulement la méthode entière ou l'une de ces parties qui peuvent être synchronisée

- On ne peut pas synchroniser une classe ou des attributs

Synchronisation entre Threads

Comment synchroniser?

- La syntaxe est la suivante :

```
... code non protégé ...
synchronized(objet) {
        ... code protégé ...
}
... code non protégé ...
```

- Le code protégé n'est exécuté que par un seul thread à la fois, tant qu'il n'a pas terminé le bloc d'instruction.

- Durant l'exécution de ce code protégé par un thread, un autre thread ne peut exécuter celui-ci, mais peut exécuter un autre bloc `synchronized` si celui-ci n'utilise pas le même objet et qu'il n'est pas déjà en cours d'exécution

Synchronisation entre Threads

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

```
public synchronized void codeProtege() {
                    ... code protégé ...
}
```

```
public synchronized void incremente() {
    for (int i=0; i<20000;i++) {
          contenu++;
          System.out.println(getName()+"incremente "+i);
}
```

- Parfois, seulement une partie du code qui a besoin d'être protégée. Dans ce cas, la partie du code à protéger est synchronisée avec un

```
public void codeProtege() {
         synchronized(this) {
                    ... code protégé ...
         }
}
```

```
public void run() {
    synchronized (this){
          this.contenu++;
          System.out.println(getName()+"incremente ");
    }
}
```

Synchronisation entre Threads

Résumé

- Tant que t exécute du code synchronisé sur un objet o, les autres threads ne peuvent exécuter du code synchronisé sur ce même objet o (le même code, ou n'importe quel autre code synchronisé sur o) ; ils sont mis en attente

- Lorsque t quitte le code synchronisé ou se met en attente par o.wait(), un des threads en attente peut commencer à exécuter le code synchronisé

- Les autres threads en attente auront la main à tour de rôle (si tout se passe bien...)

Exemple sans synchronisation sur une variable de classe

```
class Compteur extends Thread{
   private static int valeur=0 ;
   void incremente (){
          for(int i=0;i<1000;i++) {
              valeur += 1 ;
              try { Thread.sleep(100); }catch(Exception exp) {}
          }
   }
   static int Combien () {  return valeur ;   }
   public void run() {  incremente();   }
}
public class MainCompteur{
    public static void main(String[] args) {
    Compteur c1=new Compteur();
    Compteur c2=new Compteur();
    c1.setName("-compteur 1-");  c2.setName("-compteur 2-");
    c1.start();  c2.start();
    try {  c1.join(); c2.join(); }catch(Exception e) {}
    System.out.println("Total:"+Compteur.Combien());

   }
}
```

Résultat d'exécution

```
Total:1982
```

- Les objets de classe `Compteur` peuvent accéder simultanément en même temps à la variable `valeur`.

Exemple de synchronisation sur une variable de classe

```
class Compteur extends Thread{
   private static int valeur=0 ;
   void synchronized incremente (){
          for(int i=0;i<1000;i++) {
              valeur += 1 ;
              try { Thread.sleep(100); }catch(Exception exp) {}
          }

   }
   static int Combien () {  return valeur ;   }
   public void run() {  incremente();   }
}
public class MainCompteur{
    public static void main(String[] args) {
    Compteur c1=new Compteur();
    Compteur c2=new Compteur();
    c1.setName("-compteur 1-");  c2.setName("-compteur 2-");
    c1.start();  c2.start();
    try {  c1.join(); c2.join(); }catch(Exception e) {}
    System.out.println("Total:"+Compteur.Combien());

   }
}
```

Résultat d'exécution

```
Total:2000
```

- `public final Class getClass()` : renvoie la classe de l'objet.
- Tous les objets de classe `Compteur` seront bloqués dans la méthode `incremente()`.