

# La programmation concurrente

---

*Jean-Ferdy Susini*

*Maître de Conférences - CNAM*

*Département Informatique*

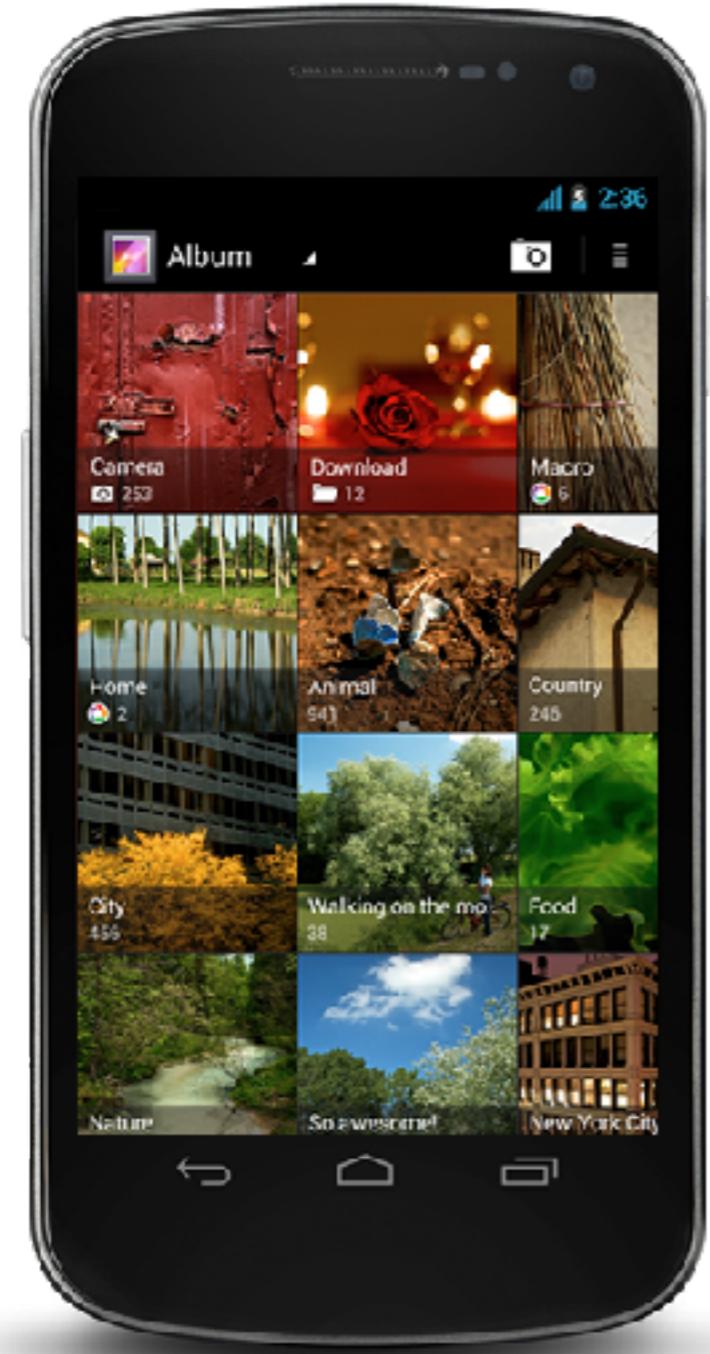
*Sources : Android Developpers, Wikipedia*

le cnam

Paris, 18/01/2018

# Architecture matérielle

2



Considérons l'architecture  
d'un smartphone

# Architecture matérielle

2



Dans un téléphone, l'ensemble des éléments sont généralement intégrés sur le même composant de taille incroyablement petite...

# Architecture matérielle

2

## Ressource de calcul:

- ALU
- registres
- caches
- ..



## Processeur Vidéo dédié



Ecran



## Bus Système North Bridge



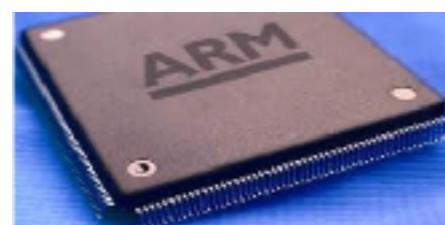
## Mémoire centrale volatile (ou non)

# Architecture matérielle

2

## Ressource de calcul:

- ALU
- registres
- caches
- ..



## Processeur Vidéo dédié



Ecran

## Bus Système North Bridge



## Mémoire centrale volatile (ou non)

Adaptation entre l'interface matérielle du périphérique et l'interface matérielle de la machine

## Contrôleur E/S



Unité de stockage persistant, illustré ici par une carte SD mais il peut s'agir également de mémoire statique directement présente dans le chip du téléphone (c'est d'ailleurs de plus en plus souvent le cas.)

# Multiprogrammation

3

- Objectif du système : diminuer les temps d'attente, optimiser l'utilisation des ressources. Économiser l'énergie et offrir de meilleures performances.
- Moyen : fournir des abstractions permettant au développeur d'identifier et de manipuler des «tâches»
  - notion de processus : une tâche (un programme) en cours d'exécution
  - notion de multiprogrammation : possibilité de créer et de gérer plusieurs tâches concurremment ou parallèlement : allocation des ressources de calcul
  - service de contrôle des tâches et d'allocation des ressources systèmes (mémoire, fichiers, ...)

# Les processus légers : threads

- Un premier constat : les processus sont des entités lourdes : création (duplication), partage d'information compliqué (IPC), changements de contexte lourds...
- Une proposition alternative : les processus légers ou threads
  - **espace mémoire partagé** => changement de contexte simplifié, échange de données facile
  - **structure de contrôle** proche du processus => réutilisation des notions d'ordonnancement
    - ➡ profiter des apports de la multiprogrammation au sein même d'un processus, profiter du parallélisme physique des machines (plusieurs ressources de calcul disponibles)

# Les processus légers : threads

- Processus classique :

Fil d'exécution (contexte processeur : CO, pile/registres, ...)	Ressources (fichiers, signaux, sockets,...)	Espace d'adressage mémoire du processus
---	---	--

- Processus “multi-threadé” :

Fil d'exécution (contexte processeur : CO, pile/registres, ...)	Ressources (fichiers, signaux, sockets,...)	Espace d'adressage mémoire du processus
Fil d'exécution (contexte processeur : CO, pile/registres, ...)		

# Les threads en Java

6

- Intègre la notion de thread issue de la programmation système dans un langage de programmation généraliste
- Chaque thread **exécute un code séquentiel**
- Gestion des ressources par la JVM : partage de l'espace d'adressage (mémoire centrale de la JVM), pile et registres séparés pour chaque thread.
- Correspondances ("Mapping") avec les processus légers du système d'exploitation : un à un, plusieurs à un, plusieurs à plusieurs
- S'appuie sur un service d'ordonnancement (interne à la JVM et/ou fourni par le système d'exploitation)

# Android et le UIThread

7

- Le système Android impose un fonctionnement particulier du fil d'exécution principal («main») dans une application. L'exécution du «main» est sous contrôle de l'OS et non du développeur de l'application.
- Rebaptisée «UIThread», ce fil d'exécution est en charge de traiter la boucle événementielle de l'application, traitant ainsi la totalité des événements et du cycle de vie de tous les composants applicatifs.
- Nécessite que toutes les opérations attachées aux traitements des événements et des cycles de vie des composants soient brèves sous peine de bloquer le «thread principal».

# Impact sur la programmation

8

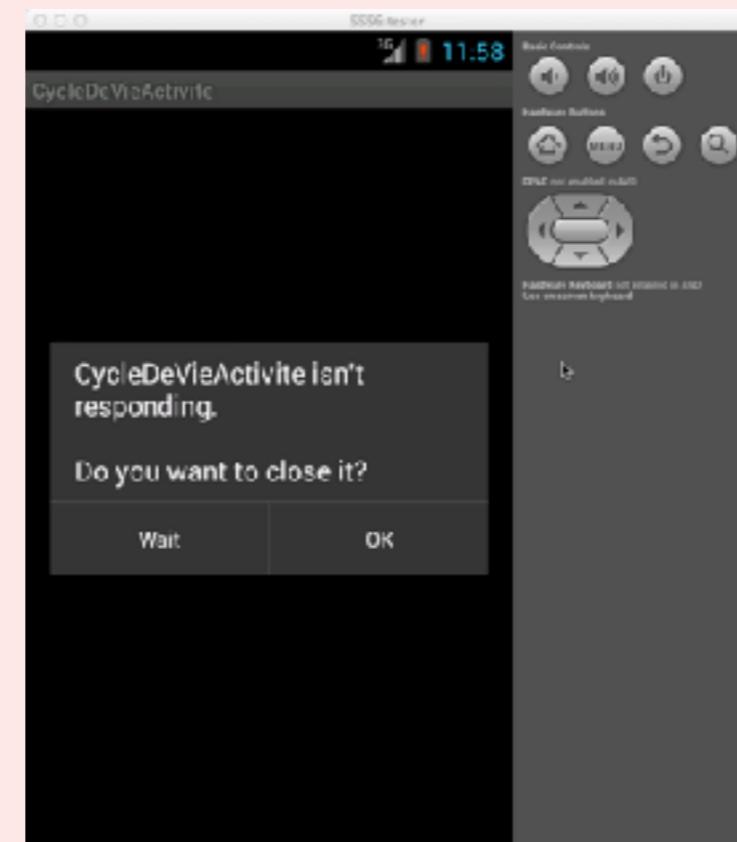
- Un timer supervise le processus d'une application => arrêt forcé en cas de non réponse de l'activité à l'issu de la période de test (5s de non réponse à un événement utilisateur). dialogue ANR
- Toute tâche complexe doit donc être déléguée à un thread spécifique et éventuellement sous la responsabilité d'un autre composant applicatif non graphique afin de préserver la réactivité du système.
- Différentes API permettent de gérer la concurrence sous Android. Ces mécanismes proviennent du langage java lui même ou sont spécifiques à Android ou au système Linux.

# Impact sur la programmation

8

- Un timer supervise le processus d'une application => arrêt forcé en cas de non réponse de l'activité à l'issu de la période de test (5s de non réponse à un événement utilisateur). dialogue ANR
- Toute tâche thread spéciale est responsable de la responsabilité de la réponse à un événement graphique à l'écran.
- Différences entre les deux plateformes Android. Celle-ci gère elle-même ce type d'erreur alors que Linux.

Application Not Responding



réponse à un événement graphique non traité par un thread système.  
occurrence sous programme java au système

# Impact sur la programmation

8

- Un timer supervise le processus d'une application => arrêt forcé en cas de non réponse de l'activité à l'issu de la période de test (5s de non réponse à un événement utilisateur). dialogue ANR
- Toute tâche complexe doit donc être déléguée à un thread spécifique et éventuellement sous la responsabilité d'un autre composant applicatif non graphique afin de préserver la réactivité du système.
- Différentes API permettent de gérer la concurrence sous Android. Ces mécanismes proviennent du langage java lui même ou sont spécifiques à Android ou au système Linux.

# Les threads

9

- Le langage Java est un des premiers langages à populariser la programmation concurrente :
  - Intègre dans le cœur du langage la notion de thread
    - la classe `java.lang.Thread`
    - l'interface `java.lang.Runnable`
  - Au niveau syntaxique : 2 mots clés
    - **synchronized**
    - **volatile**
- La machine virtuelle Java implante directement la notion de fils de contrôle multiples, les classes du package standard `java.lang` établissent le lien entre les programmes exprimés dans le langage et les mécanismes sous-jacent de la VM (threads, verrous...).

# Thread et Runnable

10

- L'interface **Runnable** propose une méthode à exécuter (équivalent au «main») pour chaque fil d'exécution :

```
1 public interface Runnable  
2 {  
3     public void run();  
4 }
```

- Cette méthode ne devrait pas être appelée par le développeur d'application, mais c'est un thread géré par la machine virtuelle qui devrait appeler cette méthode. La méthode s'exécutera alors dans son propre contexte d'exécution mais dans l'espace mémoire partagé de l'application.

# Les Threads

11

- La classe **Thread** représente le lien entre les programmes Java et les mécanismes internes de gestion des fils d'exécution multiple de la VM (JVM ou DalvikVM)
- La classe **Thread** implémente l'interface **Runnable**.
- Les threads Java sont gérés par un mécanisme d'ordonnancement interne à la JVM. Il n'y a pas de mécanisme permettant de modifier la politique d'ordonnancement, mais on peut paramétriser le thread pour influencer son ordonnancement (priorités...).

# Les Threads

12

- La JVM définit une politique d'ordonnancement qui peut varier grandement d'une plate-forme à une autre. Le service d'ordonnancement, dépend souvent de l'OS sous-jacent voir du matériel (CPU...)
- Pour définir un comportement commun aux différentes plate-forme, java donne une sémantique «assez floue» au rôle d'un certain nombre de méthodes : `sleep()`, `yield()`, `setPriority()`, `interrupt()`, `getState()`, `isAlive()`...
- De plus, les méthodes `stop()`, `destroy()`, `suspend()` et `resume()` sont «deprecated» depuis la version Java 1.1 !

# Les Threads

13

- La mémoire partagée pose le problème de la concurrence. L'ordonnancement des Threads étant défini de façon indépendante du code à exécuter, il faut en permanence se poser la question de ce que produit l'accès à une cellule mémoire partagée  
=> **problème de la synchronisation.**
- Java propose des mécanismes de bas niveau de synchronisation : les moniteurs de Hoare
  - notion de verrou porté par chaque objet : défini dans la classe `java.lang.Object`.
  - mot clé **synchronized** permettant de contrôler l'exécution d'un bloc de code Java par l'obtention ou non du verrou.

# Les Threads

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16 }
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

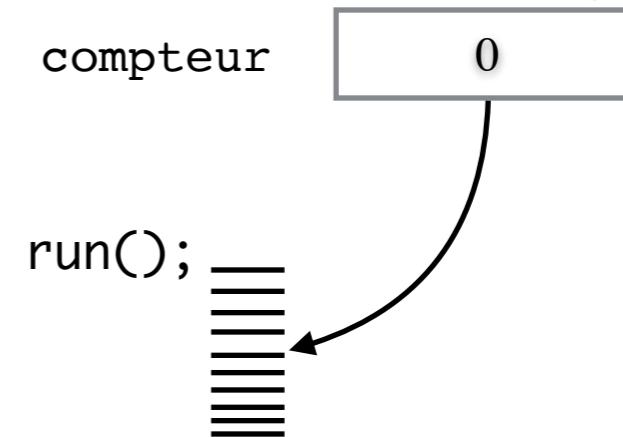
time java ThreadTest  
counter = 1008906  
  
real 0m0.116s  
user 0m0.097s  
sys 0m0.031s

Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16 }
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

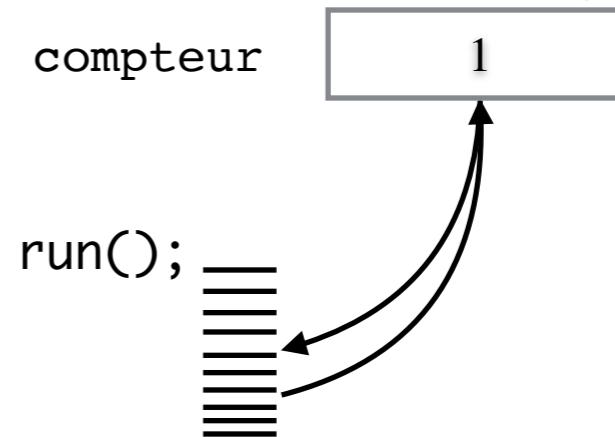


Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16 }
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```



Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

compteur

1

run();



co →

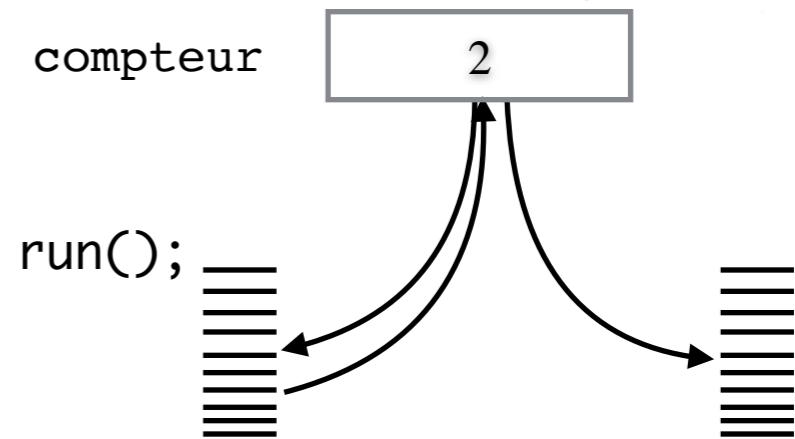
8: getstatic #3  
11: iconst\_1  
12: iadd  
13: putstatic #3

Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}  
17  
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```



co ➔

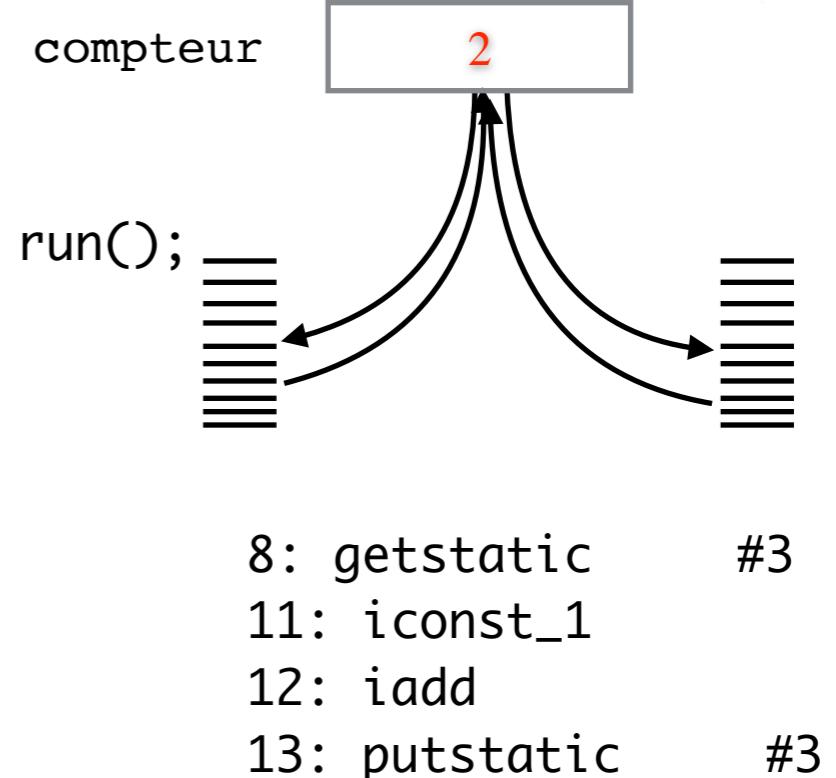
8: getstatic #3  
11: iconst\_1  
12: Changement de contexte  
13: imposé par la politique d'ordonnancement

Illustrons cela à travers un exemple parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

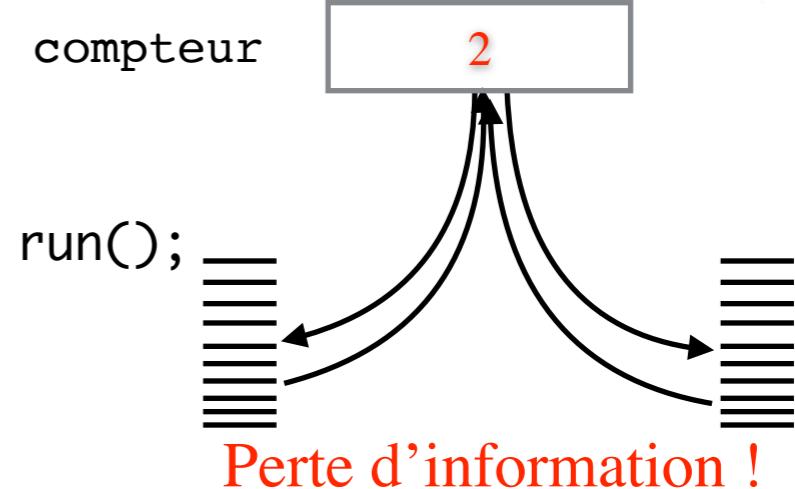


Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```



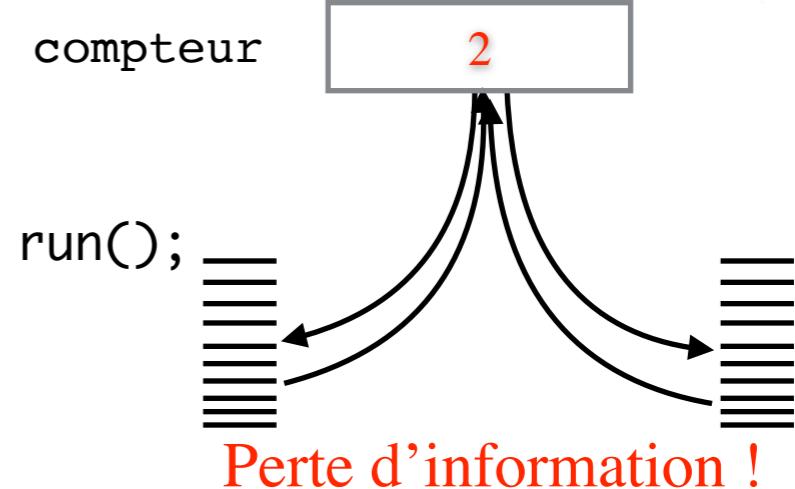
8: getstatic #3  
11: iconst\_1  
12: iadd  
13: putstatic #3

Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

# Les Threads

14

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```



8: getstatic #3  
11: iconst\_1  
12: iadd  
13: putstatic #3

Illustrons cela à travers un scénario  
parmi un très grand nombre de possibles

Ok, il y a là un problème que nous devons essayer de corriger. Pour cela  
nous allons essayer de mettre en œuvre des mécanismes de protection...

# Les Threads

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static synchronized void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16 }
17
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static synchronized void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
```

On considère un scénario similaire

```
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
10    public static synchronized void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
```

compteur

0

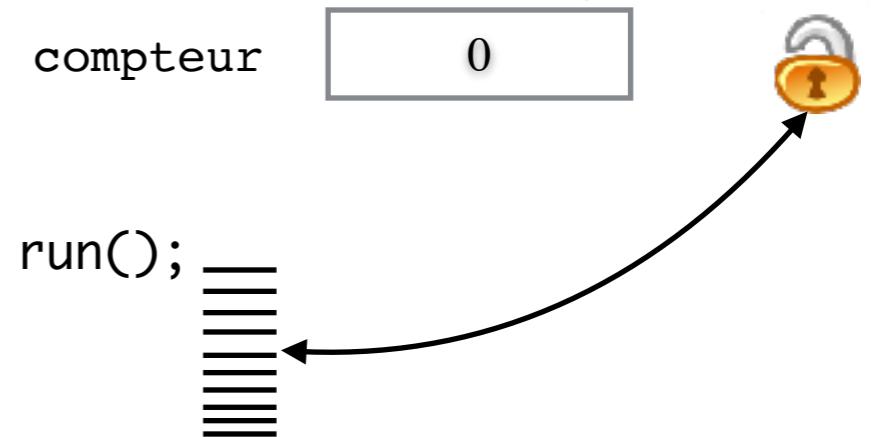
On considère un scénario similaire

```
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



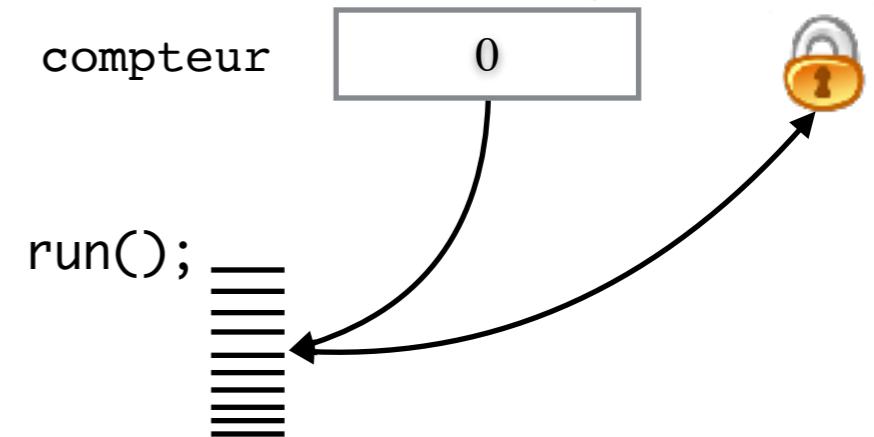
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}  
17  
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

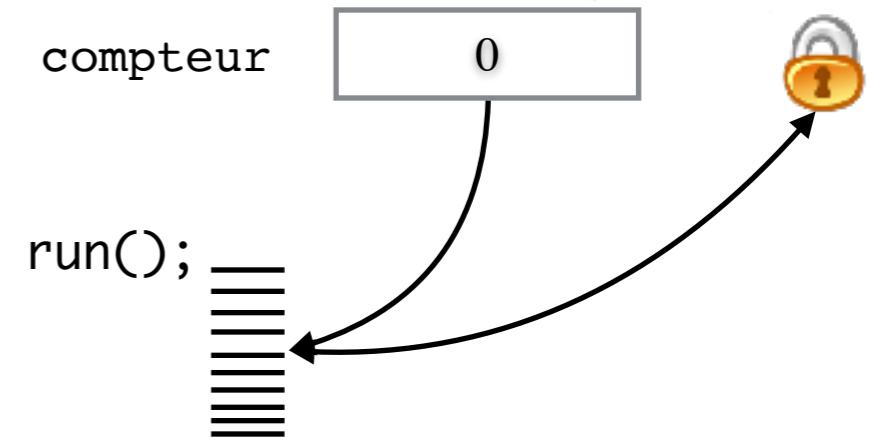


On considère un scénario similaire

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}  
17  
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

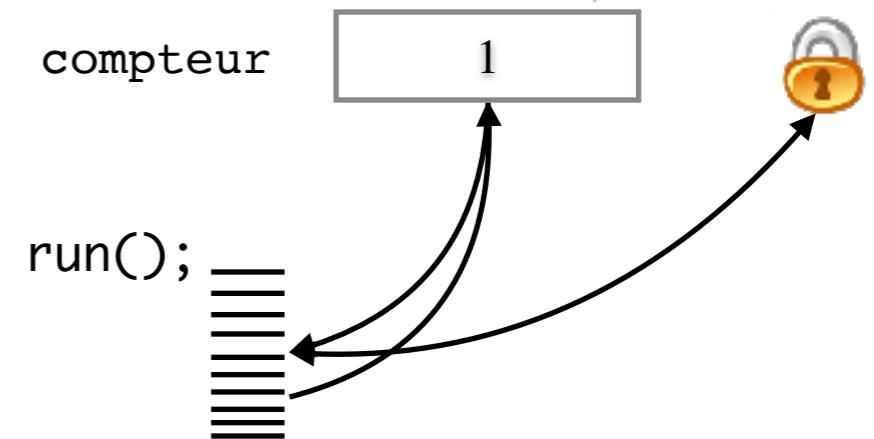


On considère un scénario similaire

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}  
17  
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```



On considère un scénario similaire

# Les Threads

15

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
```

compteur

1



```
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
```

run();



```
10    public static synchronized void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
```

```
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
```

On considère un scénario similaire

```
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

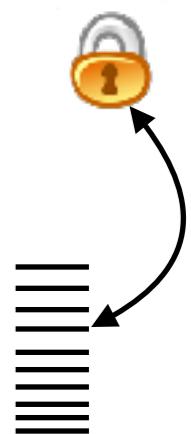
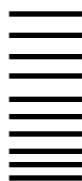
15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```

compteur

1

run();



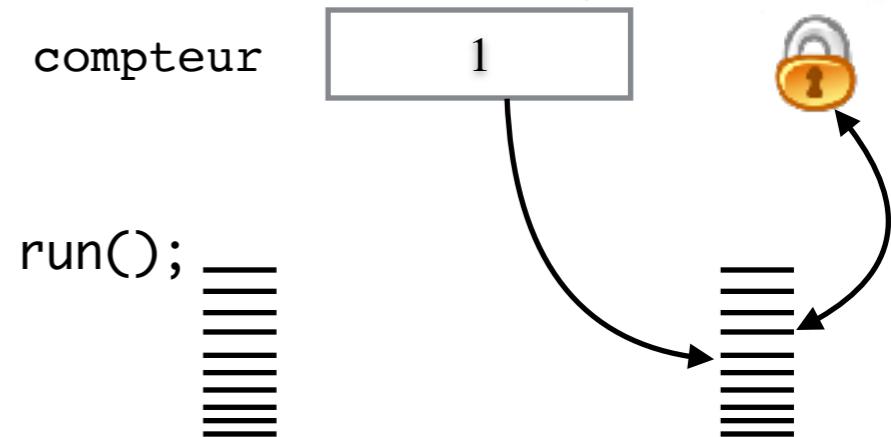
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



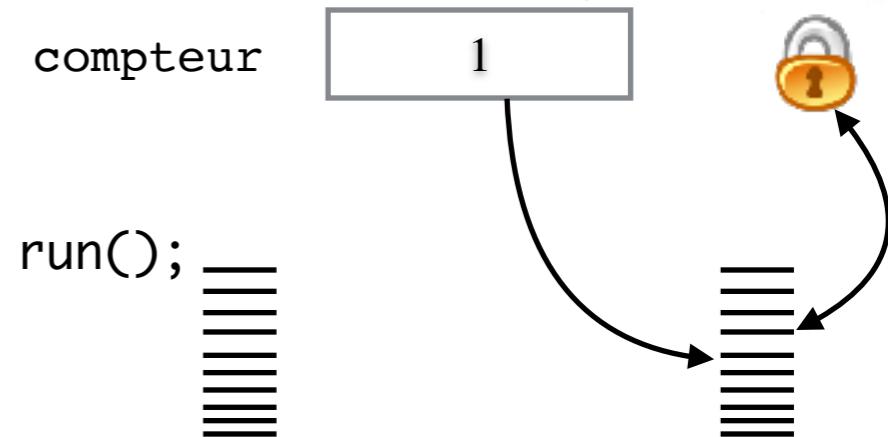
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



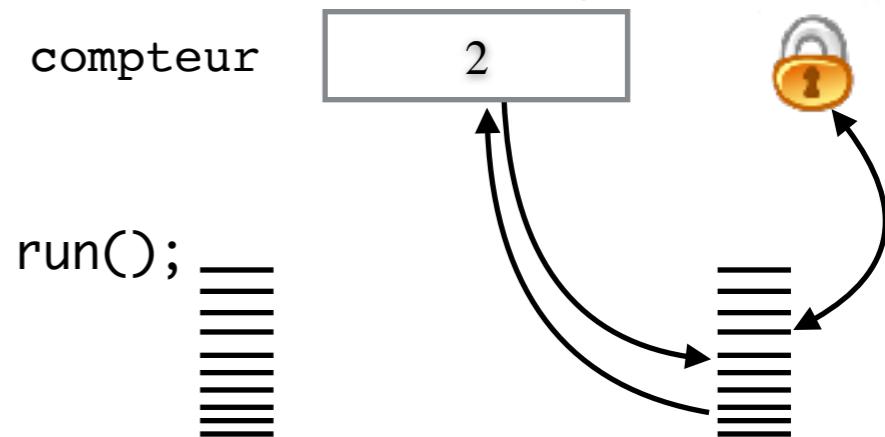
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
```

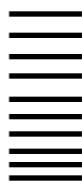
compteur

2



```
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeCompteur();
8         }
9     }
```

run();



```
10    public static synchronized void incrementeCompteur(){
11        compteur = compteur + 1;
12    }
```

```
13    public static void display(){
14        System.out.println("counter = "+compteur);
15    }
16}
```

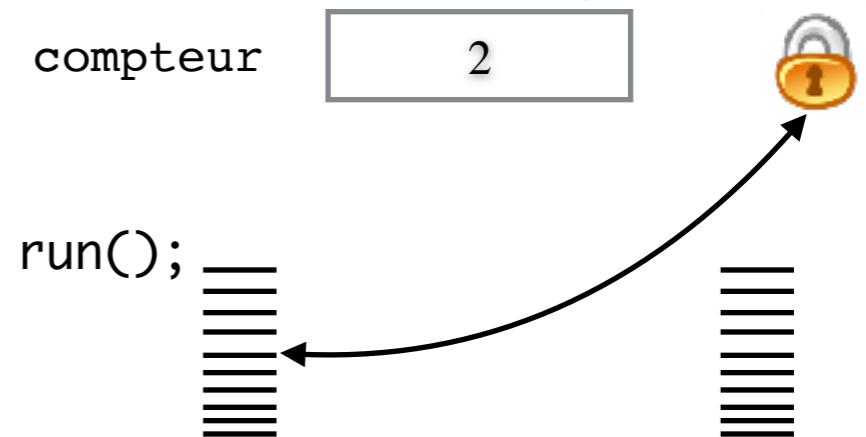
On considère un scénario similaire

```
18 class ThreadTest
19 {
20     public static void main(String[] args) throws InterruptedException{
21         A a1 = new A();
22         A a2 = new A();
23         a1.start();
24         a2.start();
25         a1.join();
26         a2.join();
27         A.display();
28     }
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



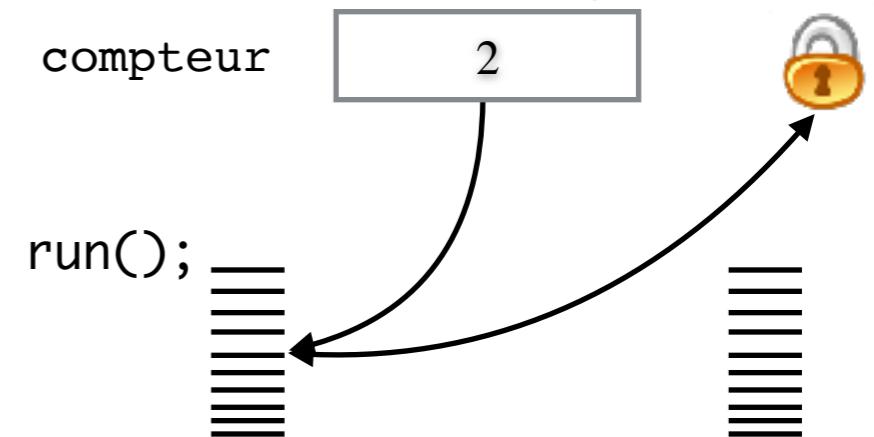
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



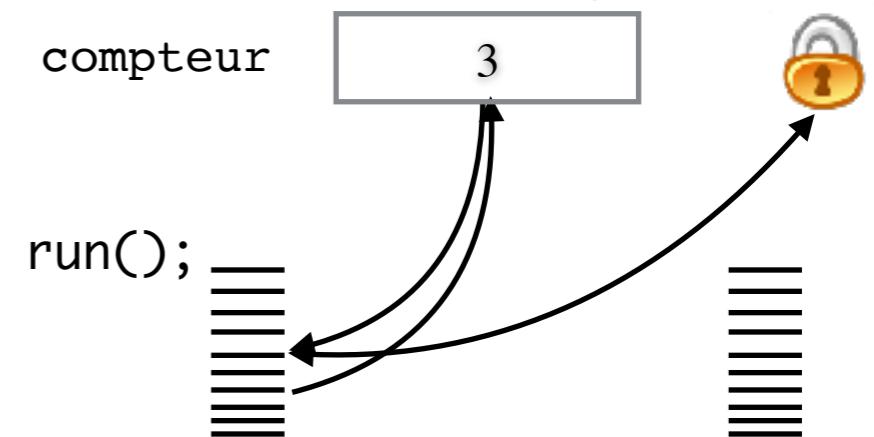
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



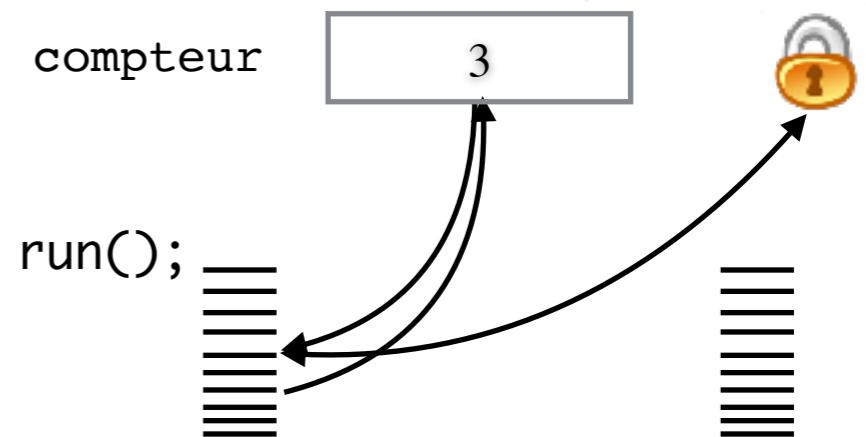
On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



On considère un scénario similaire

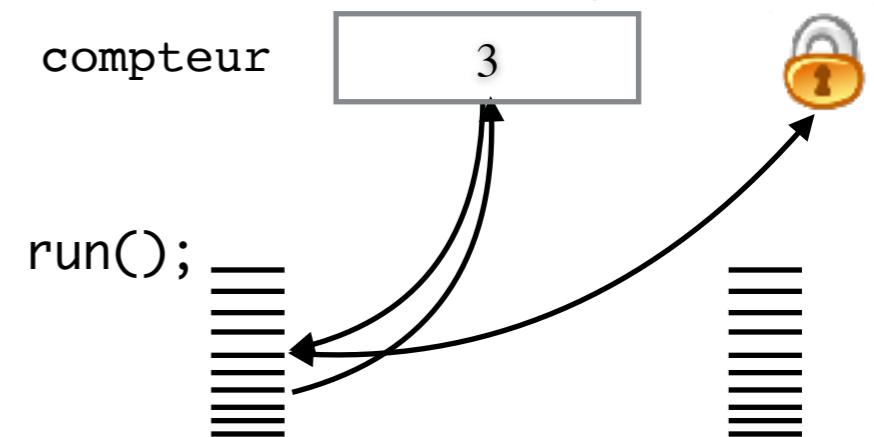
```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

Super ! notre code est cohérent...

# Les Threads

15

```
1 class A extends Thread  
2 {  
3     private static int compteur = 0;  
4  
5     public void run(){  
6         for(int n = 0; n < 1000000; n++){  
7             incrementeCompteur();  
8         }  
9     }  
10    public static synchronized void incrementeCompteur(){  
11        compteur = compteur + 1;  
12    }  
13    public static void display(){  
14        System.out.println("counter = "+compteur);  
15    }  
16}
```



On considère un scénario similaire

```
18 class ThreadTest  
19 {  
20     public static void main(String[] args) throws InterruptedException{  
21         A a1 = new A();  
22         A a2 = new A();  
23         a1.start();  
24         a2.start();  
25         a1.join();  
26         a2.join();  
27         A.display();  
28     }  
29 }
```

Super ! notre code est cohérent...  
Oui mais on a rien gagné... mis à part du code «plus compliqué» et moins performant, surtout sur une machine réellement parallèle : le code concurrent est plus de 2\* plus lent que le code purement séquentiel

# Les Threads

- Les threads conduisent à une programmation d'expert : savoir répartir code et données traitées par le code de façon à limiter les partages de données entre les threads.

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4     private int local_compteur = 0;
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeLocalCompteur();
8         }
9         incrementeCompteur(local_compteur);
10    }
11    public static synchronized void incrementeCompteur(int val){
12        compteur = compteur + val;
13    }
14    public void incrementeLocalCompteur(){
15        local_compteur = local_compteur + 1;
16    }
17    public static void display(){
18        System.out.println("counter = "+compteur);
19    }
20 }
```

# Les Threads

- Les threads conduisent à une programmation d'expert : savoir répartir code et données traitées par le code de façon à limiter les partages de données entre les threads.

```
1 class A extends Thread
2 {
3     private static int compteur = 0;
4     private int local_compteur = 0;
5     public void run(){
6         for(int n = 0; n < 1000000; n++){
7             incrementeLocalCompteur();
8         }
9         incrementeCompteur(local_compteur);
10    }
11    public static synchronized void incrementeCompteur(int val){
12        compteur = compteur + val;
13    }
14    public void incrementeLocalCompteur(){
15        local_compteur = local_compteur + 1;
16    }
17    public static void display(){
18        System.out.println("counter = "+compteur);
19    }
20 }
```

```
time java ThreadTest
counter = 2000000
real 0m0.115s
user 0m0.100s
sys 0m0.029s
```

# Les Threads

17

- Une programmation complexe et risquée :
    - **risque de programme non correcte** en cas d'un défaut de protection (veiller à ne pas mélanger un code utilisant un verrou avec un code qui n'utilise pas ce même verrou -> pas de protection)
    - **risque de deadlocks** la prise de nombreux verrous doit respecter des règles pour éviter les deadlocks (prendre toujours les verrous dans le même ordre au sein d'une séquence... et les relâcher dans l'ordre inverse)
    - **risque sur les performances** (vitesse, énergie...)
- Il s'agit d'une programmation de spécialistes et les bonnes pratiques sont longues à acquérir.

# La concurrence en Java

18

- Les packages `java.util.concurrent.*` définissent un ensemble d'outils classiques sous forme de framework, facilitant la programmation concurrente :
  - barrières de synchronisation
  - «pool» de threads exécutant des tâches (Workers)
  - tests non bloquant d'accès à un verrou
  - accès atomiques (exclusion mutuelle)
  - files, piles et autres structures de données classiques synchronisées
  - Timer et TimerTask...
- Ces outils facilitent la tâche en matière de programmation concurrente et à ces outils de base, Android ajoute ses propres mécanismes...

# Utilisation des Runnable dans Android

19

- Il est possible de faire exécuter des portions de code par un thread via des objets implantant **Runnable**.
- Par exemple, seul le *UIThread* peut gérer l'affichage. Les autres threads peuvent alors soumettre des tâches au *UIThread* pour modifier l'affichage :
  - `runOnUiThread()` d'une activité
  - `post(), postDelayed(), postOnAnimation(), postOnAnimationDelayed(), scheduleDrawable()` des objet `android.view.View`
- Un thread peut donc soumettre du code à exécuter à l'*UIThread* sous forme de **Runnable**.

# Loopers, Messages et Handlers

20

- Le paquetage `android.os` propose différents mécanismes facilitant la programmation concurrente.
- Un thread peut implanter une boucle de traitement de messages (événements ou messages) : le `Looper` et gère une file d'attente de messages la `MessageQueue`. Un `Handler` permet de traiter des messages, tandis que des objets `Message` peuvent-être échangés entre threads.
  - Un message est obtenu (ou construit) par le thread appelant et initialisé avec les données du message.
  - Le message est soumis au `Handler` qui devra le traiter
  - Lorsque le `Looper` peut traiter le message il exécute alors le `handleMessage()` correspondant.

# Loopers, Messages et Handlers

21

Thread

```
public void run(){
```

```
}
```

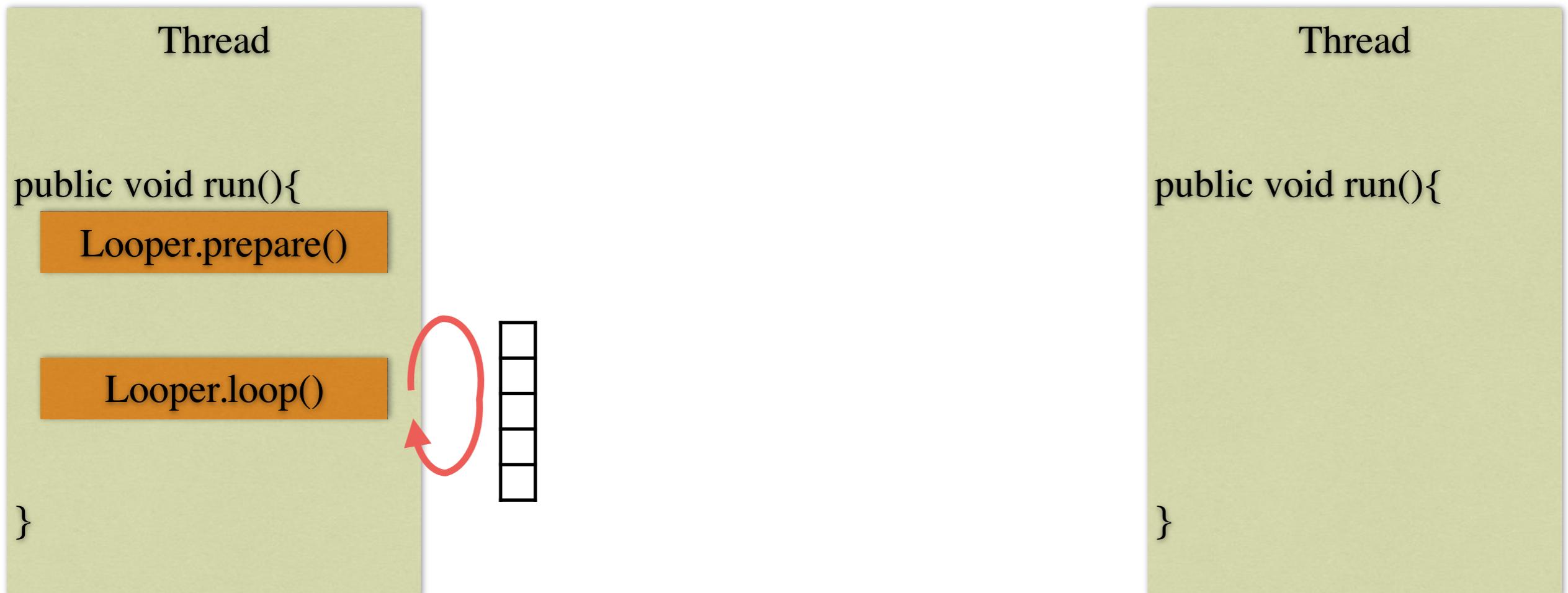
Thread

```
public void run(){
```

```
}
```

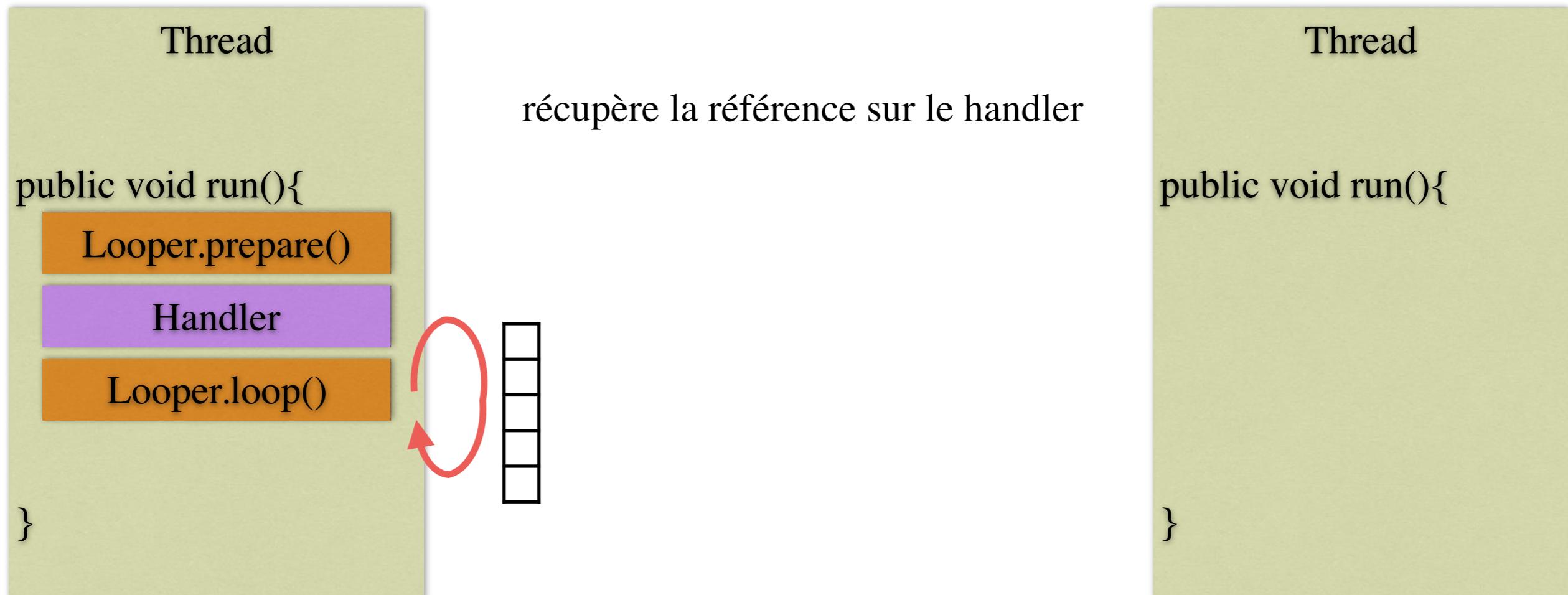
# Loopers, Messages et Handlers

21



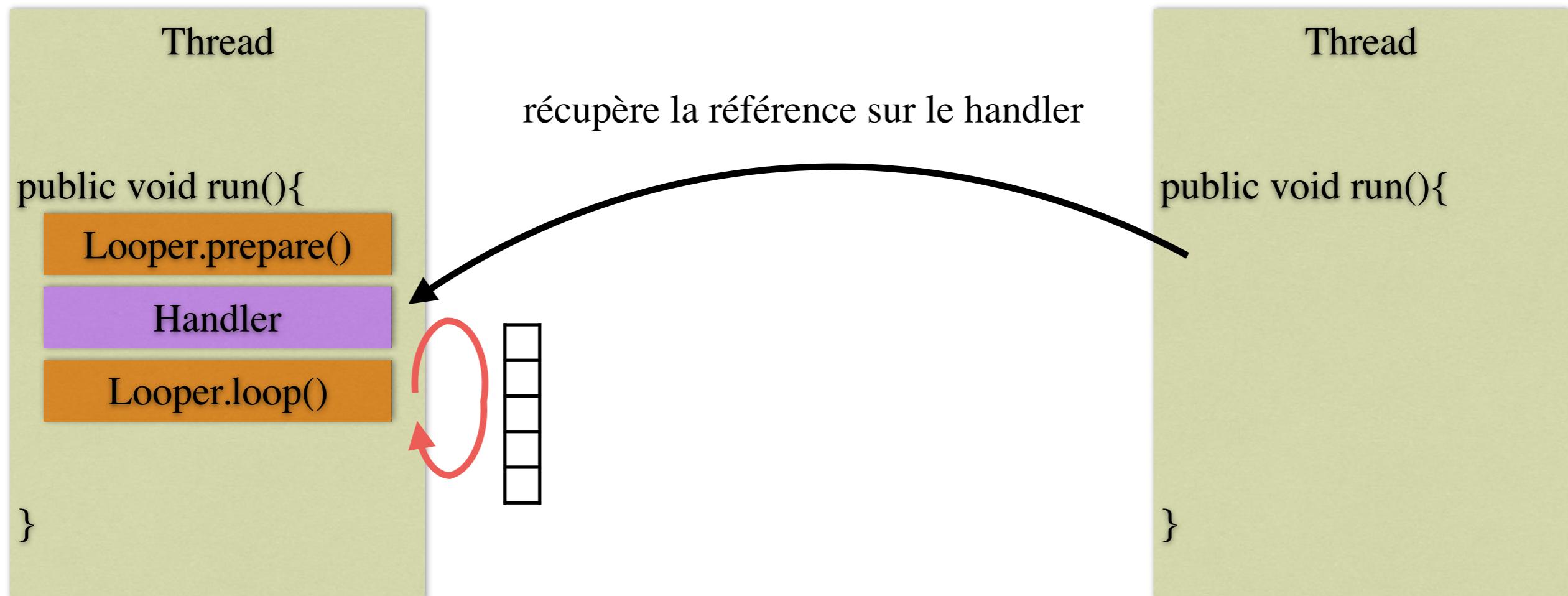
# Loopers, Messages et Handlers

21



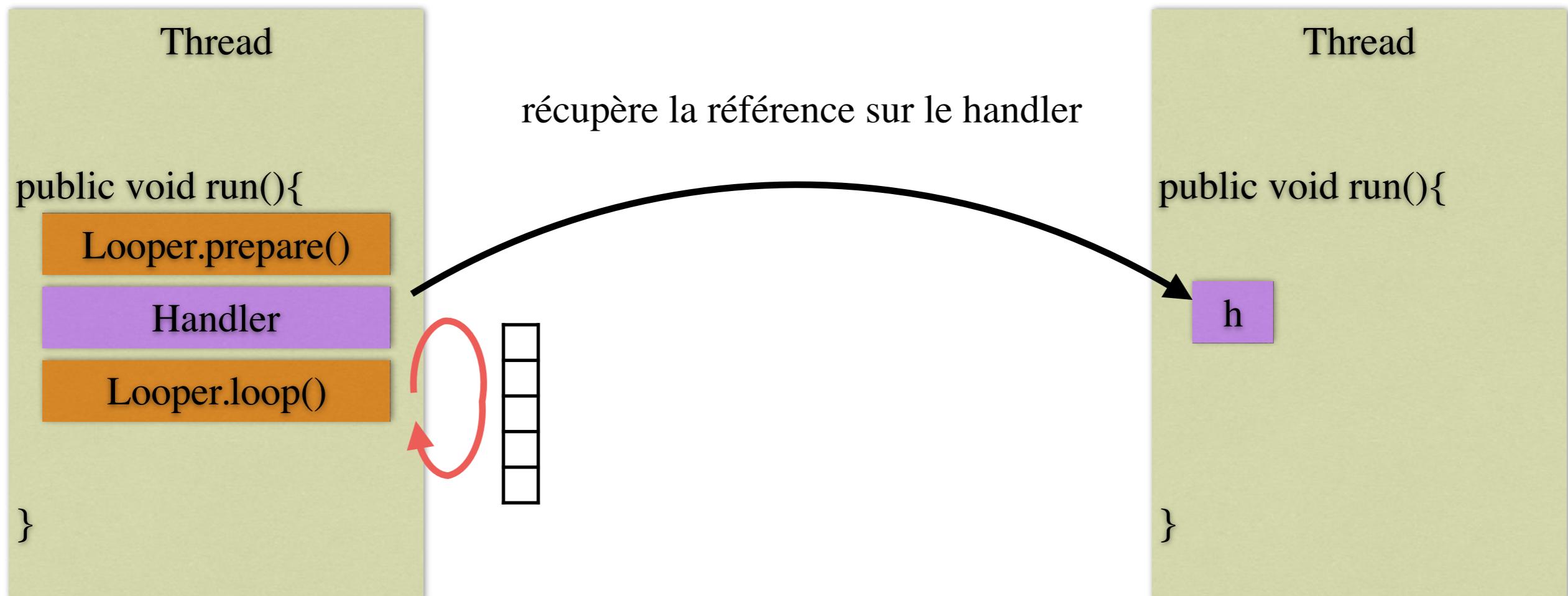
# Loopers, Messages et Handlers

21



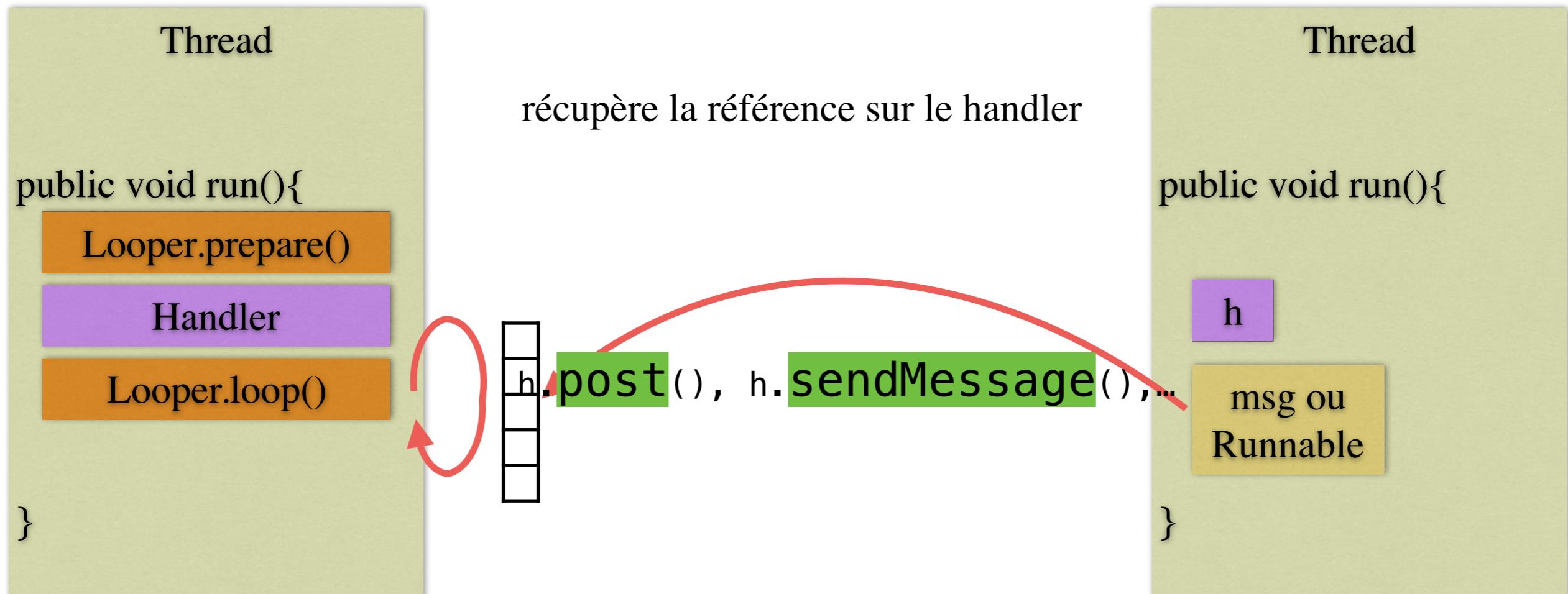
# Loopers, Messages et Handlers

21



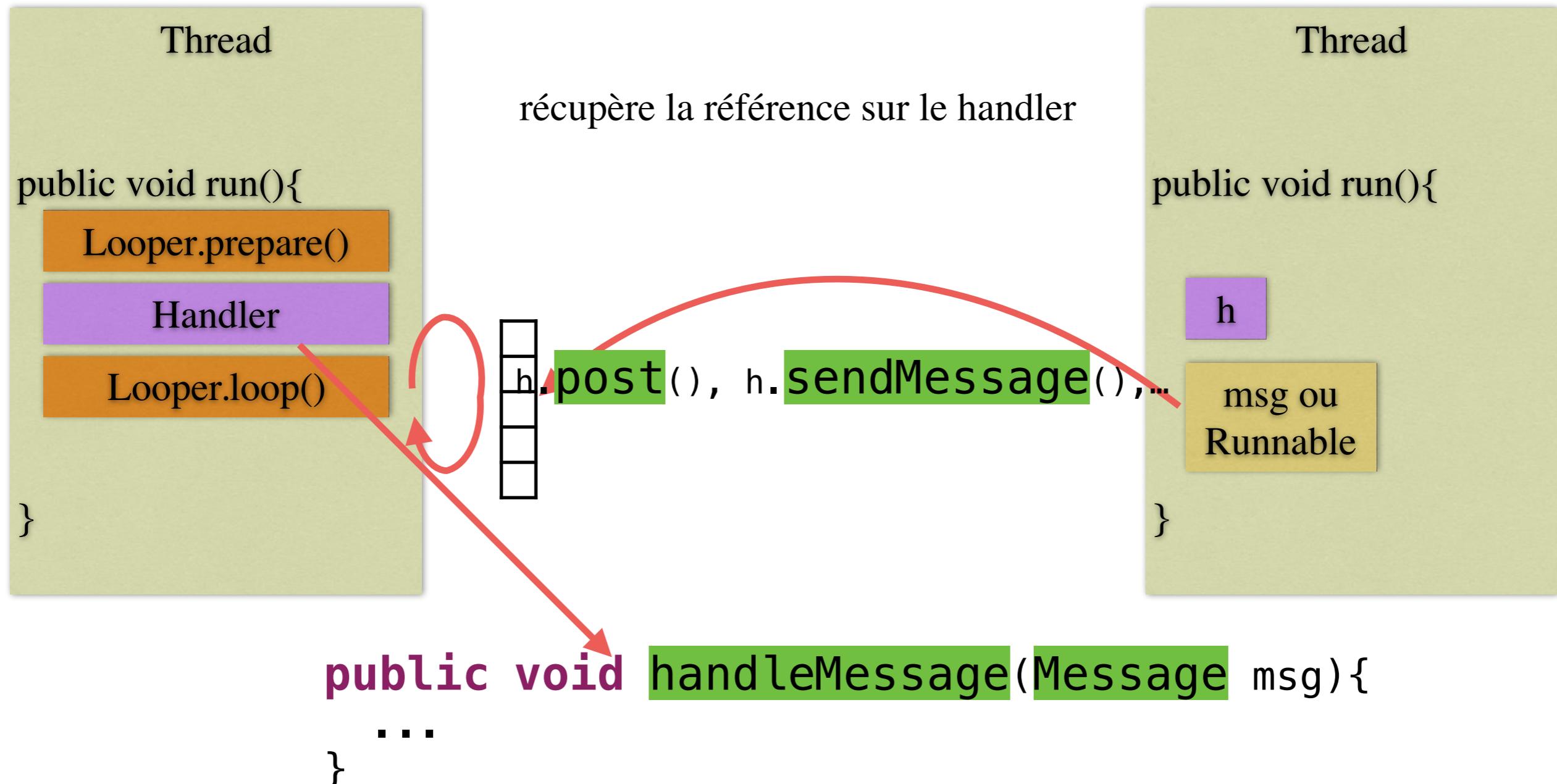
# Loopers, Messages et Handlers

21



# Loopers, Messages et Handlers

21



# AsyncTask

22

- C'est une classe paramétrique :  
`android.os.AsyncTask<Params, Progress, Result>`
- Il s'agit d'un outil de haut niveau permettant d'implanter facilement des tâches concurrentes interagissant avec le *UIThread*. Cette classe évite de manipuler directement les threads. Elle est dédiée à des tâches bornées dans le temps et pas «trop longues».
- Utilisation typique : création de sous classes qui généralement précise les types des paramètres :
  - Params : le type du paramètre à traiter par la tâche
  - Progress : le type de l'information sur le progrès de la tâche
  - Result : type du résultat de la tâche

# AsyncTask

23

- Le principe de fonctionnement repose sur une suite de méthodes :
  - `onPreExecute()` : appelée par le *UIThread* et permettant d'initialiser la tâche.
  - `doInBackground(Params...)` : demande de traitement, appelée par un autre thread (par exemple le *UIThread*). Peut-être appelée dès que `onPreExecute()` a terminé.
  - `onProgressUpdate(Progress...)` : appelée par le *UIThread* lorsqu'une information de progrès est publiée par la tâche. La publication est faite dans le corps de la méthode `doInBackground()` en appelant `publishProgress()`

# AsyncTask

24

- `onPostExecute(Result)` : appelée par le *UIThread* lorsque la tâche est terminée.
- `onCancelled(0bject)` : peut-être appelée de façon alternative si la tâche est annulée.
- la méthode `cancel()` : permet d'annuler un tâche en cours. L'annulation ne pourra être effective que si la méthode `doInBackground()` termine. Pour se faire, l'implantation de la méthode `doInBackground()` doit régulièrement vérifier l'absence d'annulation en appelant la méthode `isCancelled()` (responsabilité du programmeur)
- Une AsyncTask doit-être instanciée et démarrée (méthode `execute(Params...)`) dans le *UIThread*.