

Langage Java

Introduction aux threads

Qu'est-ce qu'un thread ?

- Un *thread* est une entité responsable de **charger du bytecode** en mémoire et de **l'exécuter**
- On appelle ça un "**processus léger**" ou "**fil d'exécution**"
- Il n'y a **qu'un seul *thread* principal** et son rôle est d'exécuter la méthode `main(String[] args)`
- À partir du *thread* principal, on peut lancer d'autres *threads*
- Tous les *threads* s'exécutent **en parallèle**
- **Note** : La JVM crée aussi des *threads* "spéciaux" pour son fonctionnement interne (par exemple *l'EventDispatchThread* pour gérer les événements graphiques Swing/JavaFX)

Et côté implémentation ?

- Pour afficher le nom du *thread* courant (ie. le *thread* dans lequel notre code s'exécute actuellement) :

```
public class MonProgramme {  
  
    public static void main(String[] args) {  
        Thread current = Thread.currentThread();  
        System.out.println(current.getName());  
    }  
}
```

```
> java MonProgramme  
main
```

Comment créer et lancer un nouveau *thread* ?

- Pour exécuter du code dans un *thread* autonome, on doit :
 - Écrire une classe qui implémente l'interface **Runnable**
 - Écrire notre code dans la méthode **run()**, exactement comme s'il s'agissait d'un `main()`
- Exemple : une classe Producteur dont le but sera de produire des objets :

```
public class Producteur implements Runnable {  
  
    @Override  
    public void run() { /* notre code ici */ }  
  
}
```

Comment créer et lancer un nouveau *thread* ?

- La méthode `run()` sera le **point d'entrée** de notre *thread*
- Ajoutons du code pour produire un téléviseur chaque seconde :

```
public class Producteur implements Runnable {  
    public void run() {  
        while (true) {  
            System.out.println("1 nouveau téléviseur produit");  
            Thread.sleep(1000); // On attend 1 seconde  
        }  
    }  
}
```

- À ce stade, nous avons toujours le choix d'exécuter ce code de manière **synchrone** ou **asynchrone**

Lancement synchrone

- Le lancement synchrone (bloquant) d'un objet Runnable revient simplement à exécuter sa méthode `run()` :

```
public static void main(String[] args) {  
  
    Producteur p = new Producteur();  
    p.run();  
  
    System.out.println("Fin du programme");  
}
```

- **Remarque** : on ne verra jamais le message "Fin du programme" et notre programme ne se terminera jamais

- Le lancement asynchrone (non bloquant) consiste à créer un **Thread** en lui donnant l'objet `Runnable` en paramètre du constructeur, puis à le démarrer avec la méthode `Thread.start()` :

```
public static void main(String[] args) {  
  
    Producteur prod = new Producteur();  
    Thread t = new Thread(prod, "Thread-Producteur");  
    t.start();  
  
    System.out.println("Fin du programme");  
}
```

- Remarque** : cette fois on verra le message "Fin du programme" (mais le programme ne s'arrêtera pas car il reste un thread en cours d'exécution)

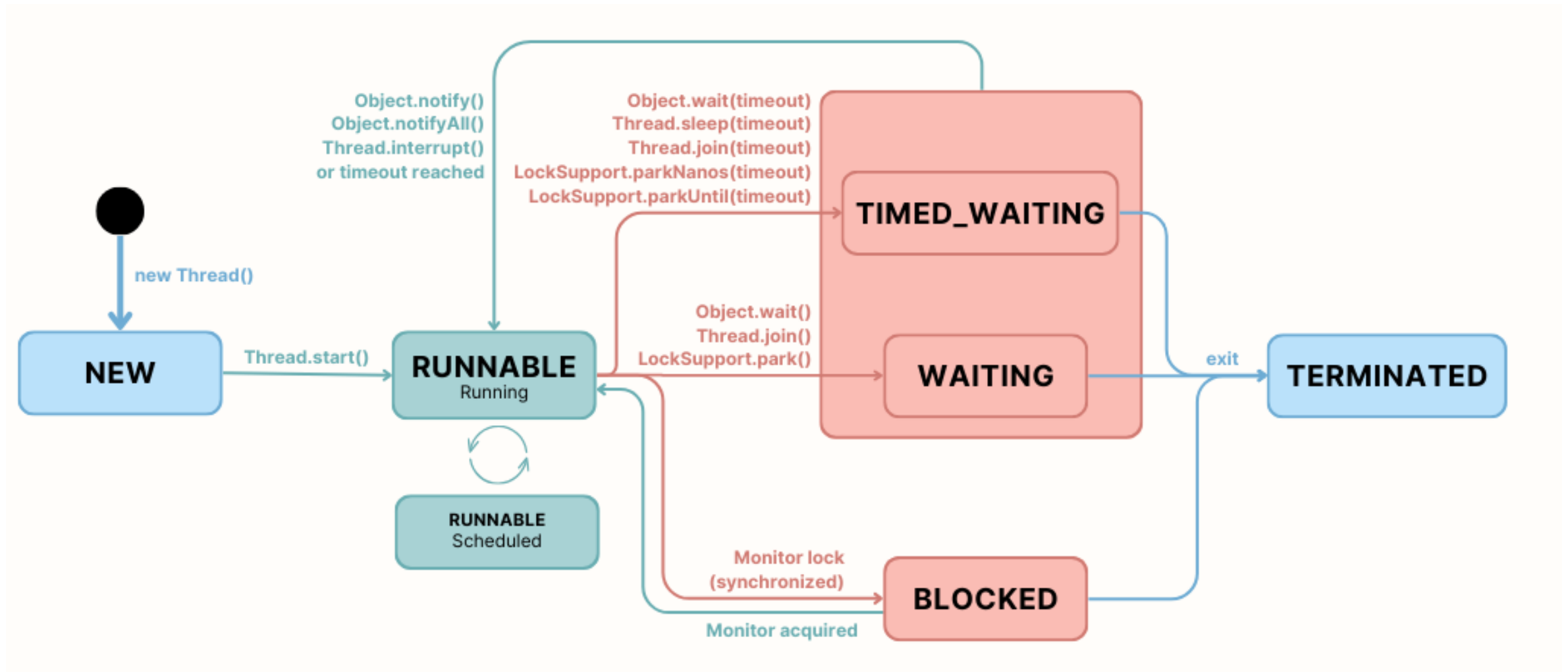


Cycle de vie et ordonnancement

Cycle de vie d'un Thread

- **NEW** : Etat dans lequel est un Thread juste après sa création (`new Thread()`)
- **RUNNABLE** : En cours d'exécution (après un appel à `Thread.start()`)
- **BLOCKED** : Lorsqu'il est dans cet état, un Thread est bloqué en attente d'un verrou détenu par un autre Thread (section **synchronized**)
- **WAITING** : Lorsque le Thread a été mis en attente par un appel à `Object.wait()` ou `Thread.sleep()`
- **TIMED_WAITING** : Idem WAITING mais pour une durée limitée dans le temps (typiquement lorsqu'on appelle les méthodes `Thread.sleep(millis)` ou `Object.wait(millis)`)
- **TERMINATED** : Le Thread est terminé
- Plus de détails ici : <https://www.baeldung.com/java-thread-lifecycle>

Cycle de vie d'un Thread



L'ordonnancement des *threads* RUNNABLE

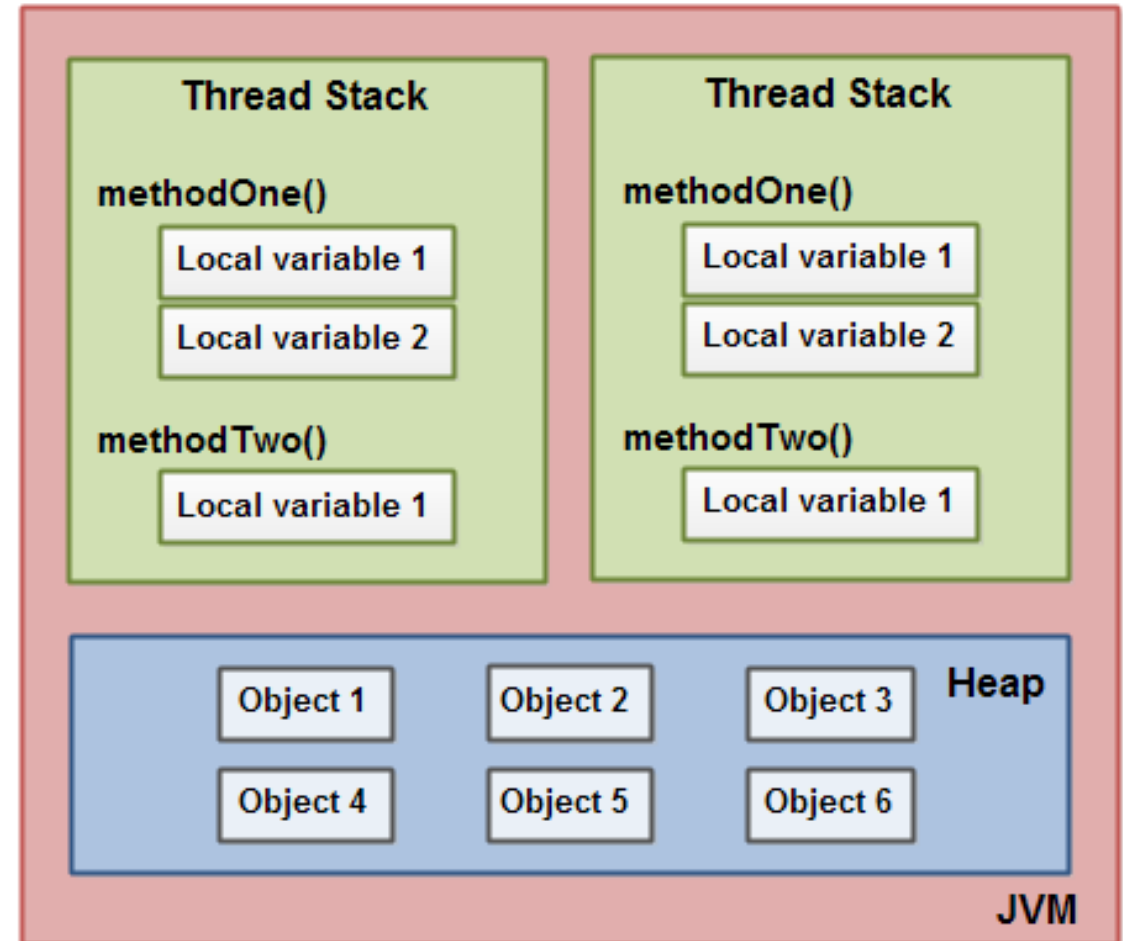
- Les *threads* à l'état RUNNABLE ne sont pas exécutés en parallèle : ils sont exécutés **tour à tour** pour **simuler** une exécution en parallèle (*sauf cas particulier du ForkJoinPool et *threads* virtuels)
- Un *thread* dans l'état RUNNABLE sera en réalité soit "RUNNING" (actif) ou "SCHEDULED" (programmé) mais ces **2 sous-états ne sont pas modélisés dans l'API**
- Quand un *thread* devient le *thread* actif (RUNNING), on dit qu'**il prend la main** ou qu'il **préempte** les ressources de calcul (il obtient la priorité)
- Le **temps d'exécution** alloué à chaque *thread* dépend de sa **priorité** ; c'est la JVM qui arbitre les durées d'exécution et qui fait en sorte de "réveiller" plus souvent les *threads* de forte priorité



La pile d'exécution

Le modèle de mémoire en Java (simplifié)

- En Java il existe 2 grandes zones de mémoire gérées par la JVM :
- **Heap space** : zone globale où sont stockés tous les objets créés avec "new" (y compris les *threads* créés avec `new Thread()` !)
- **Stack memory** : contient ce qu'on appelle la **pile d'exécution** de chaque **thread**. C'est un **concept fondamental** pour comprendre les *threads*.



<https://jenkov.com/tutorials/java-concurrency/java-memory-model.html>

Exemple

```
public class SomeClass
{
    public void methodOne()
    {
        int x = 5;
        int y = methodTwo(x);
    }

    public int methodTwo(int x)
    {
        int y = x*x + 5;
        return y ;
    }

    ...
}
```

Exemple

```
public class SomeClass
{
    public void methodOne() 2 variables locales
    {
        int x = 5;
        int y = methodTwo(x);
    }

    public int methodTwo(int x) 2 variables locales
    {
        int y = x*x + 5;
        return y ;
    }

    ...
}
```


Exemple

// On crée 2 threads pour exécuter `SomeClass.methodOne()` en parallèle

```
public static void main(String[] args)
{
    SomeClass sc = new SomeClass();

    new Thread(() -> sc.methodOne(), "T1").start();
    new Thread(() -> sc.methodOne(), "T2").start();
}
```

Exemple

// On crée 2 threads pour exécuter `SomeClass.methodOne()` en parallèle

```
public static void main(String[] args)
{
    SomeClass sc = new SomeClass();

    new Thread(() -> sc.methodOne(), "T1").start();
    new Thread(() -> sc.methodOne(), "T2").start();
}
```

Objets créés dans le HeapSpace

Démarrage de 2 threads = création de 2 piles
d'exécution dans le StackSpace

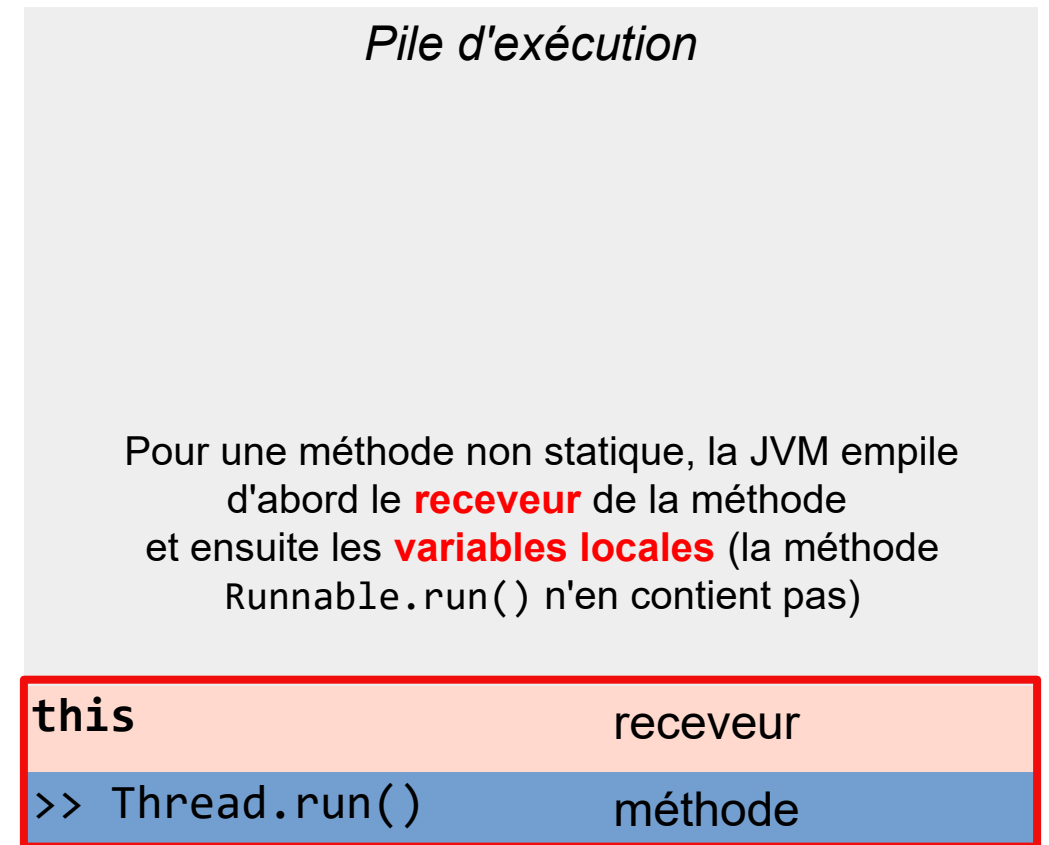
Que se passe-t-il quand on crée un *thread* ?

- Quand vous créez un *thread*, la JVM alloue une nouvelle **pile d'exécution** (une structure de données de type FIFO) en mémoire
- À chaque appel de méthode : la JVM **empile un nouveau contexte** d'exécution ; cette étape est coûteuse mais les futurs accès à la pile (push/pop) sont rapides
- La **première méthode** empilée sera la méthode **run()** du *thread*
- Durant l'exécution d'une méthode, des résultats de calculs sont empilés et dépilés en fonction des opérations (*opcodes*)
- À la fin de la méthode : la JVM **dépile le contexte** et laisse le résultat de la méthode sur la pile (ou rien si c'est une méthode void)

Comment fonctionne une pile d'exécution ?

- À chaque fois qu'une **méthode** est **appelée**, la JVM empile la référence de la méthode appelée
- La JVM empile ensuite **l'adresse du receveur(*)**
- À chaque opération suivante, des opérandes sont empilés et dépilés lorsque l'opération est terminée

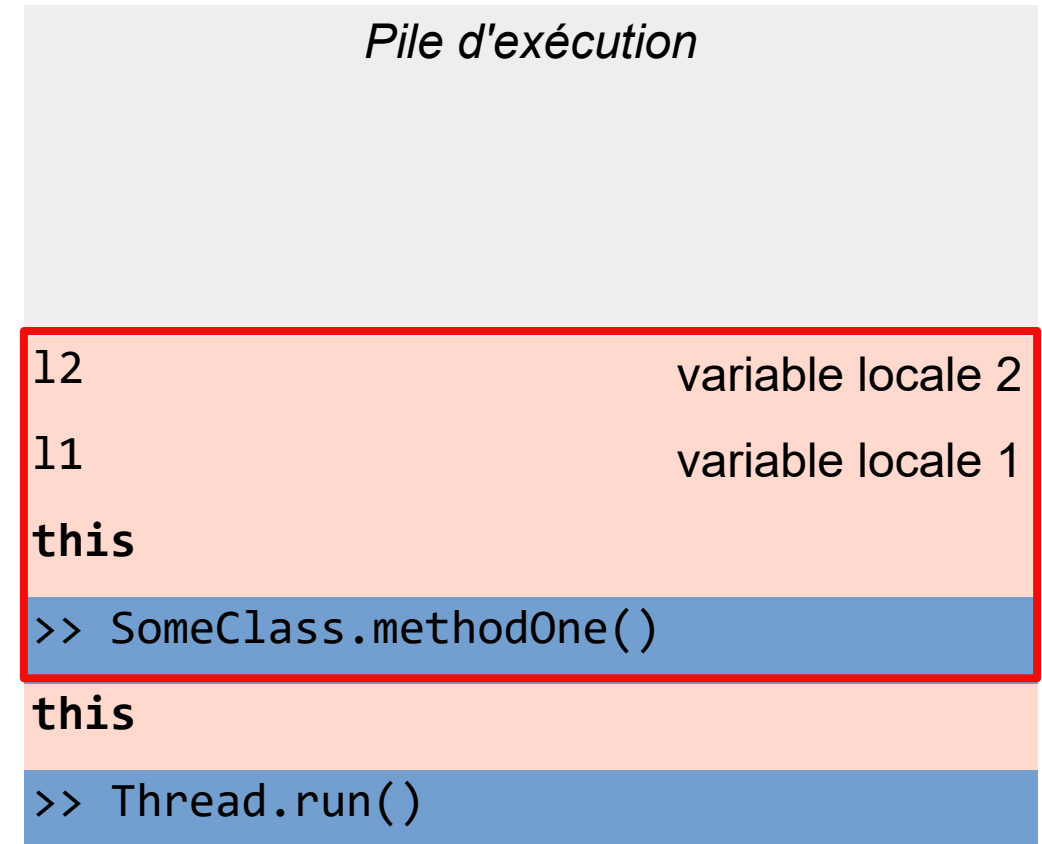
(*) En Programmation Orientée Objet, un appel de méthode sur un objet "obj" est un envoi de message et "obj" est nommé "**receveur**" du message.



En Java, 1 mot mémoire = 4 octets = 32 bits
Ici, 2 mots ⇒ **64 bits**

Comment fonctionne une pile d'exécution ?

- À chaque fois qu'une **méthode** est **appelée**, la JVM empile la référence de la méthode appelée
- La JVM empile ensuite **l'adresse du receveur(*)**
- À chaque opération suivante, des opérandes sont empilés et dépilés lorsque l'opération est terminée



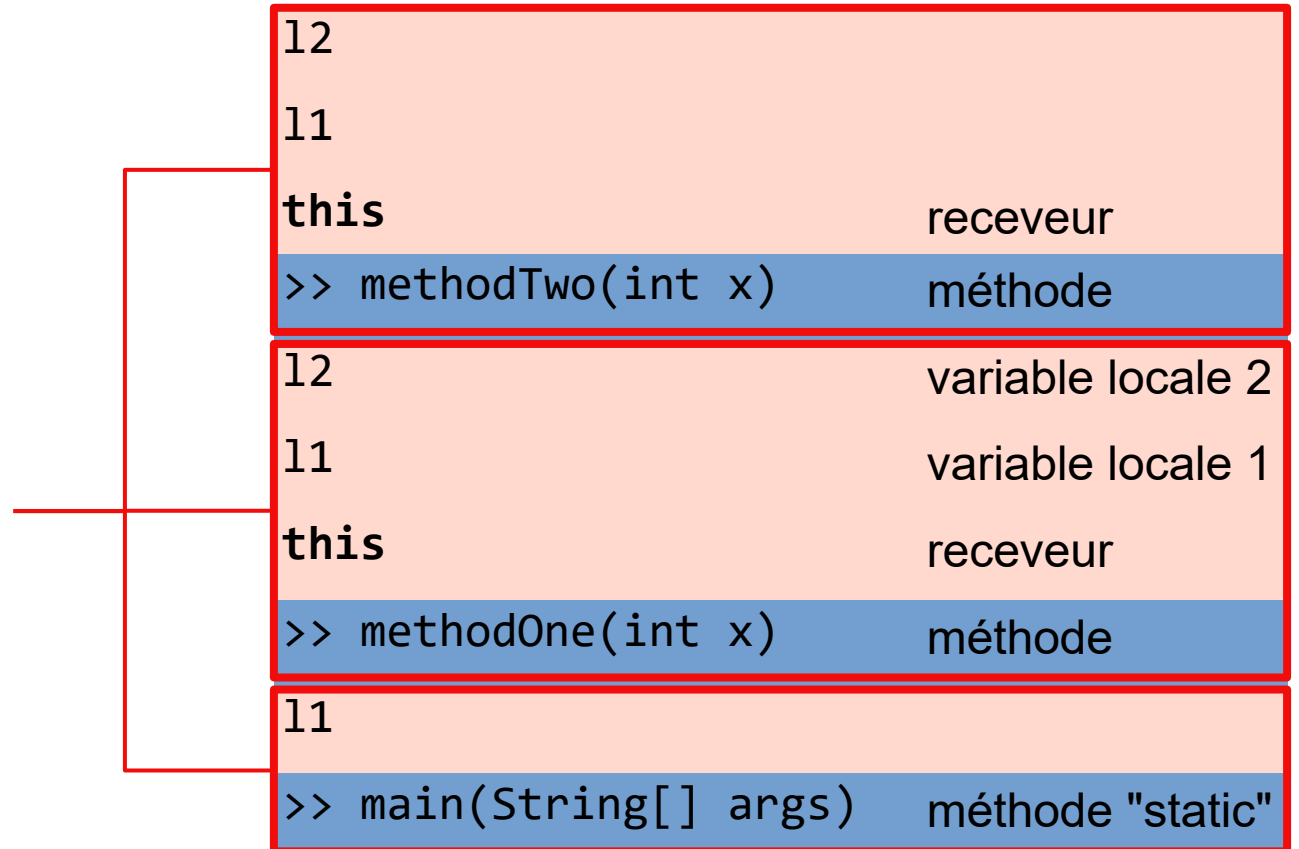
Comment fonctionne une pile d'exécution ?

- À chaque fois qu'une **méthode** est **appelée**, la JVM empile la référence de la méthode appelée
- La JVM empile ensuite **l'adresse du receveur(*)**
- À chaque opération suivante, des opérandes sont empilés et dépilés lorsque l'opération est terminée

```
12  
11  
this  
>> methodTwo(int x)  
12  
11  
this  
>> methodOne(int x)  
this  
>> run()
```

Comment fonctionne une pile d'exécution ?

- Une seule pile,
3 contextes d'exécution



Comment fonctionne une pile d'exécution ?

- **En fin de méthode** (instruction "return"), le contexte de la méthode est "**dépilé**"
- Si la méthode retourne un résultat (type de retour \neq void) il sera **laissé au sommet de la pile**

```
public int methodTwo(int x)
{
    int y = x*x + 5;
    return y ;
}
```

result of methodTwo()

12

11

this

>> methodOne(int x)

this

>> run()

Comment fonctionne une pile d'exécution ?

- En fin de méthode (instruction "return"), le contexte de la méthode est **dépilé**
- Si la méthode retourne un résultat (type de retour \neq void) il sera **laissé au sommet de la pile**
- Il sera ensuite **dépilé à son tour puis stocké dans la variable locale** dédiée de la méthode appelante (sauf s'il est ignoré dans le code)



```
l2=result of methodTwo()
```

```
l1
```

```
this
```

```
>> methodOne(int x)
```

```
this
```

```
>> run()
```

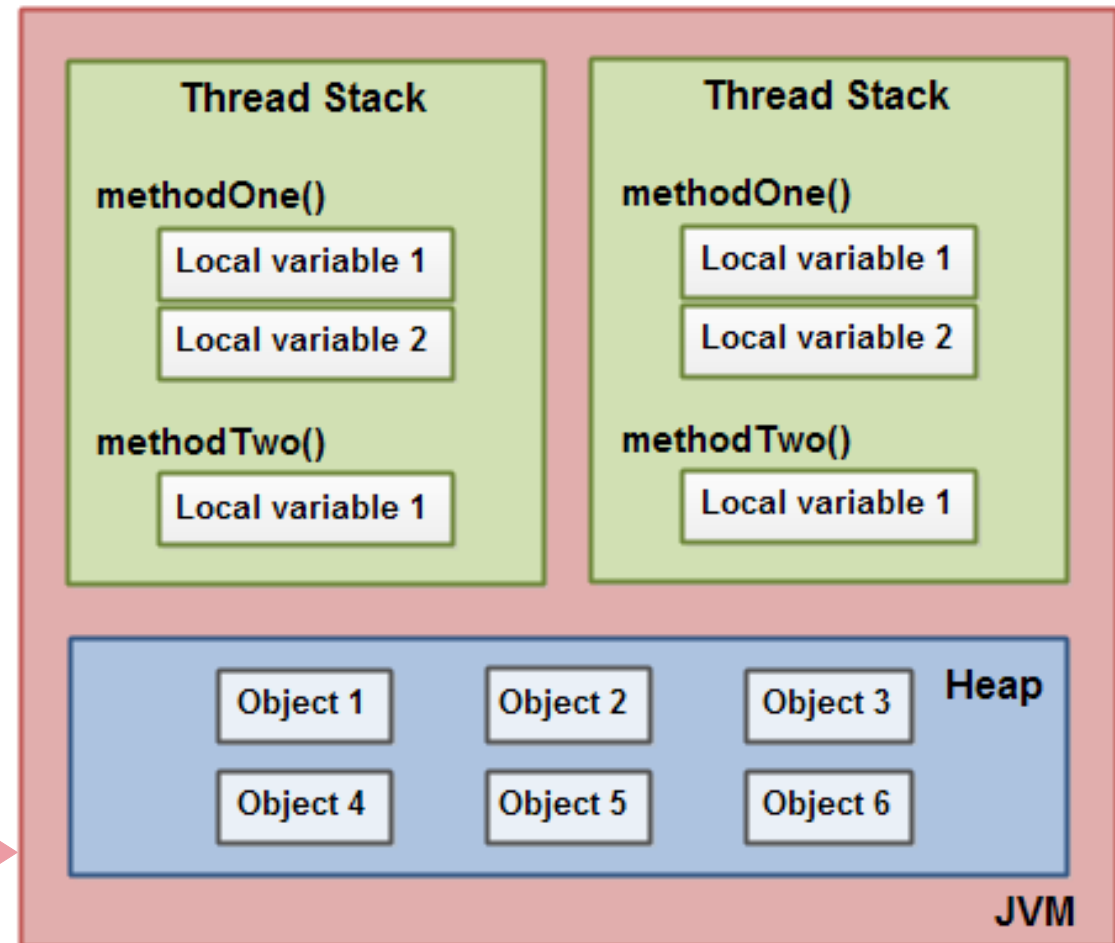
Exécution en debug avec point d'arrêt

- Voici un schéma de la pile :

```
public class SomeClass
{
    public void methodOne()
    {
        int x = 5;
        int y = methodTwo(x);
    }


    public int methodTwo(int x)
    {
        int y = x*x + 5;
        return y ;
    }
}
```

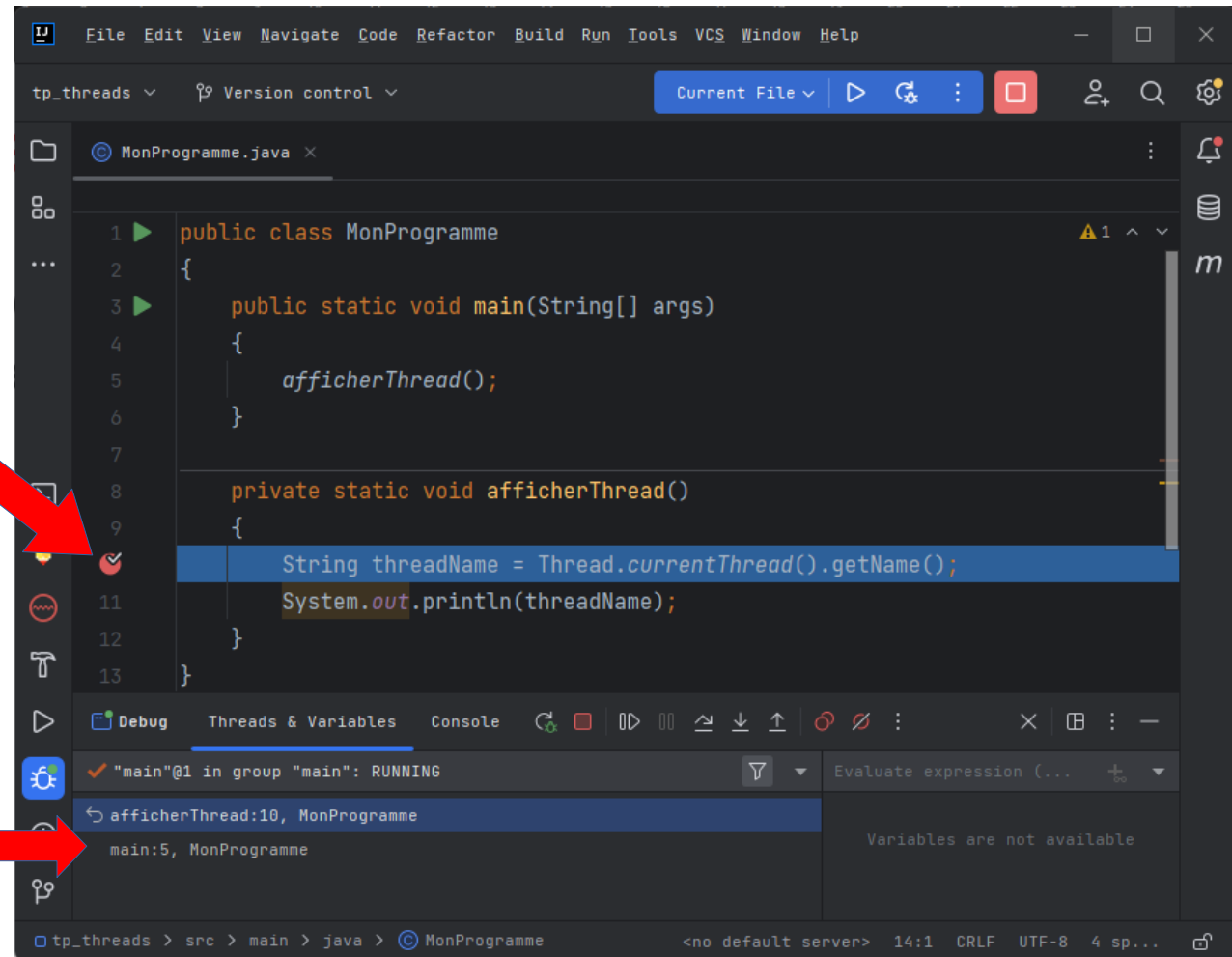
Breakpoint



<https://jenkov.com/tutorials/java-concurrency/java-memory-model.html>

Comment voir la pile d'exécution d'un *thread* ?

- Dans votre IDE, il faut mettre un point d'arrêt sur la ligne à laquelle vous voulez afficher l'état de la pile
- Puis lancer le programme en **mode Debug** 
- La pile s'affiche quand le point d'arrêt est atteint



Comment voir la pile d'exécution d'un *thread* ?

- Méthode 1 : avec `Throwable.printStackTrace()`

```
// Première méthode (pratique pour du debogage)  
new Throwable().printStackTrace();
```

```
// Affichera dans la console :
```

```
java.lang.Throwable  
    at MonProgramme.afficherThread(MonProgramme.java:10)  
    at MonProgramme.main(MonProgramme.java:5)
```

Comment voir la pile d'exécution d'un *thread* ?

- Méthode 2 : avec `Thread.getStackTrace()`

```
// Seconde méthode (ex.: pour rechercher une méthode particulière)
StackTraceElement[] array = Thread.currentThread().getStackTrace();
for (StackTraceElement elt : array) {
    if (elt.getMethodName().equals("someMethod")) ...
}
```

```
// Ici l'objectif n'est pas d'afficher la pile mais de naviguer
// dedans pour rechercher si on se trouve dans le flot d'exécution
// d'une méthode "someMethod()"
```

Comment voir l'état de tous les *threads* ?

- Voir tous les *threads* avec ThreadMXBean :

```
ThreadMXBean mx = ManagementFactory.getThreadMXBean();
ThreadInfo[] threadInfos = mx.dumpAllThreads(false, false);
for (ThreadInfo threadInfo : threadInfos)
{
    System.out.println(threadInfo);
}
```


Comment voir l'état de tous les *threads* ?

- Voir tous les *threads* avec ThreadMXBean :

```
ThreadMXBean mx = ManagementFactory.getThreadMXBean();  
ThreadInfo[] threadInfos = mx.dumpAllThreads(false, false);  
for (ThreadInfo threadInfo : threadInfos)  
{  
    System.out.println(threadInfo);  
}
```



```
"main" prio=5 Id=1 RUNNABLE  
  at java.management@19.0.2/sun.management.ThreadImpl.dumpThreads0(Native Method)  
  at java.management@19.0.2/sun.management.ThreadImpl.dumpAllThreads(ThreadImpl.java:541)  
  at java.management@19.0.2/sun.management.ThreadImpl.dumpAllThreads(ThreadImpl.java:528)  
  at app//MonProgramme.afficherThread(MonProgramme.java:17)  
  at app//MonProgramme.main(MonProgramme.java:12)  
  
"Reference Handler" daemon prio=10 Id=8 RUNNABLE  
  at java.base@19.0.2/java.lang.ref.Reference.waitForReferencePendingList(Native Method)  
  at java.base@19.0.2/java.lang.ref.Reference.processPendingReferences(Reference.java:245)  
  at java.base@19.0.2/java.lang.ref.Reference$ReferenceHandler.run(Reference.java:207)  
  
"Finalizer" daemon prio=8 Id=9 WAITING on java.lang.ref.NativeReferenceQueue$Lock@4d405ef7  
  at java.base@19.0.2/java.lang.Object.wait0(Native Method)  
  - waiting on java.lang.ref.NativeReferenceQueue$Lock@4d405ef7  
  at java.base@19.0.2/java.lang.Object.wait(Object.java:366)  
  at java.base@19.0.2/java.lang.Object.wait(Object.java:339)  
  at java.base@19.0.2/java.lang.ref.NativeReferenceQueue.await(NativeReferenceQueue.java:48)  
  at java.base@19.0.2/java.lang.ref.ReferenceQueue.remove0(ReferenceQueue.java:158)  
  at java.base@19.0.2/java.lang.ref.NativeReferenceQueue.remove(NativeReferenceQueue.java:89)  
  at java.base@19.0.2/java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:173)
```

- Permet d'afficher l'état des *threads*, leur priorité, leur type (normal/daemon), leurs moniteurs et leur pile d'exécution

Piloter l'exécution d'un thread

Les bonnes pratiques

- Un *thread* est fait pour exécuter du code sans être perturbé
- **Une fois démarré, on le laisse s'exécuter**
- **Il n'y a rien à faire pour le piloter !**
- La seule chose à faire c'est sécuriser les sections critiques des classes qui sont utilisées en contexte multithread, en utilisant des verrous/mutex...
- Mais en aucun cas il ne faut piloter manuellement un *thread* avec `Thread.suspend()`, `Thread.resume()` ou `Thread.stop()` car ces méthodes peuvent conduire à laisser votre programme dans un état indéterminé
- Seule exception : la méthode `Thread.interrupt()`

Comment arrêter un *thread* ?

- **C'est une mauvaise idée** car un *thread* s'arrête tout seul lorsque son code est terminé. S'il ne s'arrête pas, c'est souvent pour une **bonne raison** (et parfois c'est un bug)
- **Si vous estimez que c'est nécessaire : NE PAS UTILISER `Thread.stop()`**
⇒ la suite de l'exécution du programme est imprédictible
- À la place, on envoie un message `interrupt()` sur l'objet qui représente le *thread* en espérant que le thread en tienne compte...
 - ⇒ **Cas n°1** : le *thread* est `RUNNABLE` ⇒ il continue à s'exécuter mais son statut `isInterrupted()` passe à `true`
 - ⇒ **Cas n°2** : le *thread* est `WAITING` ⇒ lance une `InterruptedException`, et exécute le premier bloc "catch (`InterruptedException e`)" trouvé sur la pile du *thread*

Comment arrêter un *thread* ?

- Documentation officielle de la méthode `Thread.stop()` :

*This method is inherently **unsafe**.*

Stopping a thread with `Thread.stop` causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack).

*If any of the objects previously protected by these monitors were in an inconsistent state, **the damaged objects become visible to other threads**, potentially resulting in **arbitrary behavior**.*

Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its `run` method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt

Exemple pour arrêter un *thread* proprement

```
public class MonProgramme {  
  
    public static void main(String[] args)  
    {  
        Producteur p = new Producteur();  
  
        Thread thread = new Thread(p);  
        thread.start();  
  
        thread.interrupt(); // Tentative d'arrêt (non garantie)  
        thread.join(); // Attente de fin, potentiellement infinie  
  
        System.out.println("Fin du programme");  
    }  
}
```

Exemple pour arrêter un *thread* proprement

```
public class Producteur implements Runnable
{
    @Override
    public void run()
    {
        try
        {
            while ( !Thread.currentThread().isInterrupted() )
            {
                System.out.println("1 nouveau téléviseur produit");
                Thread.sleep(1000); // RUNNING-->WAITING pendant 1 seconde, puis RUNNING
            }
        }
        catch (InterruptedException e) // en dehors du "while(true)" sinon on boucle !
        {
            System.out.println("La production des téléviseurs s'arrête");
        }
    }
}
```


Exemple pour arrêter un *thread* proprement

```
public class Producteur implements Runnable
{
    @Override
    public void run()
    {
        try
        {
            while ( !Thread.currentThread().isInterrupted() )
            {
                System.out.println("1 nouveau téléviseur produit");
                Thread.sleep(1000); // RUNNING-->WAITING pendant 1 seconde, puis RUNNING
            }
        }
        catch (InterruptedException e) // en dehors du "while(true)" sinon on boucle !
        {
            System.out.println("La production des téléviseurs s'arrête");
        }
    }
}
```

```
> java MonProgramme
Fin du programme
1 nouveau téléviseur produit
La production des téléviseurs s'arrête
```

Comment attendre la fin d'un *thread* ?

- On envoie un message `join()` sur l'objet qui représente le *thread*
- \Rightarrow Le *thread* qui reçoit ce message va **bloquer le code appelant** jusqu'à ce que le *thread* soit terminé

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

    thread.join(); // Bloquant jusqu'à la fin du thread

    System.out.println("Fin du programme");
}
```

Les méthodes `wait()` et `notify()`

- Pour mettre le *thread* courant en attente **sans consommer de ressources CPU**, on peut appeler la méthode `Object.wait()`
- L'objet sur lequel on appelle la méthode `wait()` est appelé **verrou** ou **moniteur**
- Le *thread* courant va passer à l'état **WAITING**
- Pour débloquer le *thread*, **un autre thread** devra appeler la méthode `notify()` sur le moniteur

Les méthodes wait() et notify()

```
public class Producteur implements Runnable {

    private Object lock = new Object(); // objet qui servira de moniteur

    public void run()
    {
        lock.wait();
        System.out.println("1 nouveau téléviseur produit")
    }

    public static void main(String[] args) throws InterruptedException
    {
        Producteur p = new Producteur();
        Thread t = new Thread(p);

        t.start(); // Le thread "t" démarre mais va se mettre en attente juste après
        Thread.sleep(2000);
        lock.notify(); // On notifie le moniteur après 2 secondes ⇒ débloque le thread

        System.out.println("Fin du programme");
    }
}
```

Les méthodes `wait()` et `notify()`

```
public class Producteur implements Runnable {
```

```
    private Object lock = new Object(); // objet qui servira de moniteur
```

```
    public void run()
    {
        lock.wait();
        System.out.println("1 nouveau téléviseur produit")
    }
```

```
    public static void main(String[] args) throws Interruption {
```

```
        Producteur p = new Producteur();
        Thread t = new Thread(p);
```

```
        t.start(); // Le thread "t" démarre mais va se mettre en attente juste après
        Thread.sleep(2000);
        lock.notify(); // On notifie le moniteur après 2 secondes ⇒ débloque le thread
```

```
        System.out.println("Fin du programme");
```

```
    }
```

```
}
```

```
> java Producteur
```

2 secondes plus tard

```
> java Producteur
1 nouveau téléviseur produit
Fin du programme
```



Programmation concurrente

Un peu de vocabulaire

- Nous l'avons dit au début : les objets créés par un *thread* sont mis dans le **Heap Space** et sont visibles par tous les *threads*
- Deux *threads* qui accèdent à un même bloc de code sont dits **concurrents** et le bloc de code en question est appelé **section critique**
- Si on ne protège pas les sections critiques d'une classe contre les accès concurrents, on crée une **race condition** ⇒ risque de laisser les objets/ressources partagées dans un état incohérent
- Une classe protégée contre les accès concurrents est appelée **thread safe** ; corrolaire : **si une classe n'est pas thread safe, ne l'utilisez pas en contexte multithread !**

Section critique et *race condition*

- Exemple de section critique :

```
public void increment()  
{  
    this.count++;  
}
```

- Ca semble être à première vue une opération atomique (ie. indivisible). En réalité, cette instruction est décomposée ainsi par le compilateur :

```
public void increment()  
{  
    this.count = this.count + 1;  
}
```


Section critique et *race condition*

- Exemple de section critique :

```
public void increment()  
{  
    this.count++;  
}
```

- Ca semble être à première vue une opération atomique (ie. indivisible). En réalité, cette instruction est décomposée ainsi par le compilateur :

```
public void increment()  
{  
    this.count = this.count + 1;  
}
```

Il y a en réalité 4 instructions au niveau du code compilé ! Et il peut y avoir **préemption** (ie. changement du *thread* actif) au milieu de ces instructions

Que se passe-t-il lors d'une *race condition* ?

- Voici l'illustration d'une *race condition* :

```
public void increment()
{
    this.count++;
}

this.count = 24; // Valeur initiale

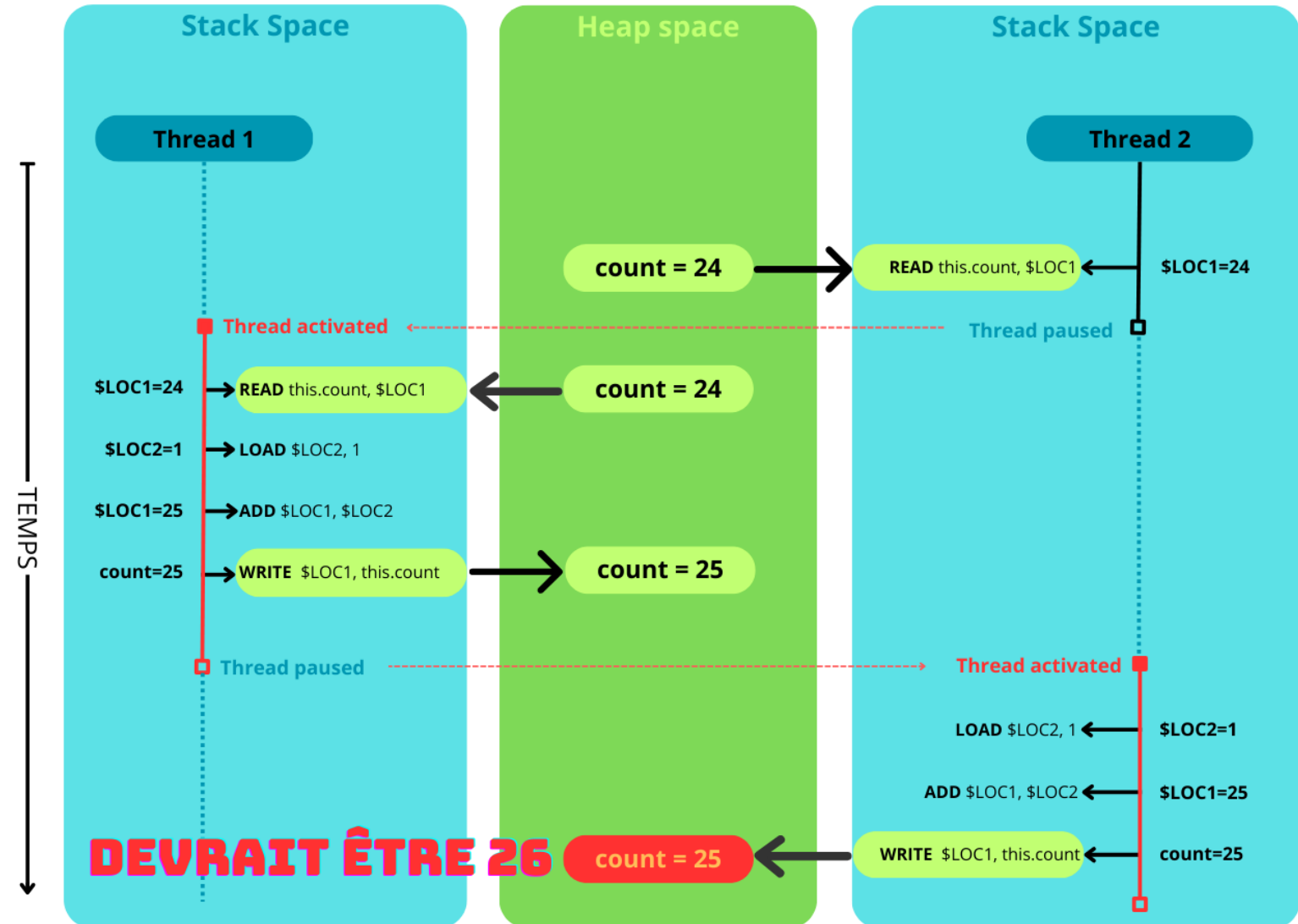
Runnable r = () -> increment();

// Lancement thread1
Thread t1 = new Thread(r, "Thread1");
t1.start();

// Lancement thread2
Thread t2 = new Thread(r, "Thread2");
t2.start();

t1.join(); // Attente thread1
t2.join(); // Attente thread2

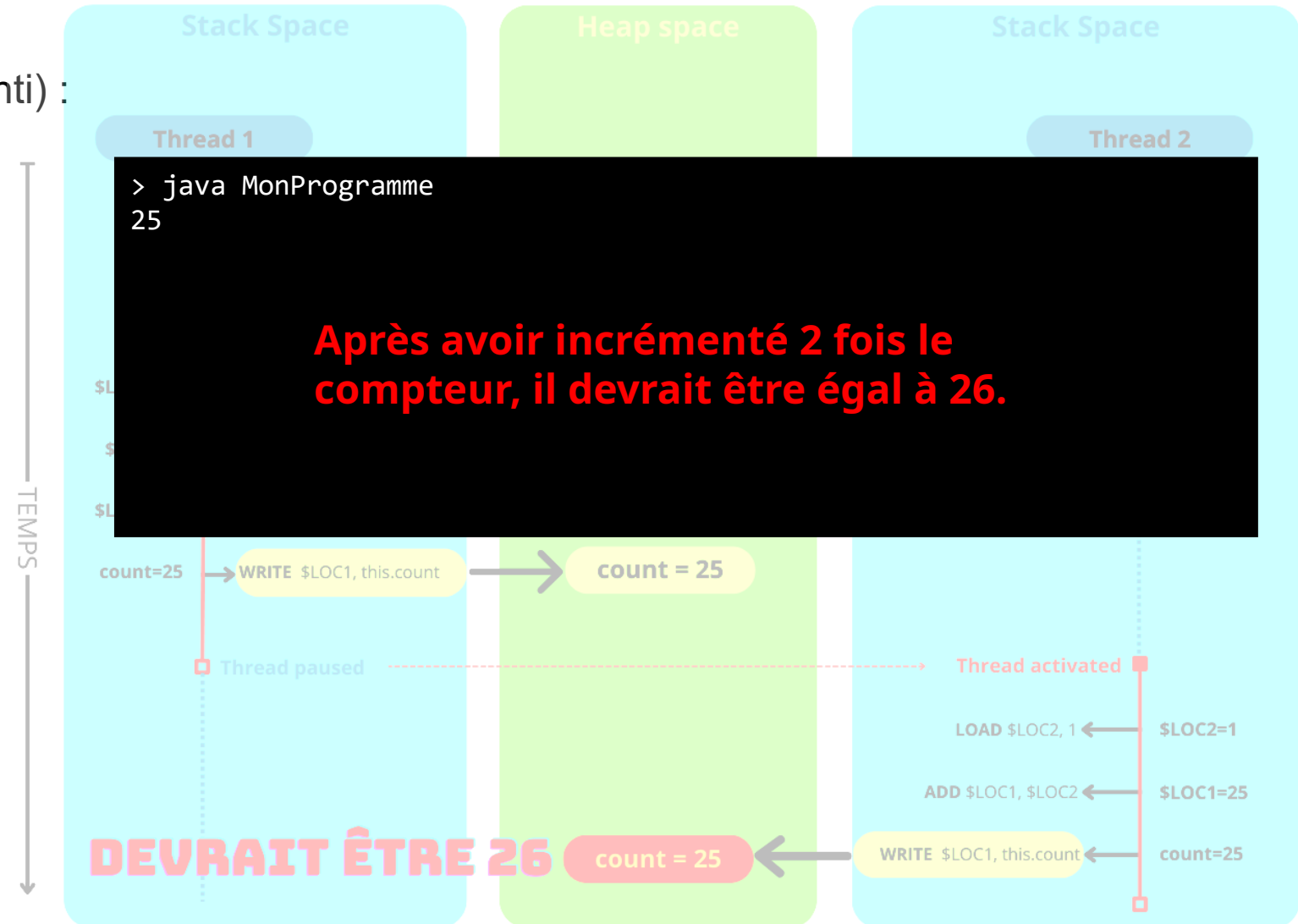
System.out.println(this.count);
```



Que se passe-t-il lors d'une *race condition* ?

- Résultat possible (non garanti) :

```
public void increment()  
{  
    this.count++;  
}  
  
this.count = 24; // Valeur initiale  
  
Runnable r = () -> increment();  
  
// Lancement thread1  
Thread t1 = new Thread(r, "Thread1");  
t1.start();  
  
// Lancement thread2  
Thread t2 = new Thread(r, "Thread2");  
t2.start();  
  
t1.join(); // Attente thread1  
t2.join(); // Attente thread2  
  
System.out.println(this.count);
```



Comment écrire du code *thread safe* ?

- Pour sécuriser une section critique, on utilise des blocs dits "**synchronized**"
- Un bloc synchronized ne peut être exécuté que par **un seul *thread* à la fois**
- Une classe peut définir plusieurs blocs synchronized, ils seront alors **TOUS mutuellement exclusifs**
- On peut synchroniser une méthode complète ou bien juste un bloc déclaré avec le mot-clé `synchronized(lock) { ... }` où lock est l'objet qui joue le rôle de **moniteur**
- Lorsqu'un *thread* veut accéder à un bloc synchronized, il doit demander le moniteur de la classe. S'il l'obtient, les autres *threads* voulant accéder à n'importe quel bloc synchronized du même moniteur seront bloqués (état **BLOCKED**) jusqu'à ce que le premier *thread* relâche ce moniteur