IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Processus

F. Dagnat & J. Mallet

Concurrence – C3

janvier 2023

1. Introduction à la modélisation de la concurrence
2. Modéliser un comportement
3. Programmation concurrente
   - C3 : Processus
   - TD5-TP4 : le tampon borné
   - C4 : Synchronisation
   - TD6-TP5 : lecteurs rédacteurs
4. Du modèle au code
5. Les propriétés et la vérification
6. µ-projet : Les trains

1 Introduction

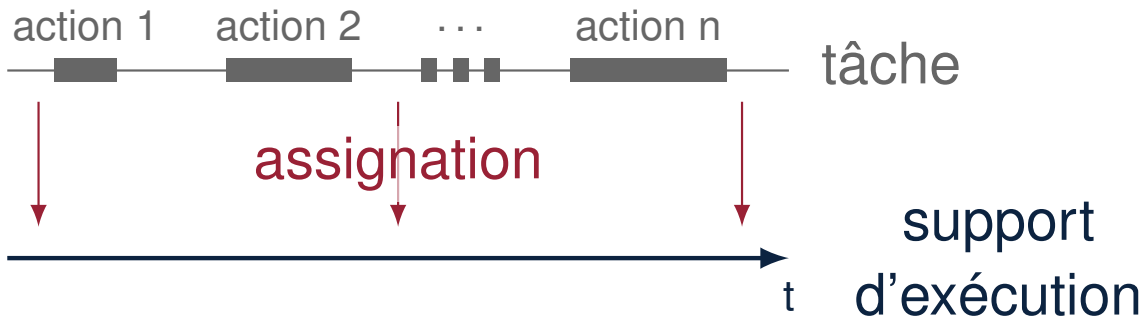
2 Les Threads Java

3 La difficulté de la concurrence

4 Exclusion mutuelle

5 Conclusion

Un unique support d'exécution

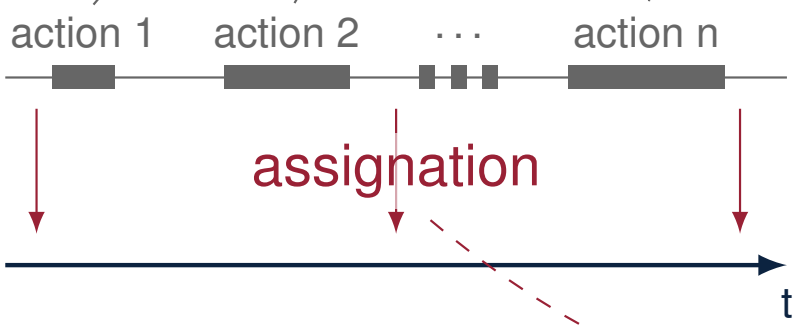


Un unique support d'exécution

```java
public class Product {
  private final String name;
  private final int number;
  private static int numberCreated = 0;
  public Product(String name) {
    this.name = name;
    this.number = ++numberCreated;
  }
  public String toString() {
    return "Produit " + this.name + " de num " + this.number;
  }
}
```

```java
public class Stock {
  private final Product[] content;
  private int size = 0;
  public Stock(int s) { this.content = new Product[s]; }
  public void add(Product p) {
    if (p == null)
      return;
    this.content[this.size++] = p;
  }
  public boolean isEmpty() { return this.size == 0; }
  public boolean isFull() { return this.size == this.content.length; }
  public Product remove() {
    Product p = this.content[--this.size];
    this.content[this.size] = null;
    return p;
  }
  public String toString() {
    if (isEmpty())
      return "Le Stock est vide.";
    String s = "Le Stock contient : ";
    for (int i = 0; i < this.size; i++)
      s += "\n\t" + this.content[i];
    return s;
  }
}
```

code des méthodes



assignation

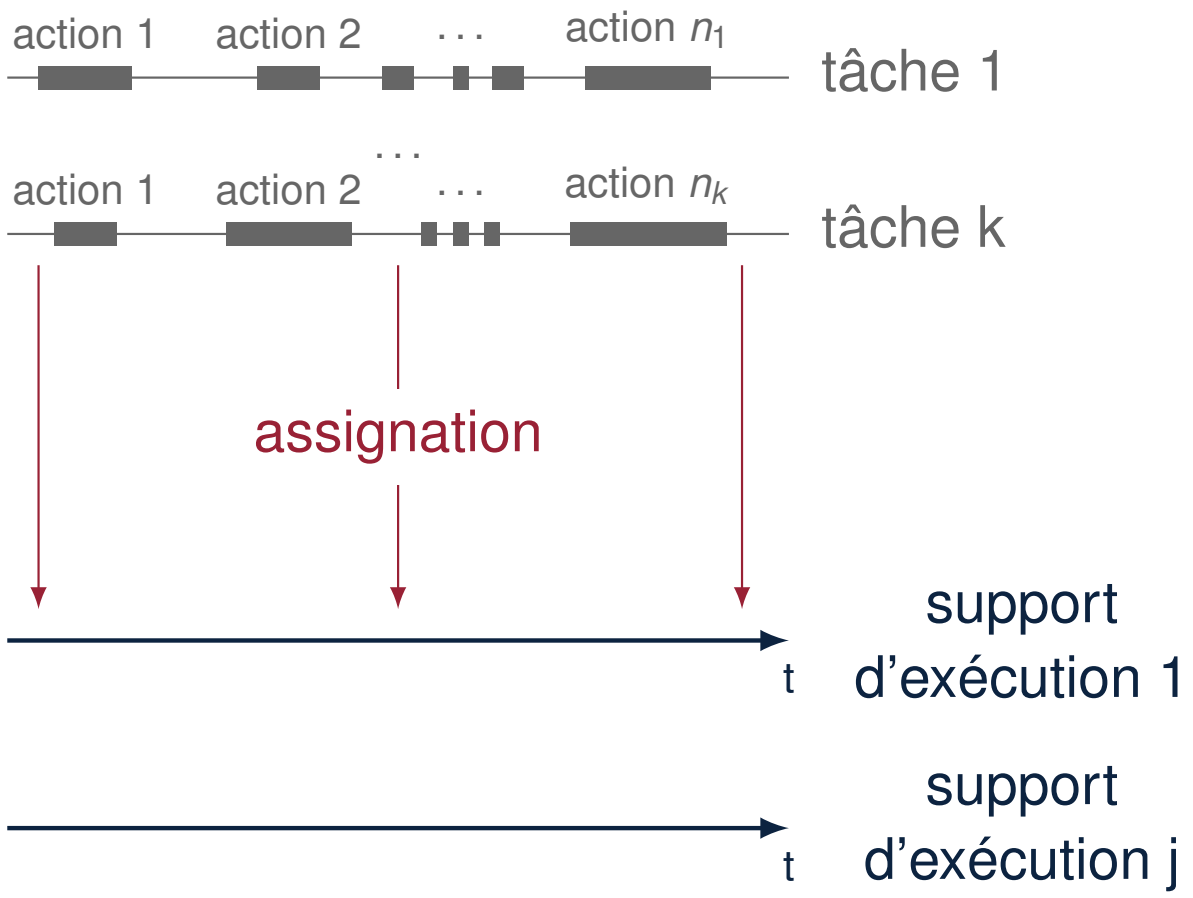
tâche

support d'exécution

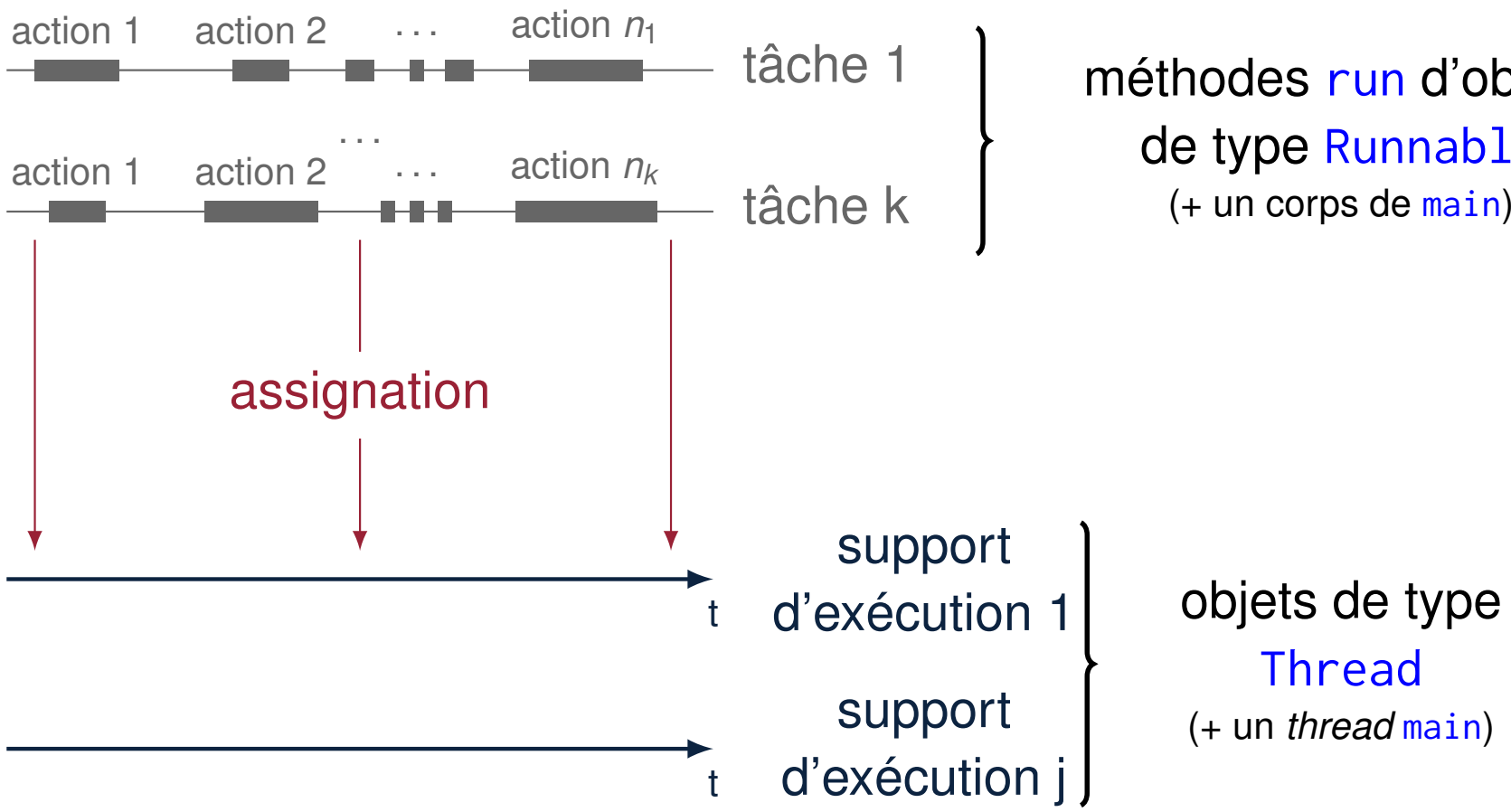
```java
public class Main {
  public static void main(String[] args) {
    Stock s = new Stock(8);
    System.out.println("Stock Vide : " + s.isEmpty());
    s.add(new Product("p1"));
    System.out.println("Stock Vide : " + s.isEmpty());
    System.out.println(s);
    s.add(new Product("p2"));
    System.out.println(s);
    System.out.println(s.remove() + " est retire");
    s.add(new Product("p3"));
    System.out.println(s);
    System.out.println("Ajoutons 13 produits");
    for (int i = 1; i <= 13; i++) {
      if (!s.isFull())
        s.add(new Product("ppp" + i));
    }
    System.out.println(s);
  }
}
```

placement dans le `main`

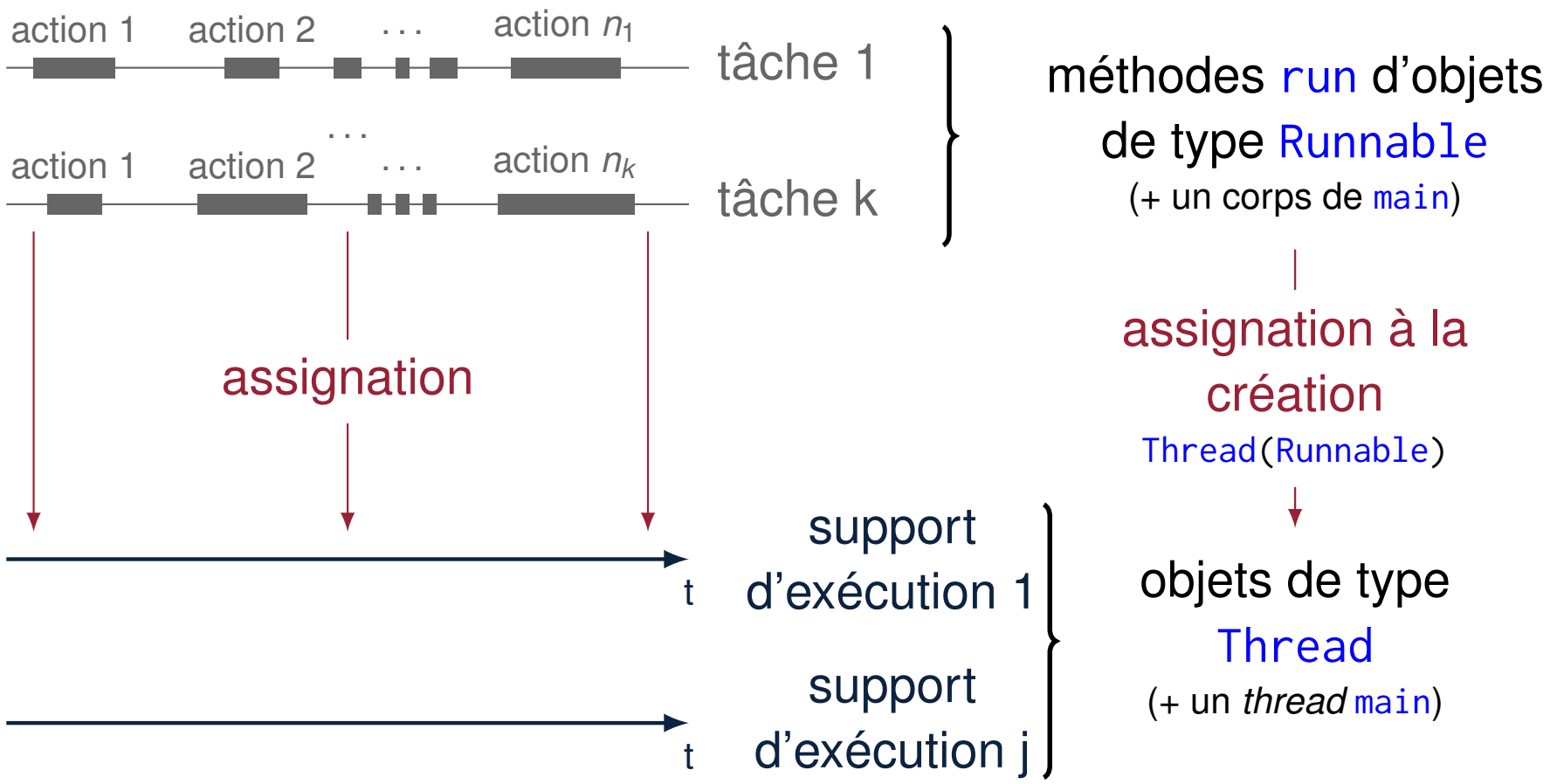
Plusieurs supports d'exécution



Plusieurs supports d'exécution



Plusieurs supports d'exécution



1 Introduction

2 Les Threads Java

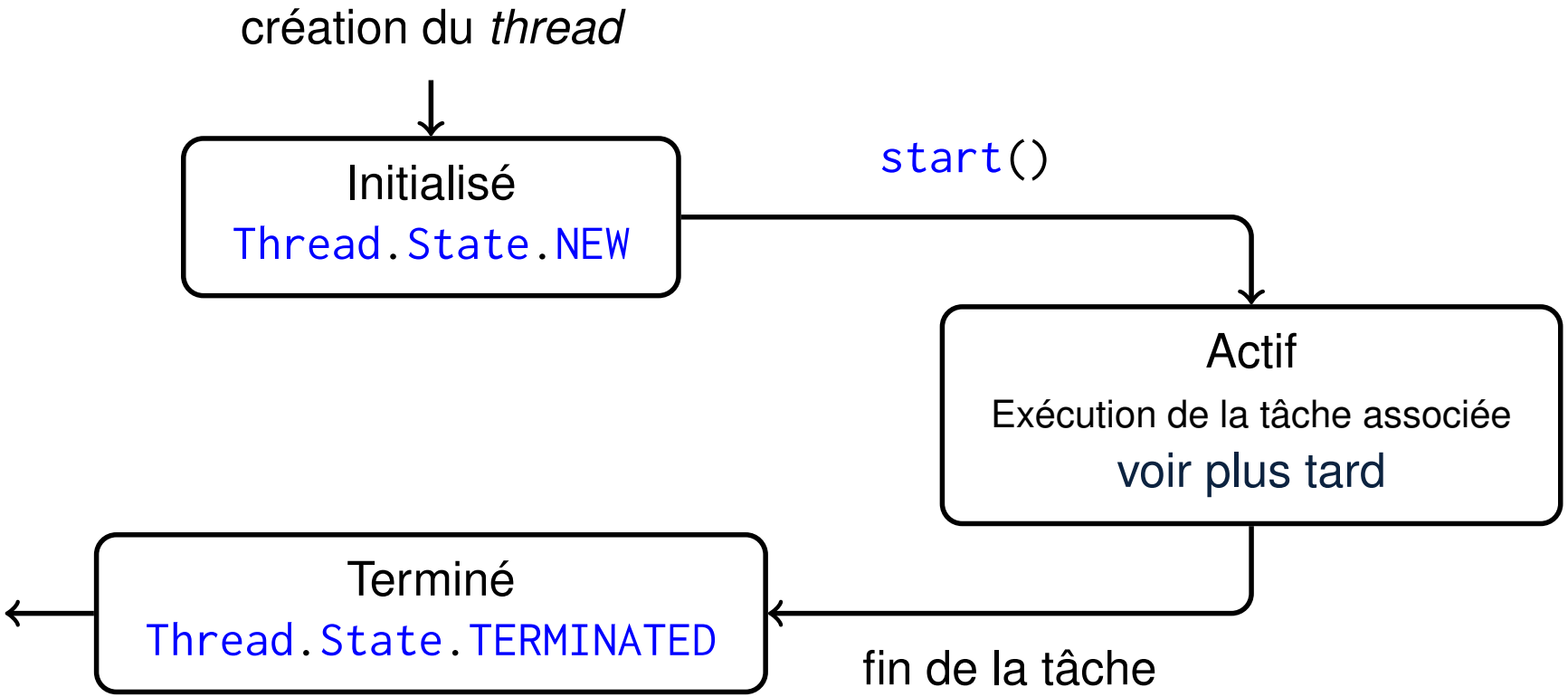
3 La difficulté de la concurrence

4 Exclusion mutuelle

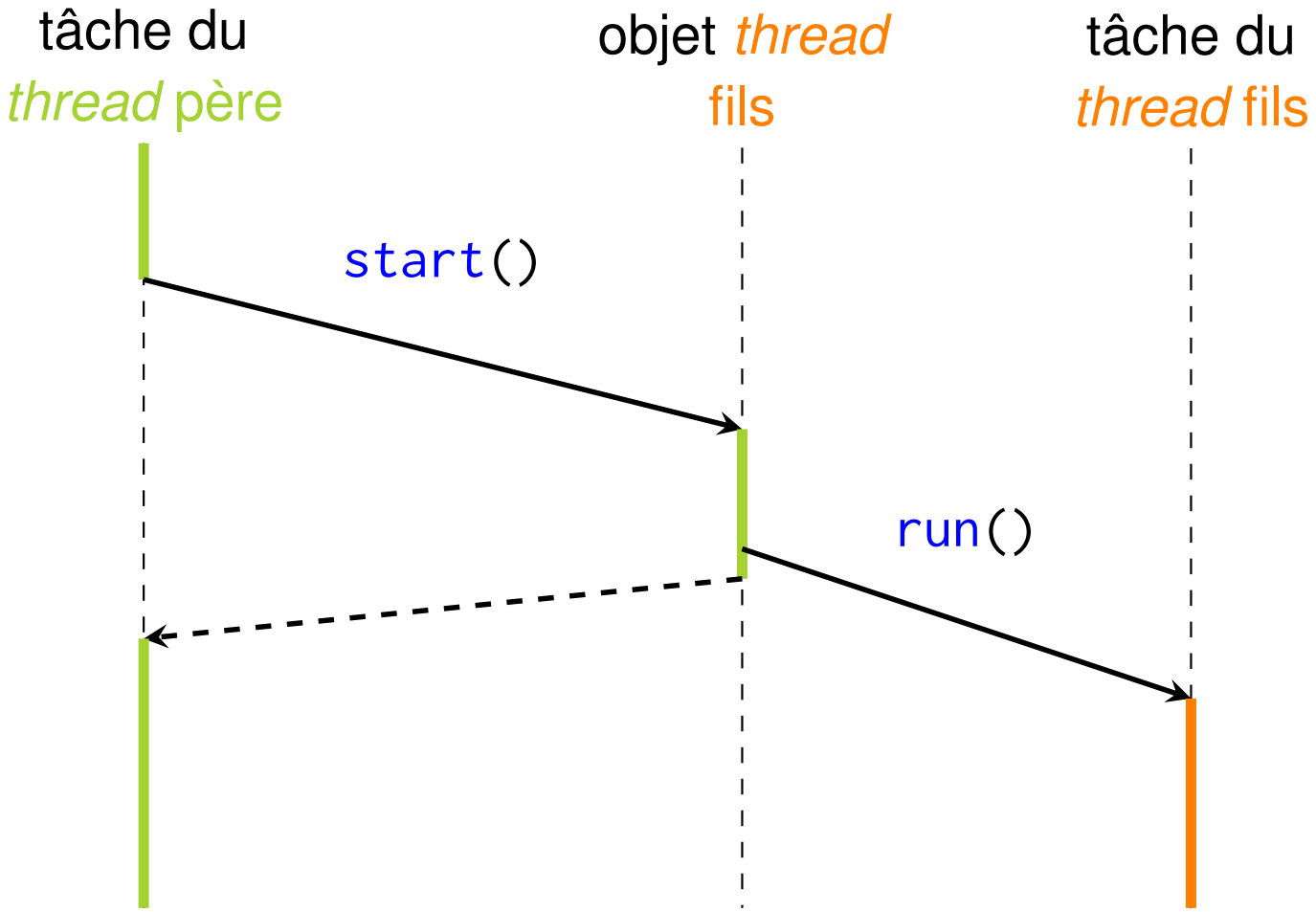
5 Conclusion

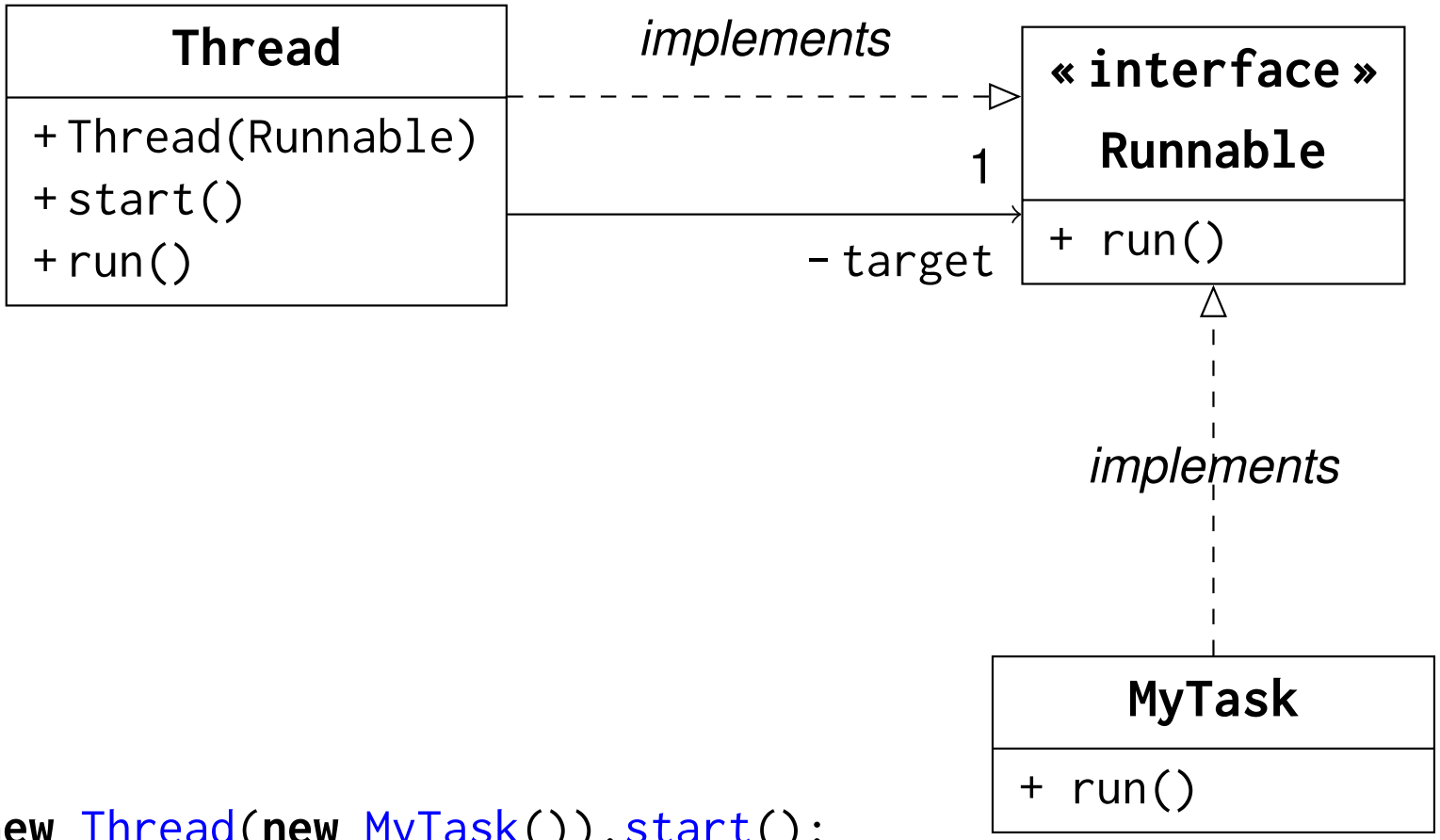
- Un objet de type `Thread` a
  - un nom — `setName`, `getName`
  - un état — `Thread.State getState()`
  - une tâche — `Thread(Runnable)`
  - une priorité — `setPriority`, `getPriority`
  - une forme (démon ou non) — `setDaemon()` avant son démarrage
  - une méthode pour être démarré — `start()`
- Thread `main`
  - démarré automatiquement, tâche = corps du `main`
  - peut créer des *threads* qui créent d'autres *threads*
  - ⇒ arborescence de *threads* (parent transmet priorité et forme)
  - ⇒ tout code est exécuté par un *thread* — `Thread.currentThread()`
  - JVM termine si uniquement des démons ou `System.exit(int)`

Le cycle de vie d'un *thread*



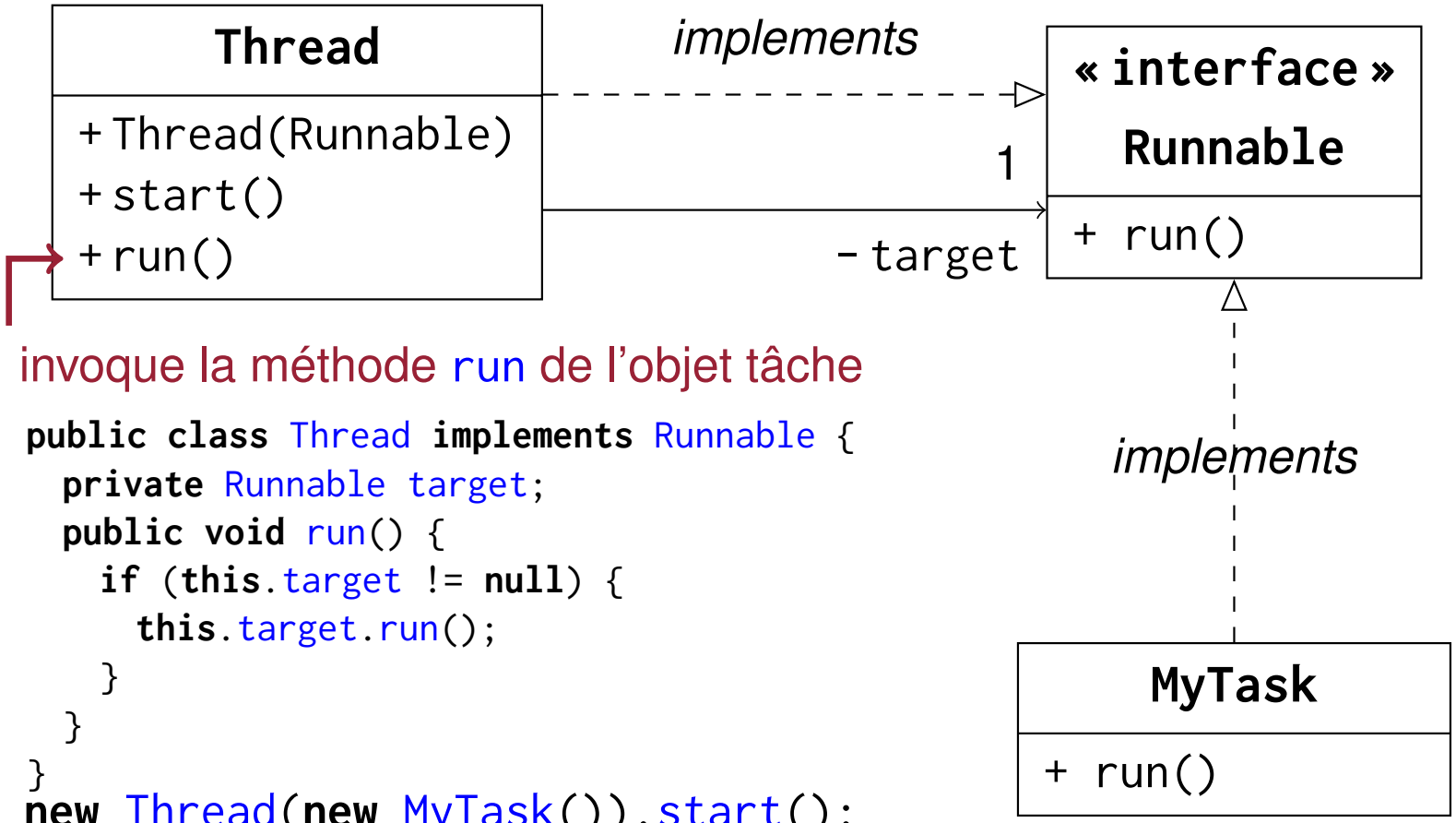
- Le *thread* ne peut être démarré qu'une seule fois
  - sinon `IllegalThreadStateException`


tâche du
thread père
objet *thread*
fils
tâche du
thread fils
start()
run()




```
new Thread(new MyTask()).start();
```

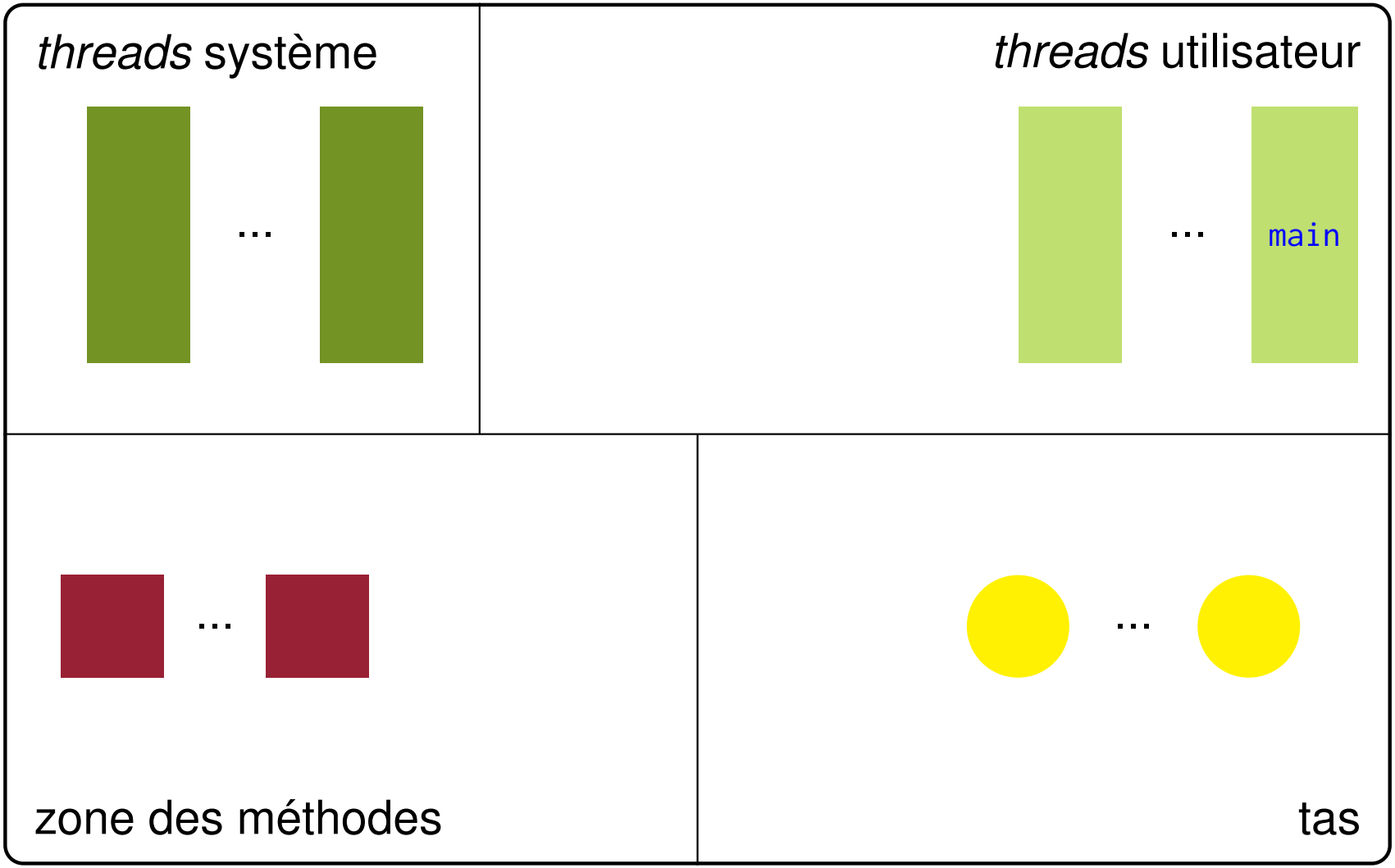
on parle de création par composition

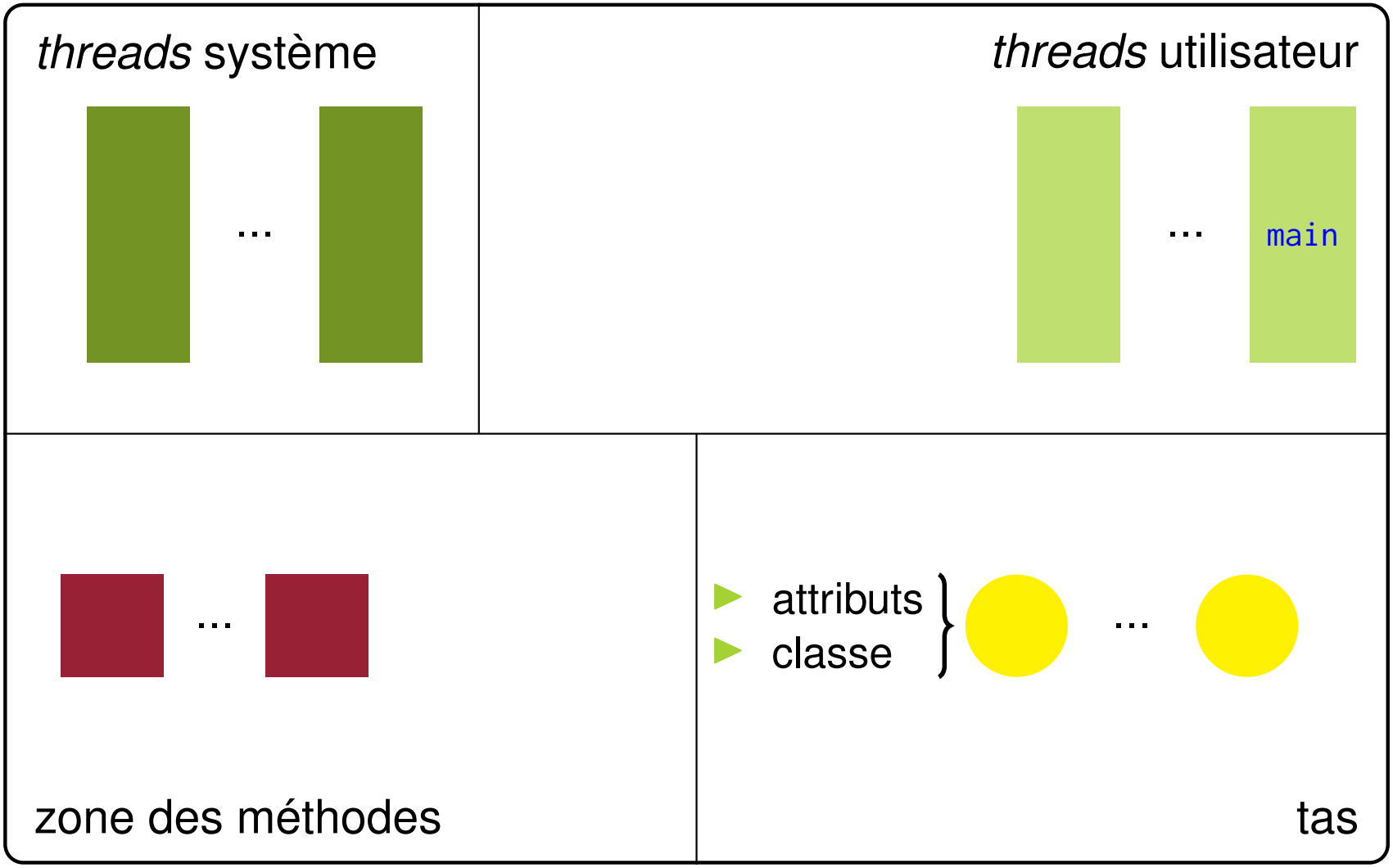


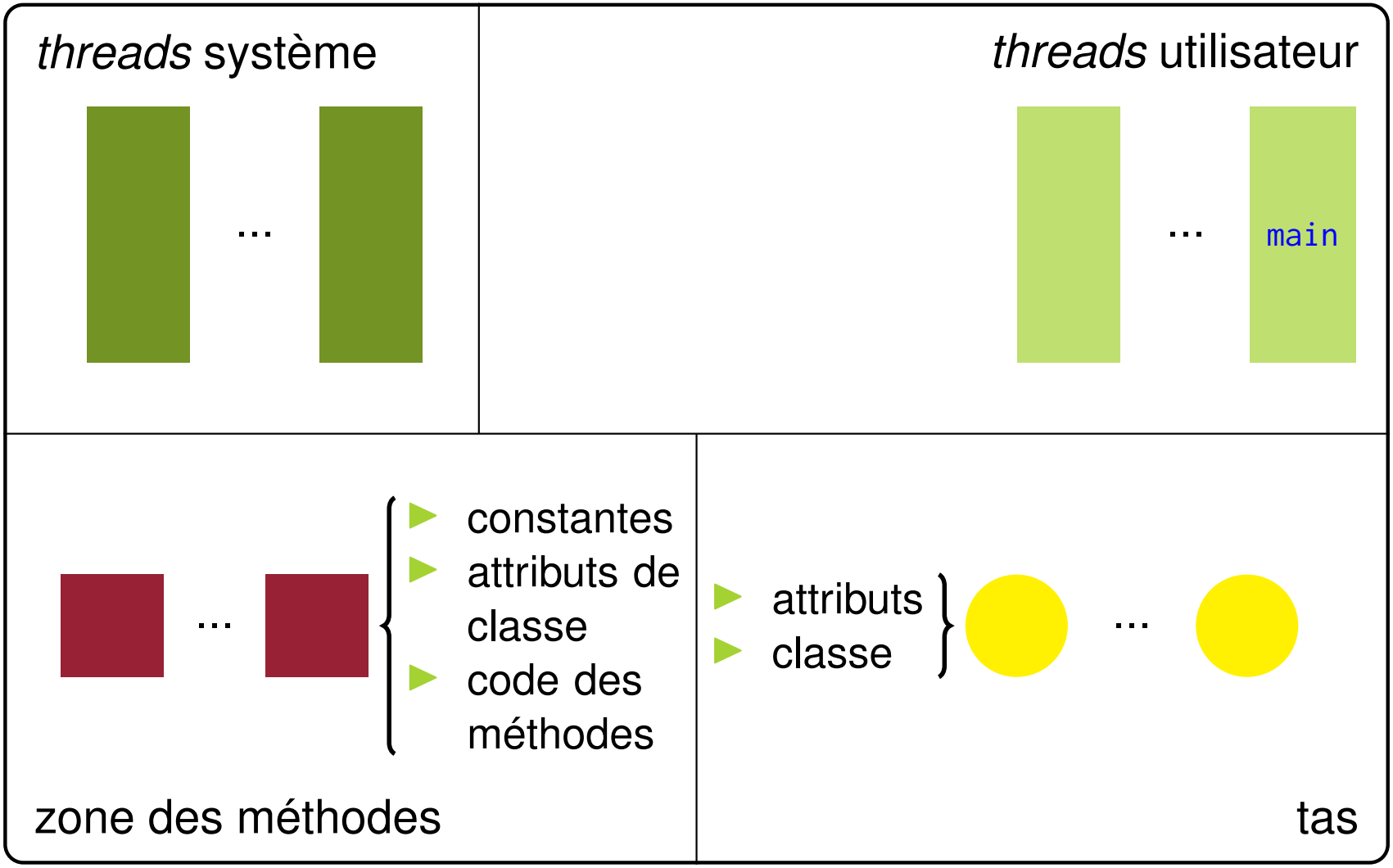
invoque la méthode `run` de l'objet tâche

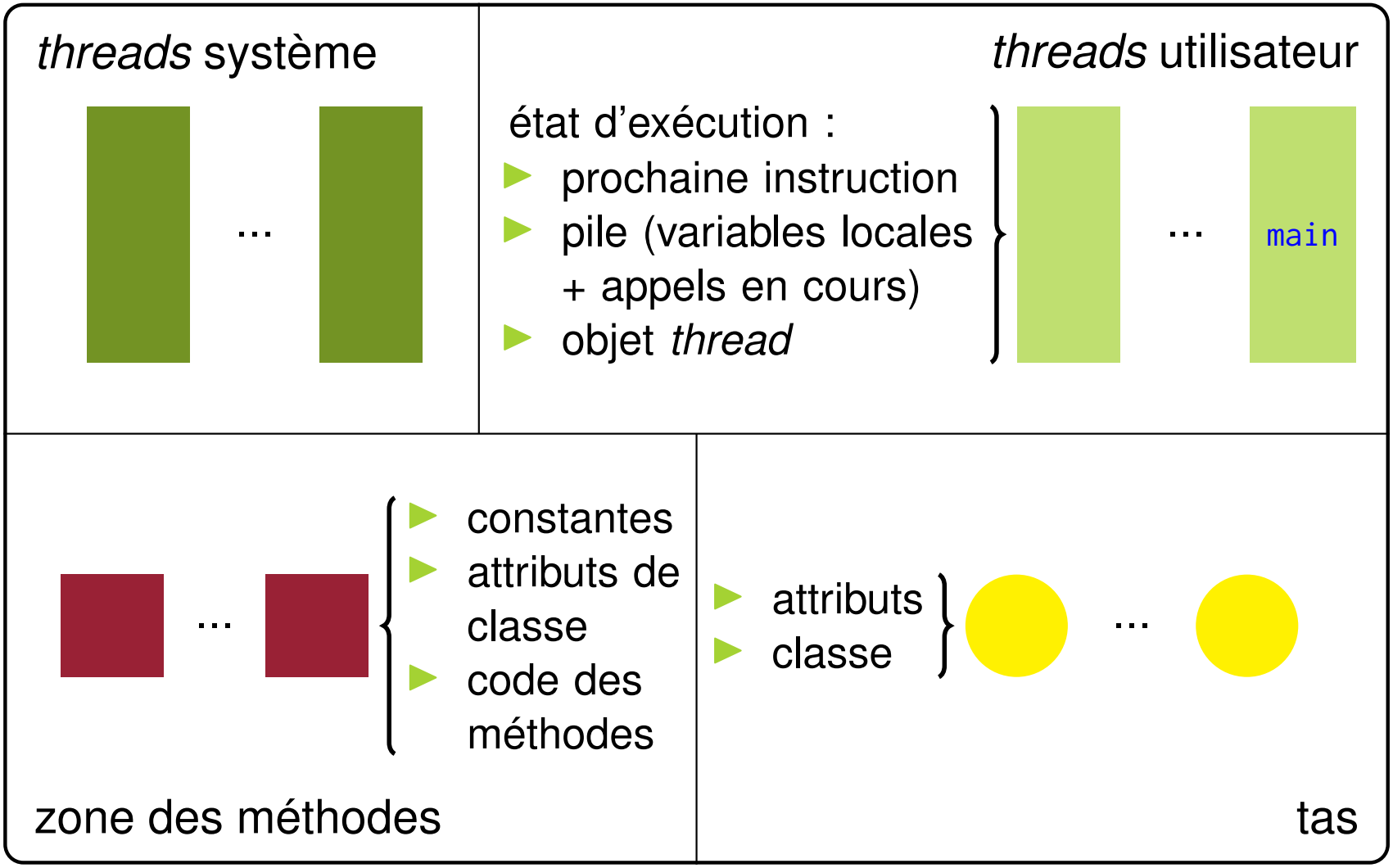
```
public class Thread implements Runnable {
  private Runnable target;
  public void run() {
    if (this.target != null) {
      this.target.run();
    }
  }
}
new Thread(new MyTask()).start();
```

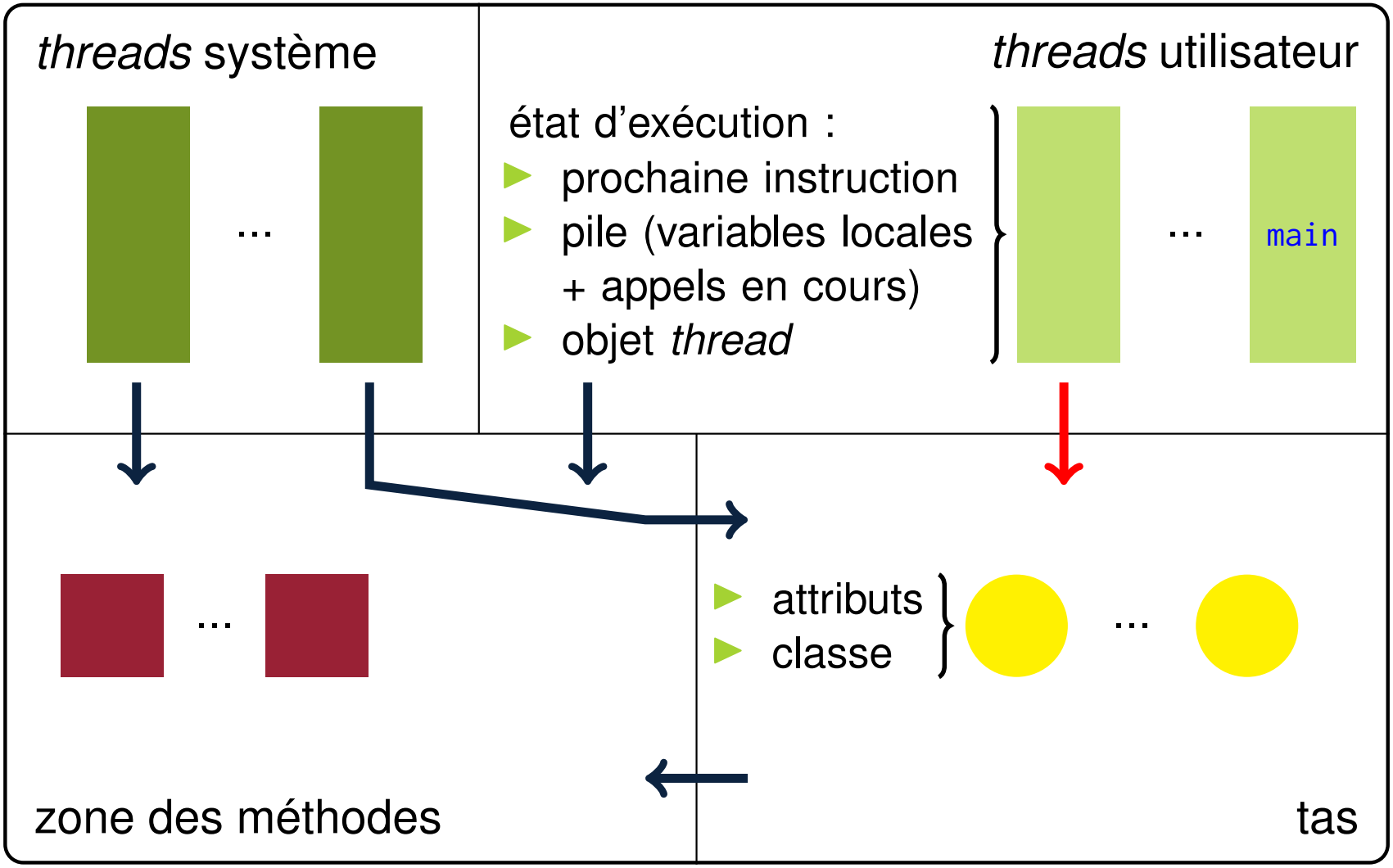
on parle de création par composition

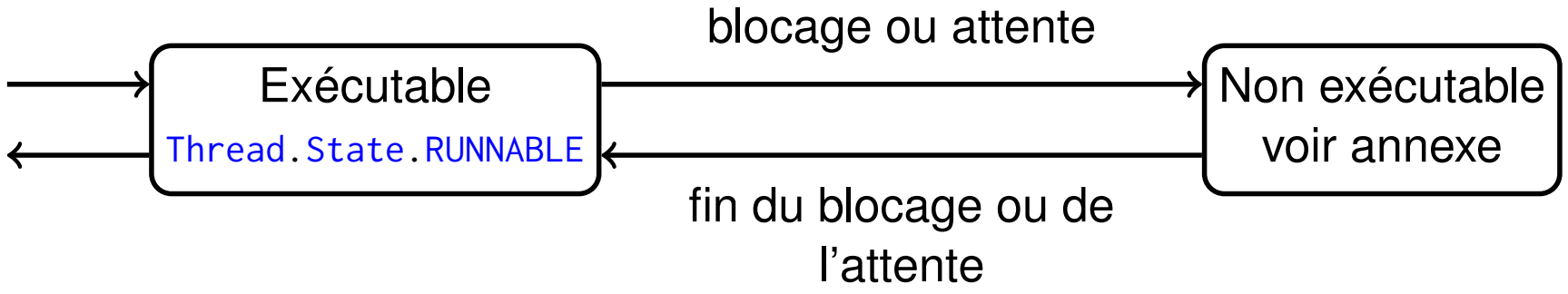

threads système
...
threads utilisateur
...
main
...
zone des méthodes
...
tas


threads système
...
threads utilisateur
...
main
...
attributs
classe
...
zone des méthodes
tas


threads système
...
threads utilisateur
...
main
...
constantes
attributs de classe
code des méthodes
zone des méthodes
attributs
classe
...
tas


threads système
...
threads utilisateur
état d'exécution :
prochaine instruction
pile (variables locales + appels en cours)
objet *thread*
...
main
constantes
attributs de classe
code des méthodes
...
zone des méthodes
attributs
classe
...
tas


threads système
threads utilisateur
état d'exécution :
prochaine instruction
pile (variables locales
+ appels en cours)
objet *thread*
...
main
attributs
classe
zone des méthodes
tas



- Si exécutables peuvent avoir du temps de calcul
- Choix du prochain à exécuter
  - le plus prioritaire et sinon au hasard
  - si un plus prioritaire arrive, on préempte celui qui s'exécute
- Priorité fixe une fois démarré
  - entre `MIN_PRIORITY` et `MAX_PRIORITY`
  - le *thread* `main` est de priorité `NORM_PRIORITY`
  - attention : *threads* égoïstes (calculs longs + forte priorité)
  - céder son tour à un *thread* de même priorité : `yield()`

- `stop`, `suspend` ou `resume` ? Non
- Car il faut rendre l'objet dans un état cohérent !
- Le code du *thread* doit avoir donc été prévu pour !
- `interrupt()` pour tenter d'interrompre un *thread*
  - si non exécutable, reactivé par `InterruptedException`
  - sinon, son *interrupted status* est positionné
  - on peut tester par `isInterrupted` (pas de remise à **`true`**) ou par `Thread.interrupted` (avec remise à **`true`**)
  - si interrompu et essaye de se bloquer, reçoit `InterruptedException`

1 Introduction

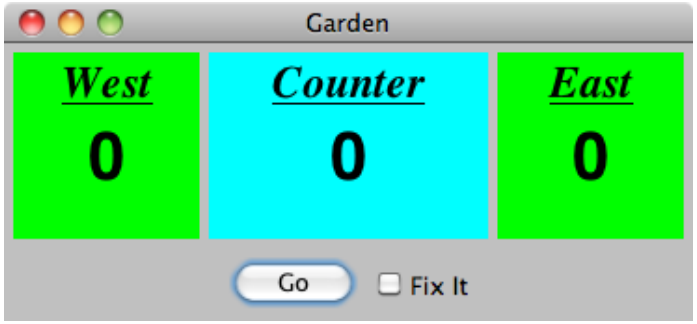
2 Les Threads Java

3 La difficulté de la concurrence

4 Exclusion mutuelle

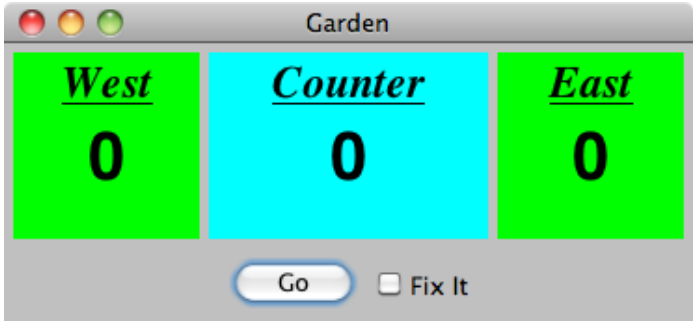
5 Conclusion

- Deux entrées : est et ouest
- Connaître le nombre de personnes dans le jardin
  - Modèle
    - un compteur global
    - 1 tâche, comptage local + global
    - 2 *threads*, un par entrée




```
1 public class Counter {
2   int value = 0;          NumberCanvas display;
3   void increment() {
4     int temp = this.value; // read[v]
5     this.value = temp + 1; // write[v+1]
6     this.display.setvalue(this.value);
7   }
8 }
```

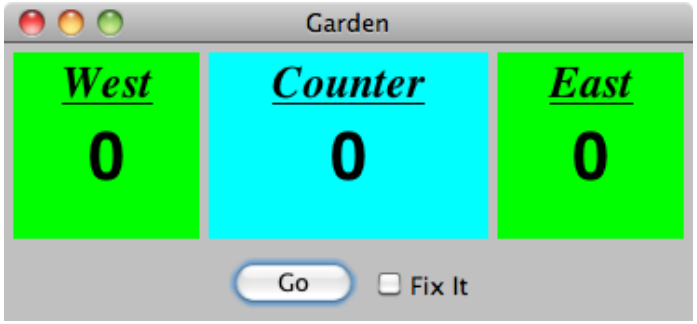
- Deux entrées : est et ouest
- Connaître le nombre de personnes dans le jardin
  - Modèle
    - un compteur global
    - 1 tâche, comptage local + global
    - 2 *threads*, un par entrée




```
1 public class Counter {
2   int value = 0;          NumberCanvas display;
3   void increment() {
4     int temp = this.value; // read[v]
5     this.value = temp + 1; // write[v+1]
6     this.display.setvalue(this.value);
7   }
8 }
```

Peut être interrompu

- Deux entrées : est et ouest
- Connaître le nombre de personnes dans le jardin
  - Modèle
    - un compteur global
    - 1 tâche, comptage local + global
    - 2 *threads*, un par entrée




```
1 public class Counter {
2   int value = 0;          NumberCanvas display;
3   void increment() {
4     int temp = this.value; // read[v]
5     this.value = temp + 1; // write[v+1]
6     this.display.setvalue(this.value);
7   }
8 }
```

Peut être interrompu

Attention vrai même si on utilise `this.value++` !

- Préemption ⇒ un *thread* peut être interrompu
- Uniquement entre deux instructions atomiques
  - atomique = instruction simple sur des données de 32 bits
  - ne sont pas atomiques : `double d = 1.2;` ou `i++;`
  - hormis à devenir expert, ne pas supposer
  - `java.util.concurrent.atomic` contient des classes qui fournissent des méthodes atomiques
- Impossible de prévoir l'ordre d'exécution des *threads*
- Les ressources partagées doivent être modifiées de façon cohérente
  - ici, il faut bien écrire la dernière valeur plus un

⇒ Les *threads* doivent s'entendre : la synchronisation

1 Introduction

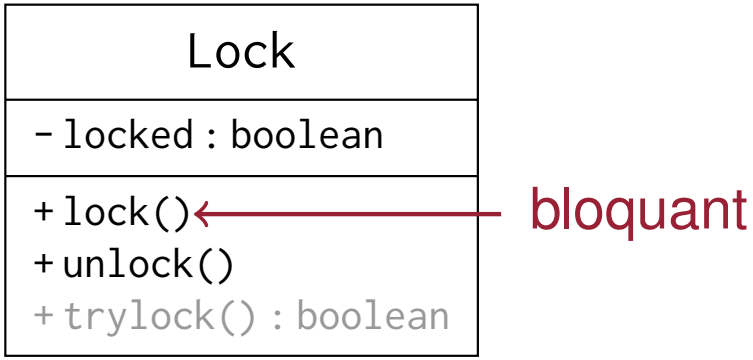
2 Les Threads Java

3 La difficulté de la concurrence

4 Exclusion mutuelle

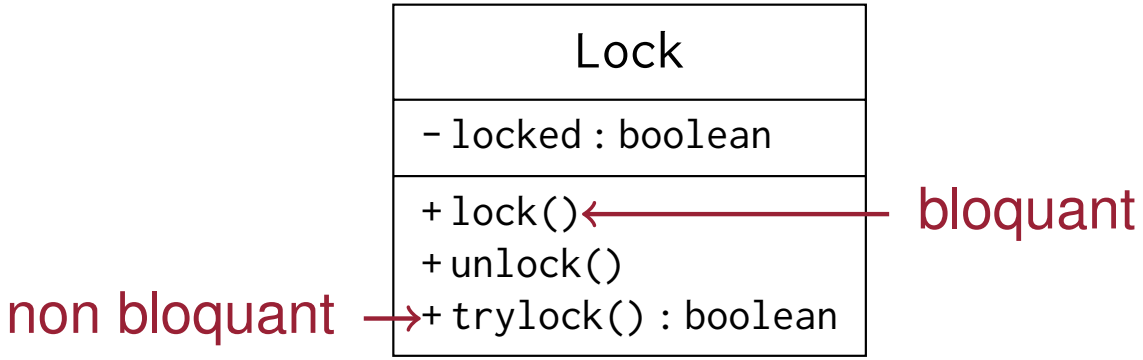
5 Conclusion

- Le principe
  - un *thread* indique aux autres qu'il veut accéder à la ressource partagée
  - si personne ne « détient » la ressource, elle lui est attribuée
    - le *thread* fait ses traitements et modifications (section critique)
    - il signale aux autres qu'il a fini (il relâche la ressource)
  - sinon, il attend son tour
- Différentes techniques
  - les verrous
  - les sémaphores (que l'on ne verra pas)
  - les moniteurs



- Usage classique

```
1 Lock l = ...;
2 l.lock();
3 try {
4   // section critique
5 } finally {
6   l.unlock();
7 }
```



- Usage classique

```
1 Lock l = ...;
2 l.lock();
3 try {
4   // section critique
5 } finally {
6   l.unlock();
7 }
```

- Attente active

```
1 Lock l = ...;
2 boolean hasLock = false;
3 while(!(hasLock = l.tryLock()))
4   // faire autre chose
5 if (hasLock)
6   try {
7     // section critique
8   } finally {
9     l.unlock();
10  }
```

1 Introduction

2 Les Threads Java

3 La difficulté de la concurrence

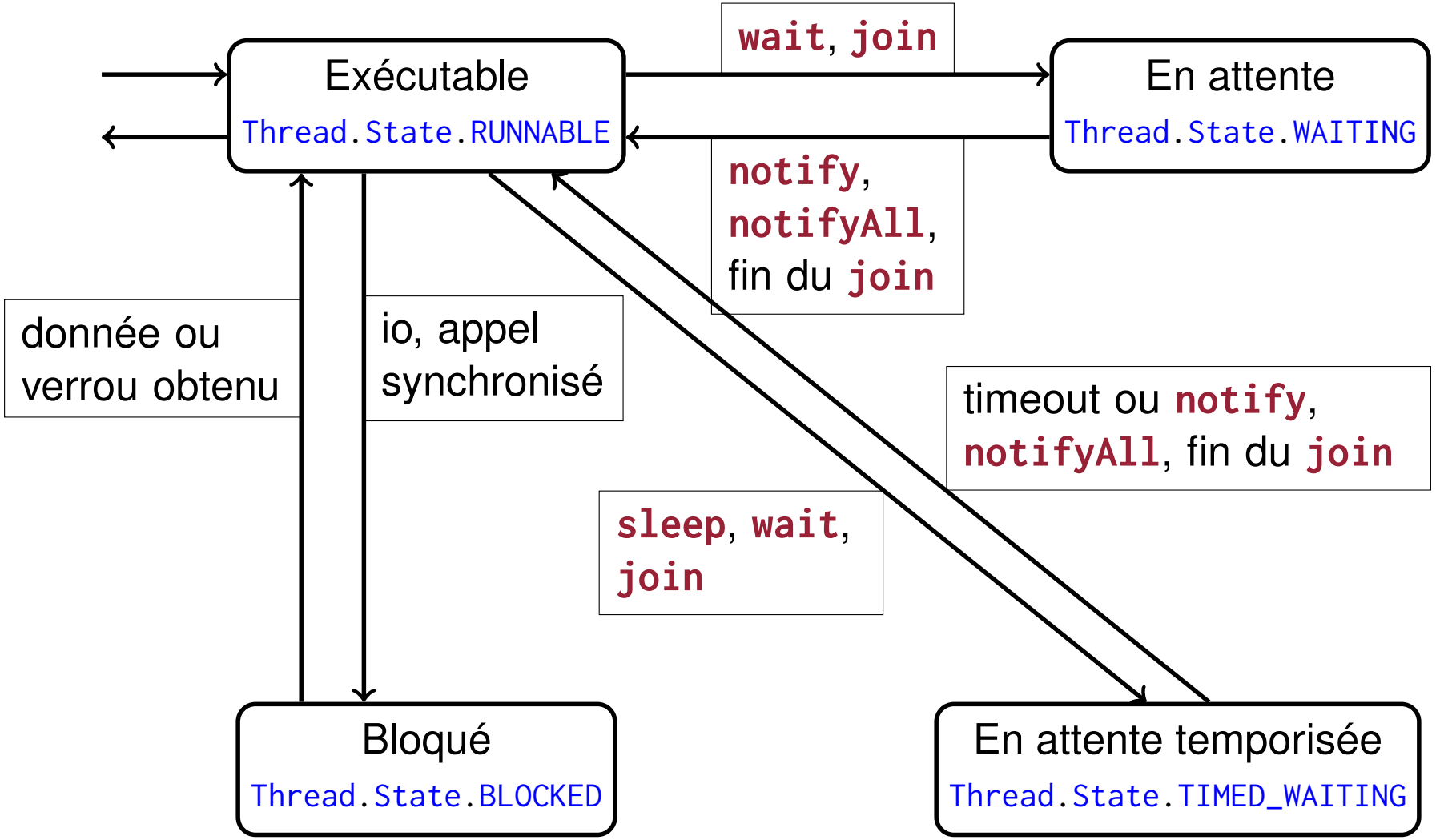
4 Exclusion mutuelle

5 Conclusion

- Java fournit directement des mécanismes pour gérer la programmation d'activité concurrente
  - *thread* concurrent
  - exclusion mutuelle
- Mise en pratique sur le tampon borné
- Le prochain cours portera sur la synchronisation de ces tâches
- Attention
  - même avec de tels mécanismes, la programmation concurrente reste complexe et facilemment incorrecte
  - il est nécessaire d'adopter une méthode claire, précise et rigoureuse (par ex, FSP)

Des transparents en plus...

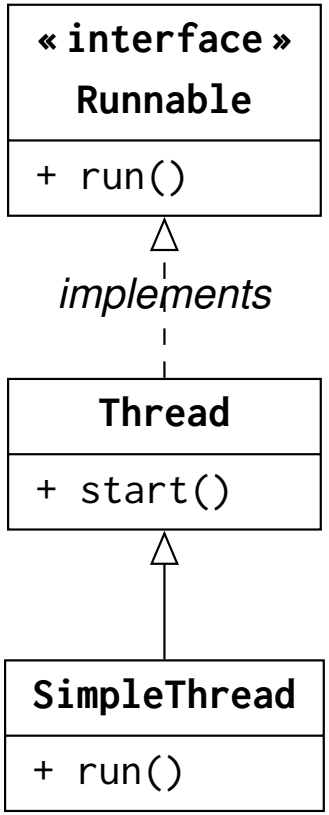
L'état Actif pour de vrai



Création d'un *thread* par héritage

- La tâche peut être définie par le *thread*
- On hérite de la classe Thread et on redéfinit le run

```
1  class SimpleThread extends Thread {
2    public void run() {
3      // Je m'endors 10 fois entre 0 et 1 seconde...
4      for (int i = 0; i < 10; i++) {
5        System.out.println(getName() + " : " + i);
6        sleep();
7      }
8      System.out.println(getName() + " a fini!");
9    }
10   public static void main(String[] args) {
11     for (int i = 0; i < 5; i++)
12       new SimpleThread("Thread " + i).start();
13   }
14 }
```



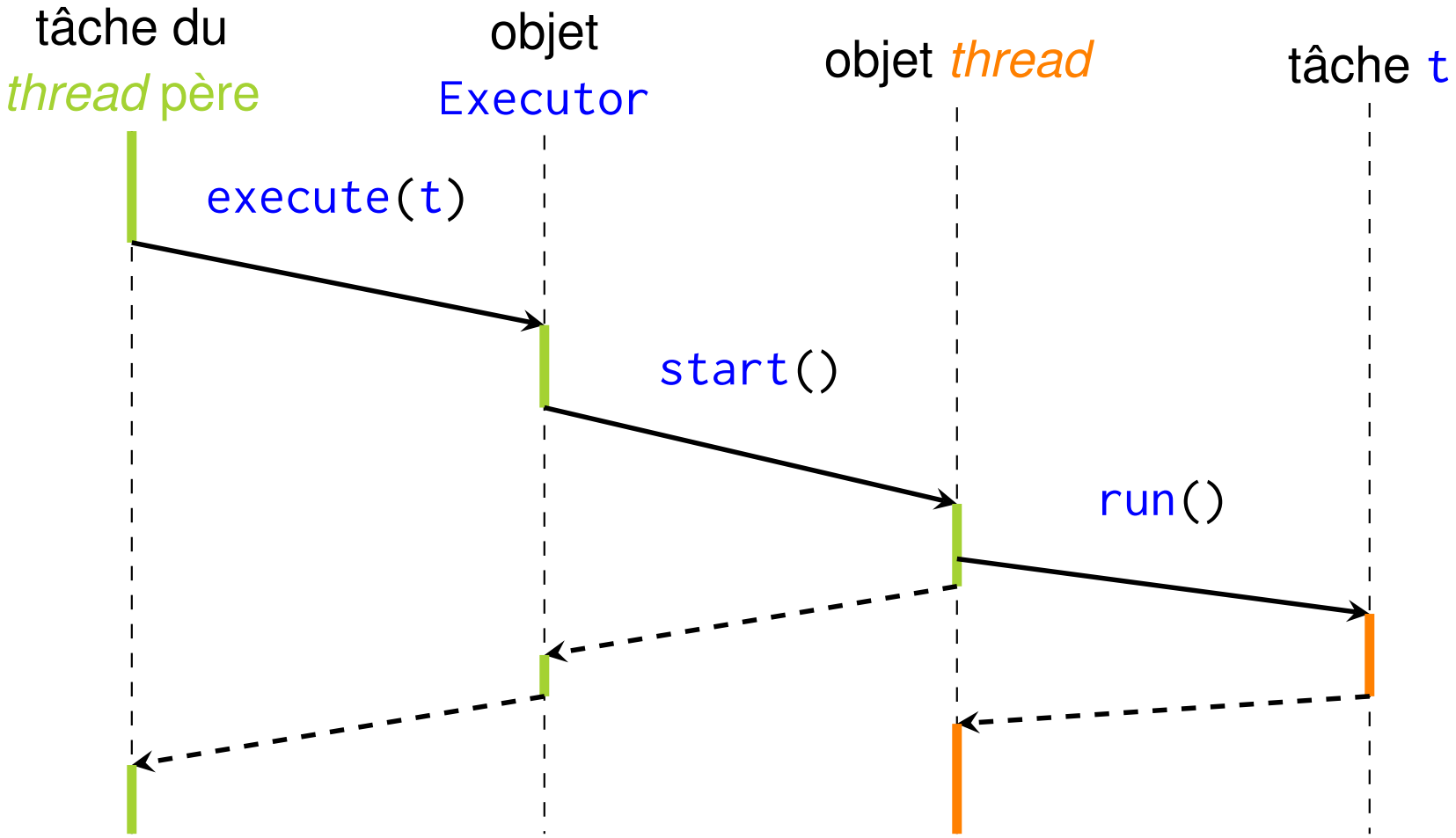
- Ne pas gérer les *threads*

⇒ Fabriques à *threads* : les exécuteurs
  - objet de type `java.util.concurrent.Executor`
  - a une méthode `execute(Runnable)`
- Fabrique à `Executor` dans la classe `Executors`
  - `Executor newSingleThreadExecutor()`
    - Utilise un seul *thread*, les tâches peuvent être mises en attente
  - `Executor newFixedThreadPool(int n)`
    - Idem avec au maximum n *threads* qui sont réutilisés
  - `Executor newCachedThreadPool()`
    - Idem mais crée autant de *threads* que nécessaire qui sont réutilisés

Un exemple


```java
class SimpleExecutor implements Runnable {
  private final String name;
  public SimpleExecutor(String str) {
    this.name = str;
  }
  public void run() {
    // Même corps que SimpleThread
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 6; i++)
      exec.execute(new SimpleExecutor("T" + i));
  }
}
```

- Avantage, pas de gestion des *threads*
- Changer de politique = changer de fabrique


tâche du
thread père
objet
Executor
objet thread
tâche t
execute(t)
start()
run()