Parallélisme en Java

Patrice Torguet

Plan

- **Introduction**
- Notion de processus
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Introduction

- Une application est **multitâche** quand elle exécute (ou peut exécuter) plusieurs parties de son code en même temps
- A un instant donné, l'application comporte plusieurs **points d'exécution** liés aux différentes parties qui s'exécutent en parallèle.
- Tous les systèmes d'exploitation modernes sont multitâches et permettent l'exécution d'applications multitâches
- Sur une machine monoprocesseur/mono-coeur cette exécution en parallèle est simulée
- Le multitâche s'appuie sur les processus ou les threads (fils d'exécution en français)
- Si le système est **préemptif**, il peut à tout moment suspendre un processus/thread pour en exécuter un autre
- Sinon, les processus/threads doivent indiquer explicitement ou implicitement (par exemple quand ils sont bloqués par des entrées/sorties) qu'ils veulent passer la main à un autre processus/thread

Introduction

- Exemple de multitâche à l'intérieur d'une application :
  - l'interface graphique peut lancer un thread pour charger une image pendant qu'elle continue de traiter les événements générés par des actions de l'utilisateur
  - une application serveur qui attend les demandes de connexions venant des clients peut lancer un processus/thread pour traiter les demandes de plusieurs clients simultanément
  - la multiplication de 2 matrices (m,p) et (p,n) peut être effectuée en parallèle par m * n threads
- Utilité du multitâche :
  - amélioration des performances en répartissant les différentes tâches sur différents processeurs
  - profiter des temps de pause d'une tâche (attente d'entrées/sorties ou d'une action utilisateur) pour faire autre chose
  - réagir plus vite aux actions de l'utilisateur en rejetant une tâche longue et non-interactive dans un autre thread (exemple : correction de l'orthographe pour un éditeur de texte)

Introduction

- Problèmes du multitâche :
  - il est souvent plus difficile d'écrire un programme multitâche
  - et surtout, il est plus difficile de déboguer un programme qui utilise le multitâche
- Java et le multitâche
  - Java gère aussi bien les processus que les threads
  - Les threads sont plus utilisés car bien intégrés au langage et moins gourmands en ressources mémoires

Plan

- Introduction
- **Notion de processus**
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Notion de Processus

- Chaque processus a un espace d'adressage (espace mémoire où sont rangées les variables et les objets) distinct
- La communication entre processus peut être réalisée par :
  - des signaux
  - des tubes (pipes)
  - de la mémoire partagée
  - des sockets
  - ...

Notion de Processus

- Classe Runtime
  - Permet de contrôler le processus courant : Runtime.getRuntime()
  - Méthodes permettant de connaître l'environnement d'exécution : total/free/maxMemory(), availableProcessors()
  - Contrôle du processus : gc(), exit(), halt()
  - Création d'autres processus : exec()
- Classe Process
  - Permet de contrôler un processus fils
  - Process fils = Runtime.getRuntime().exec(″commande″);
  - Méthodes :
    - waitFor() : attends la fin de l'exécution du fils
    - exitValue() : valeur renvoyée lors de la fin du fils (par exit(n))
    - destroy() : tue le fils
    - getInputStream(), getOutputStream(), getErrorStream() permettent de contrôler les E/S standard du processus

Notion de Processus

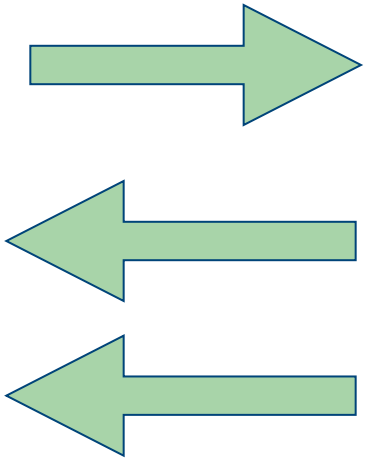
Père

Process fils = Runtime.getRuntime().exec("...");

OutputStream outFils = fils.getOutputStream();

InputStream inFils = fils.getInputStream();

InputStream errFils = fils.getErrorStream();



Fils

stdin
System.in

stdout
System.out

stderr
System.err

Notion de Processus

- Il existe plusieurs méthodes exec qui permettent de préciser :
  - L'application à démarrer
  - Des paramètres
  - Des variables d'environnement
  - Le répertoire courant
- Exemples :
  - Runtime.getRuntime().exec(″javac Toto.java″);
  - Runtime.getRuntime().exec(new String[]{″javac″,″Toto.java″});
    // utilise un tableau pour la commande et ces arguments
  - Runtime.getRuntime().exec(″prog″, new String[] {″var=val″},
    new File(″/tmp/″));
    // exécute prog avec pour répertoire courant /tmp en ayant
    // positionné auparavant la variable d'environnement var à la
    // valeur val

Plan

- Introduction
- Notion de processus
- **Notion de thread**
- Créations de threads
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Notion de Thread

- Un thread peut se traduire par tâche ou fil d'exécution
  - Attention, Thread est différent du terme tâche dans «système multitâches»
- Permet de faire fonctionner plusieurs portions de code simultanément dans le même processus => partagent le même espace mémoire
- Intérêt :
  - Correcteur grammatical d'un traitement de texte
  - Accélération de calculs matriciels sur un multi-processeur/coeur
  - ...
  - Permet de traiter plusieurs connexions client/serveur simultanément
- L'environnement Java est multi-threads :
  - Création de threads
  - Synchronisation des threads qui partagent des données.

Plan

- Introduction
- Notion de processus
- Notion de thread
- **Créations de threads**
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Création de Threads

- A tout thread Java sont associés
  - Un objet qui détermine le code qui est exécuté par le thread
  - Un objet qui «contrôle» le thread et le représente auprès des objets de l'application ; on l'appellera le «**contrôleur de thread**»

Interface Runnable

- La classe de l'objet qui détermine le code à exécuter doit implémenter l'interface Runnable

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

méthode qui détermine le code à exécuter par le thread

Un thread n’est pas un objet !

- La méthode run() «saute» d’un objet à l’autre en exécutant les méthodes des classes de ces objets :
  o1.m1();
  o2.m2();
  ...
- Un thread est une unité d’exécution qui, à un moment donné, exécute une méthode
- A un autre moment, ce même thread pourra exécuter une autre méthode d’une autre classe

Contrôleur de thread

- Le contrôleur d'un thread est une instance de la classe **Thread** (ou d'une classe dérivée de Thread) qui :
  - est l'intercesseur entre le thread et les objets de l'application
  - permet de contrôler l'exécution du thread (pour le démarrer en particulier)
  - a des informations sur l'état du thread (son nom, sa priorité, s'il est en vie ou non,...)
- La classe Thread implémente l'interface Runnable mais sa méthode run() est vide
  - Si on dérive de Thread et que l'on définit une nouvelle méthode run() (non vide) on peut donc tout gérer avec un seul objet (contrôle le thread et définit le code à exécuter)

Création de Threads : 2 façons

- 1ère façon : créer une instance d'une classe fille de Thread qui redéfinit la méthode run()

- 2ème façon : utiliser le constructeur Thread(Runnable r) de la classe Thread :
  - créer un Runnable (pour le code qui sera exécuté par le thread)
  - le passer au constructeur de Thread

1ère façon

```
class ThreadTache extends Thread {
  ...
  public void run() {
    // Code qui sera exécuté par le thread
    ...
  }
}
```

```
ThreadTache threadTache = new ThreadTache(...);
```

2ème façon

```
class Tache implements Runnable {
  ...
  public void run() {
    // Code qui sera exécuté par le thread
    ...
  }
}
```

```
Tache tache = new Tache(...);
Thread t = new Thread(tache);
```

Comment choisir ?

- Si on veut hériter d'une autre classe pour la classe qui contient la méthode run(), on est obligé de choisir la 2ème façon (Thread(Runnable r))
- Il est aussi plus simple d'utiliser la 2ème façon pour partager des données entre plusieurs threads exécutant le même code
- Sinon, l'écriture du code est (légèrement) plus simple en utilisant la 1ère façon

Nom d’un thread

- Des constructeurs de la classe Thread permettent de donner un nom au thread lors de sa création
- Le nom va faciliter le repérage des threads durant la mise au point

Lancer l'exécution d'un thread

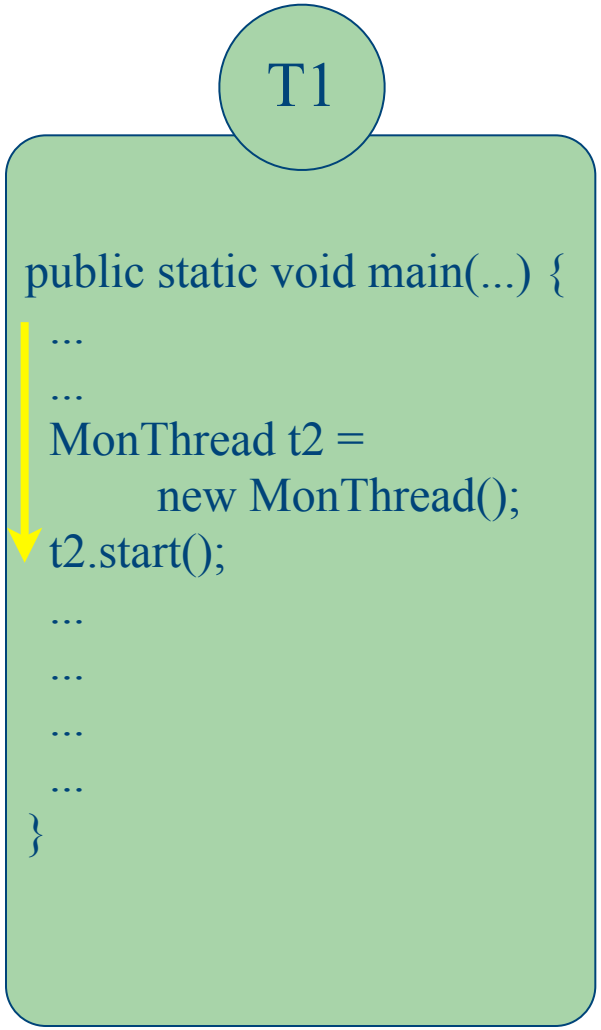
- On appelle la méthode start() du contrôleur de thread :
  - t.start();
- Le code du Runnable s'exécute **en parallèle** du code qui a lancé le thread
- **Attention**, une erreur serait d'appeler directement la méthode run() : la méthode serait alors exécutée par le thread qui l'a appelée et pas par un nouveau thread !
- On ne peut relancer un thread qui a déjà été lancé
  - si l'exécution n'est pas encore terminée on obtient une `java.lang.IllegalThreadStateException`
  - sinon, aucune exception n'est lancée mais rien n'est exécuté
  - Il faudra donc créer un nouveau contrôleur de thread travaillant sur le Runnable et refaire un start()

Création et lancement de Thread

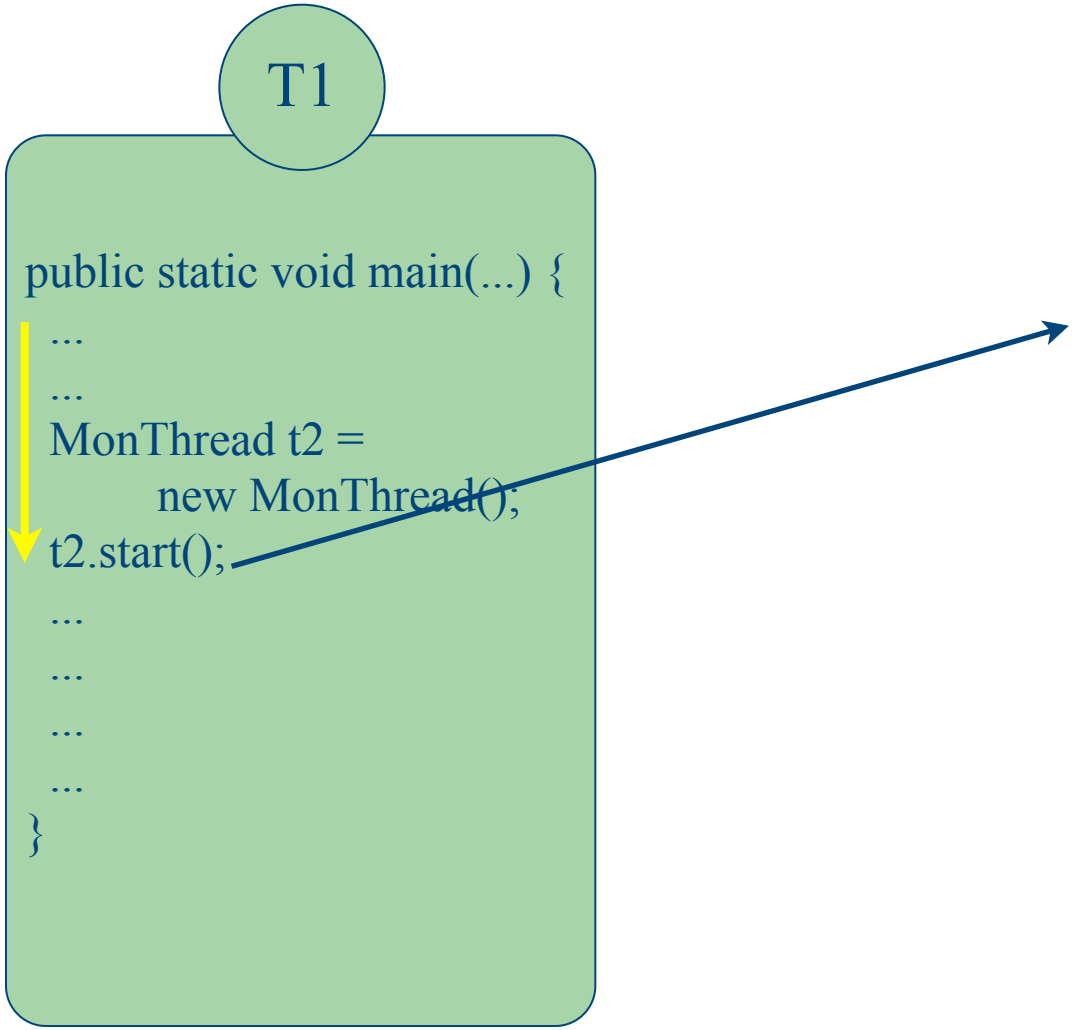
T1

```
public static void main(...) {
  ...
  ...
  MonThread t2 =
          new MonThread();
  t2.start();
  ...
  ...
  ...
  ...
}
```

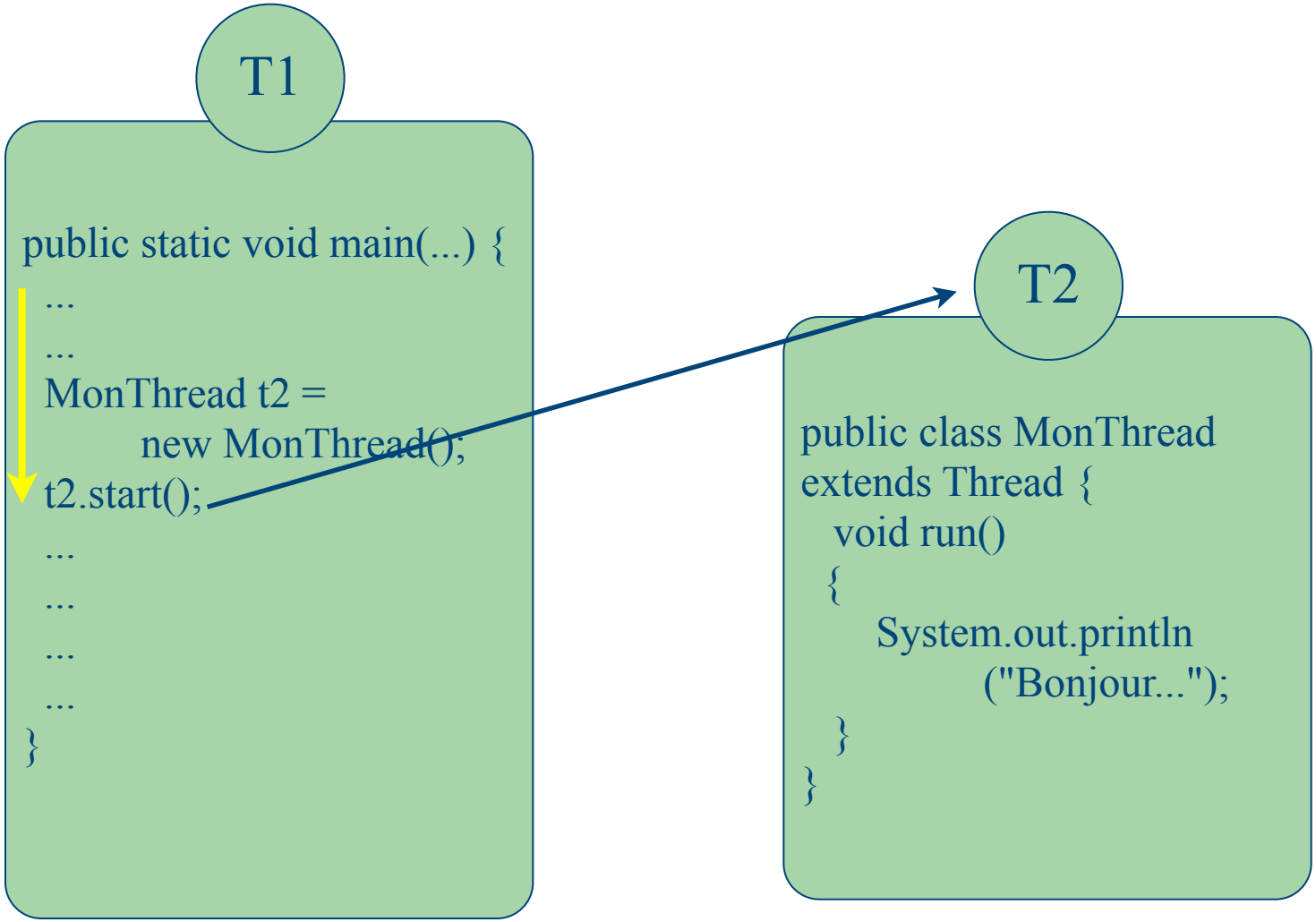
Création et lancement de Thread



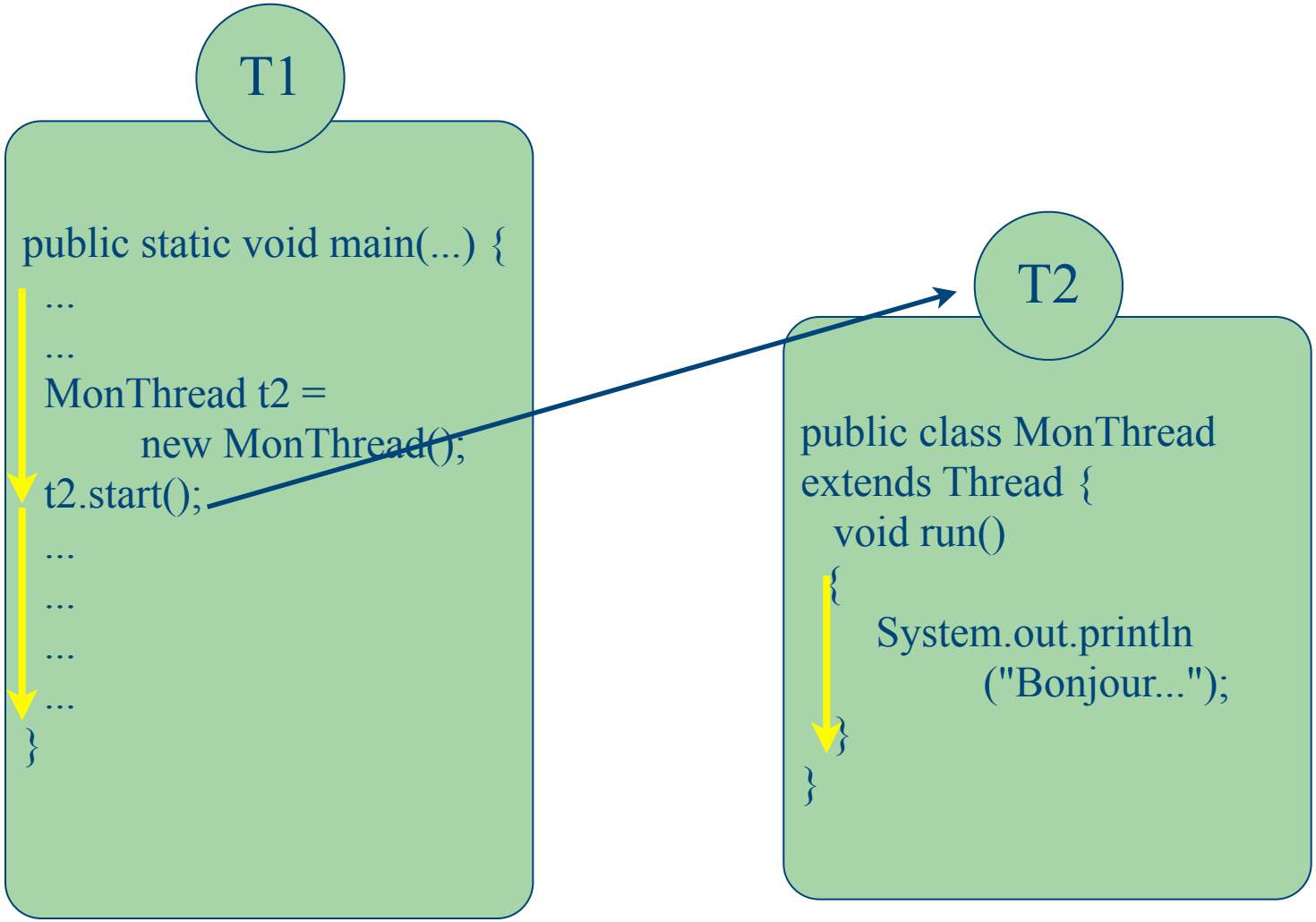
Création et lancement de Thread



Création et lancement de Thread



Création et lancement de Thread



Vie du contrôleur de thread

- Le contrôleur de thread existe indépendamment du thread,
  - avant le démarrage du thread, par exemple, pour initialiser des variables d'instances du contrôleur (priorité par exemple)
  - après la fin de l'exécution de ce thread, par exemple, pour récupérer des valeurs calculées pendant l'exécution du thread et rangées dans des variables d'instances du contrôleur

Utilisation d’une classe interne

- La méthode run() est publique
- Si on ne souhaite pas qu’elle soit appelée directement, on peut utiliser une classe interne à une classe fille de Thread pour implémenter Runnable

Utilisation d'une classe interne anonyme

- Si le code d'une tâche comporte peu de lignes (sinon ça devient vite illisible), on peut lancer son exécution en parallèle en utilisant une classe anonyme :

```
Thread t = new Thread() {
   ...
   public void run()
     ...
   }
};
t.start();
```

Ou encore :

```
new Thread(
   new Runnable() {
      ...
      public void run() {
         ...
      }
   });
```

Méthodes publiques principales de la classe Thread

```
void start()
static void sleep(long)
  throws InterruptedException
void join() throws InterruptedException
void interrupt()
static boolean interrupted()
int getPriority()
void setPriority(int)
static Thread currentThread()
static void yield()
```

Thread courant

- La méthode currentThread montre bien qu'un thread n'est pas un objet
- Placée dans une méthode de n'importe quelle classe, elle retourne l'objet Thread qui contrôle le thread qui exécute cette méthode au moment où currentThread est appelé
- On peut ainsi faire un traitement spécial dans le cas où la méthode est exécutée par un certain thread (par exemple le thread de répartition des événements dans une GUI)

Attente de la fin d'un thread

- Soit un contrôleur de thread **t**

  t.join() ;

  bloque le thread COURANT jusqu'à la fin de l'exécution du thread contrôlé par **t**

- On remarquera qu'après la fin de l'exécution du **thread** de **t** on peut encore appeler des méthodes de **l'objet** contrôleur de thread **t**

- On peut aussi interroger la tâche exécutée par le thread pour récupérer le résultat d'un calcul effectué par le thread

Passer la main

- La méthode statique de la classe Thread

  ```
  public static void yeld()
  ```

  permet, pour le thread COURANT, de passer la main à un autre thread de priorité égale ou supérieure

- Elle permet d'écrire des programmes plus portables qui s'adaptent mieux aux systèmes multitâches non préemptifs (il n'en existe plus aujourd'hui cela dit)

- Elle permet aussi de mettre au point certaines parties d'une application multitâche pour forcer les changements de threads (voir la partie sur la synchronisation des threads)

Dormir

- La méthode statique de la classe Thread

```
public static void sleep(long x)
     throws InterruptedException
```

  permet d'endormir le thread COURANT pendant x millisecondes

- Attention : si elle est exécutée dans du code synchronisé (voir plus loin), le thread garde le moniteur (au contraire de wait())

Interruption de thread

- `t.interrupt()`

  demande au thread contrôlé par t d'interrompre son exécution

- Cette méthode n'interrompt pas brutalement le thread mais positionne son état «interrupted»

- La méthode statique
  `static boolean interrupted()`
  renvoie la valeur de l'état «interrupted» du thread courant (attention, après l'exécution de cette méthode, l'état «interrupted» du thread est mis à faux)

Interruption de thread

- Pour qu'un thread interrompe vraiment son exécution, il doit participer activement à sa propre interruption
- 2 cas :
  - le thread est bloqué, par les méthodes sleep, wait, join, ou en attente d'une entrée/sortie interruptible
    - L'appel de interrupt provoque la levée d'une InterruptedException (ou ClosedByInterruptException pour les E/S) dans le thread en attente
    - Le thread «interrompu» doit gérer cette interruption
  - toutes les autres situations...
    - Le thread doit vérifier (périodiquement) s'il a été interrompu en appelant la méthode statique interrupted()

Interruption de thread

- Exemple pour le premier cas : ici on repositionne l'état «interrupted» pour pouvoir le tester plus loin

```
try {
   ...
catch(InterruptedException e) {
   ...
   Thread.currentThread().interrupt();
}
```

- Exemple pour le second cas :

```
while (! Thread.interrupted()) {
  ... // faire son travail
}
```

Interruption de thread

- Réaction possibles à une interruption :
  - Interrompre l'exécution avec un return ou en lançant une exception
  - Effectuer un traitement particulier en réaction à l'interruption puis reprendre l'exécution
  - Ignorer l'interruption

Exemple 1

```
for (int i = 0; i < 100; i++) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException e) {
      // S'interrompt après avoir fait
      // le ménage
      ...
      return;
  }
  // Effectue un traitement
  traitement();
}
```

Exemple 2

```
for (int i = 0; i < 100; i++) {
  // Effectue un traitement lourd
  traitement();
  if (Thread.interrupted()) {
    // S'interrompt après avoir fait
    // le ménage
    ...
    return;
  }
}
```

Threads et exceptions

- Si une exception n'est pas traitée (par un bloc try-catch) elle interrompt l'exécution du thread courant mais pas des autres threads

- La méthode run ne peut déclarer lancer une exception («throws)

- Une exception non saisie («catch») peut être saisie par le groupe du thread ou une méthode par défaut (voir plus loin)

Plan

- Introduction
- Notion de processus
- Notion de thread
- Créations de threads
- **Synchronisation entre threads**
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Synchronisation entre Threads

- L'utilisation de threads peut entraîner des besoins de synchronisation pour éviter les problèmes liés aux accès simultanés aux variables

- 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

Exemple de problème

- Si x = 2, le code x = x + 1;
  exécuté par 2 threads, peut donner en fin d'exécution 3 ou 4 suivant l'ordre d'exécution :

1. T1 : lit la valeur de x (2)
2. T2 : lit la valeur de x (2)
3. T1 : calcule x + 1 (3)
4. T2 : calcule x + 1 (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 !

Nécessité de synchroniser

- 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**

2 possibilités pour synchroniser du code sur un objet o

- Déclarer une méthode synchronisée m (synchronisation lors de l'appel o.m()) :

```
public synchronized int m(...) { . . . }
```

- Utilisation d'un bloc synchronisé sur l'objet o :

```
synchronized(o) {
  // le code synchronisé
  ...
}
```

Synchronized

T1

```
... void main(...) {
  Object o = new Object();
  for(...) {
     MonThread t =
       new MonThread(o);
     t.start(...);
  }
  ...
}
```

Synchronized

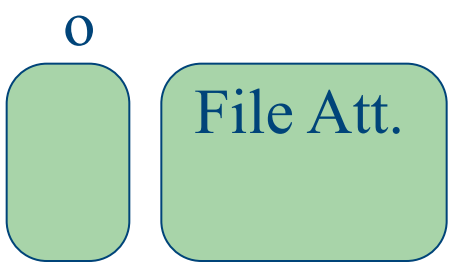
T1

```
... void main(...) {
  Object o = new Object();
  for(...) {
     MonThread t =
       new MonThread(o);
     t.start(...);
  }
  ...
}
```

Synchronized

T1

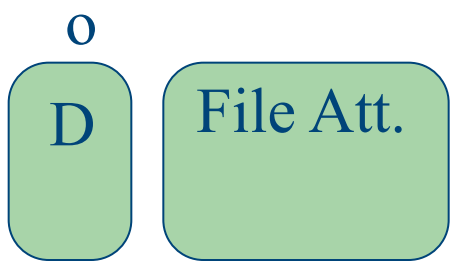
```
... void main(...) {
  Object o = new Object();
  for(...) {
     MonThread t =
       new MonThread(o);
     t.start(...);
  }
  ...
}
```



Synchronized

T1

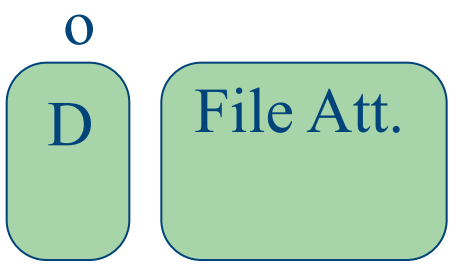
```
... void main(...) {
  Object o = new Object();
  for(...) {
     MonThread t =
       new MonThread(o);
     t.start(...);
  }
  ...
}
```



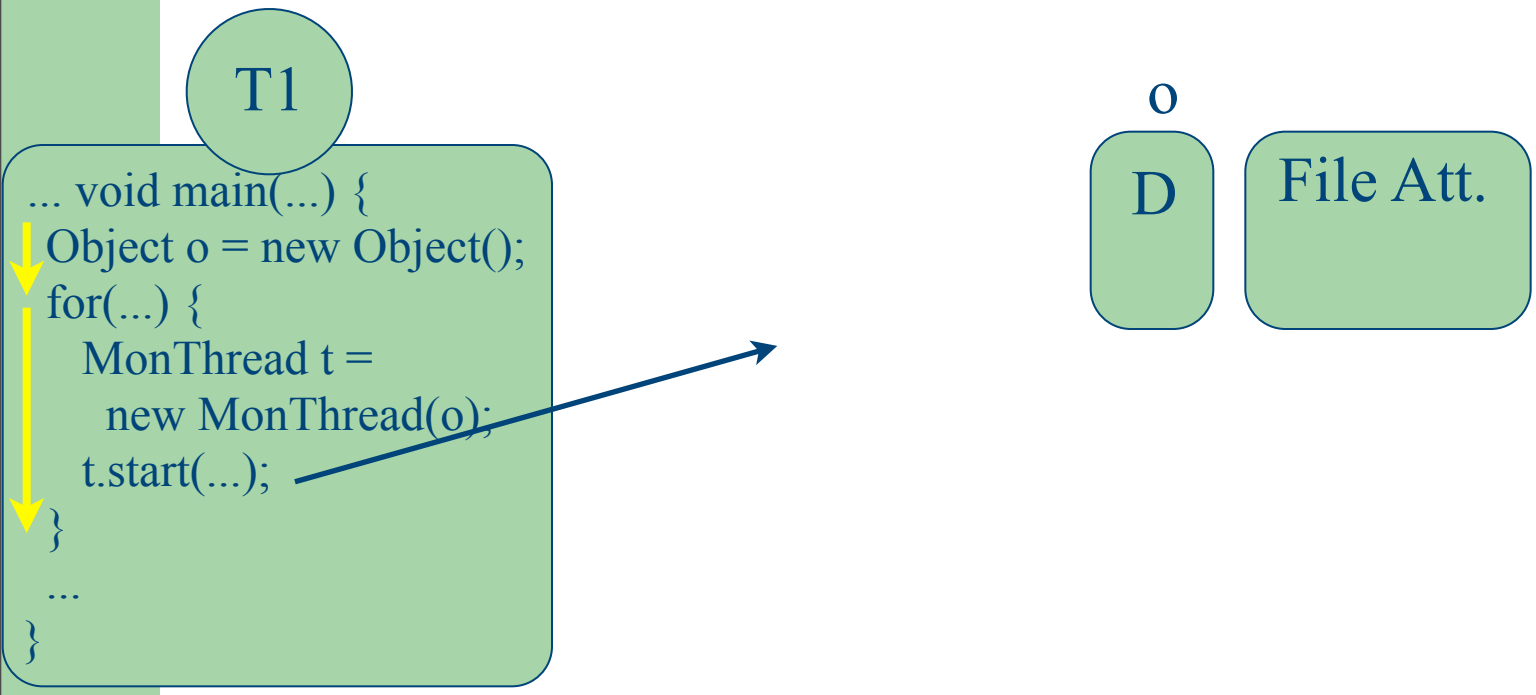
Synchronized

T1

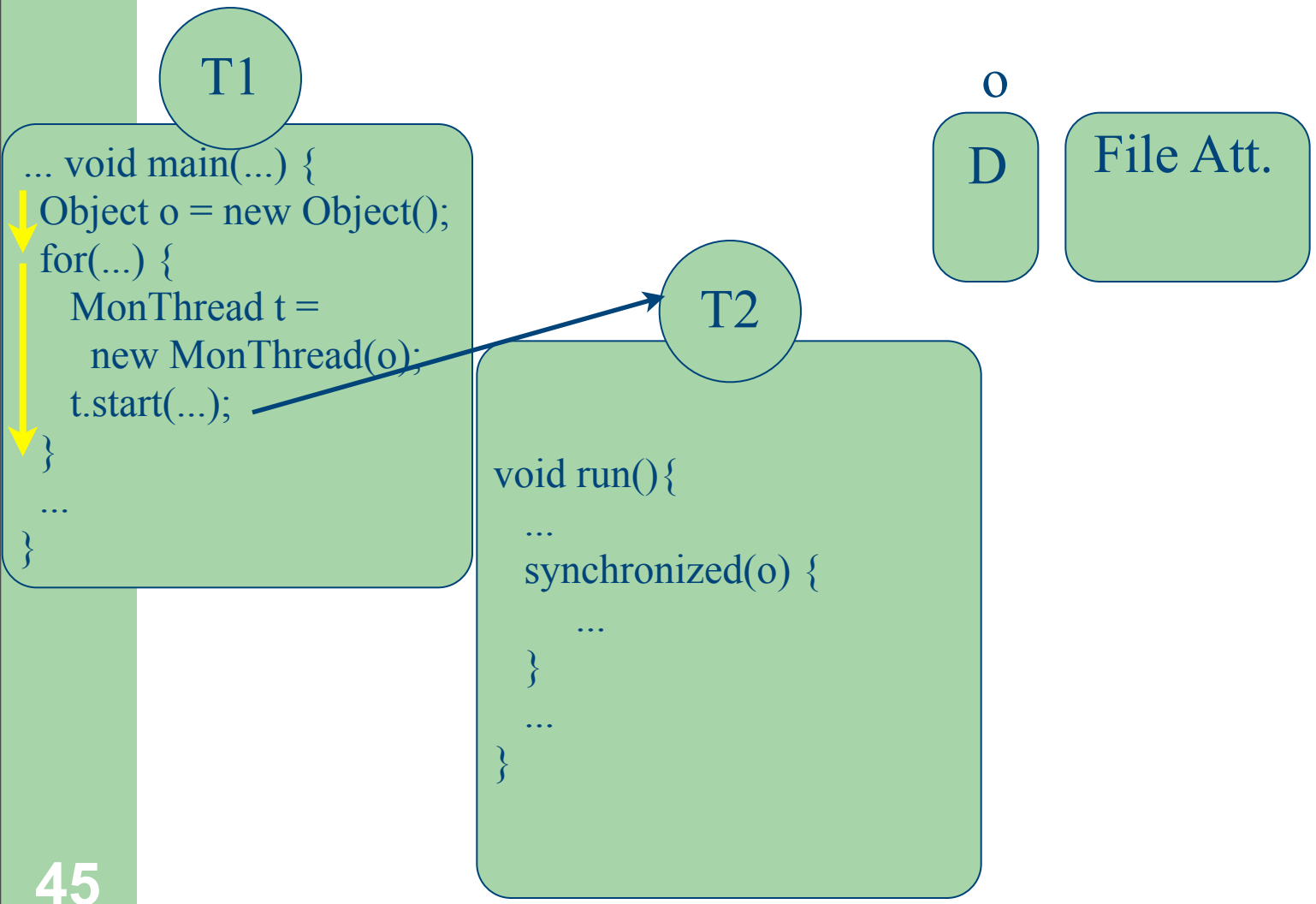
```
... void main(...) {
  Object o = new Object();
  for(...) {
     MonThread t =
       new MonThread(o);
     t.start(...);
  }
  ...
}
```



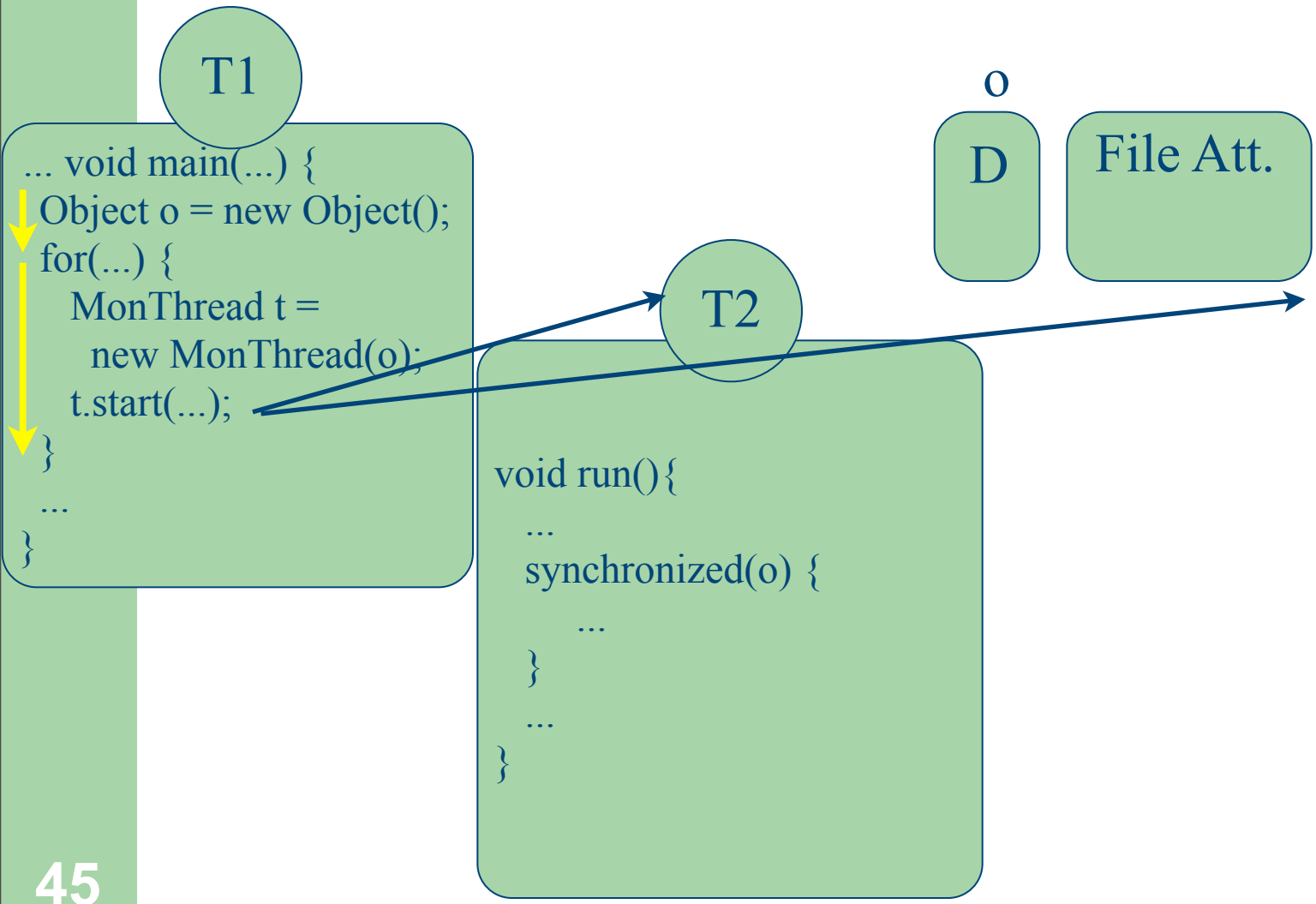
Synchronized



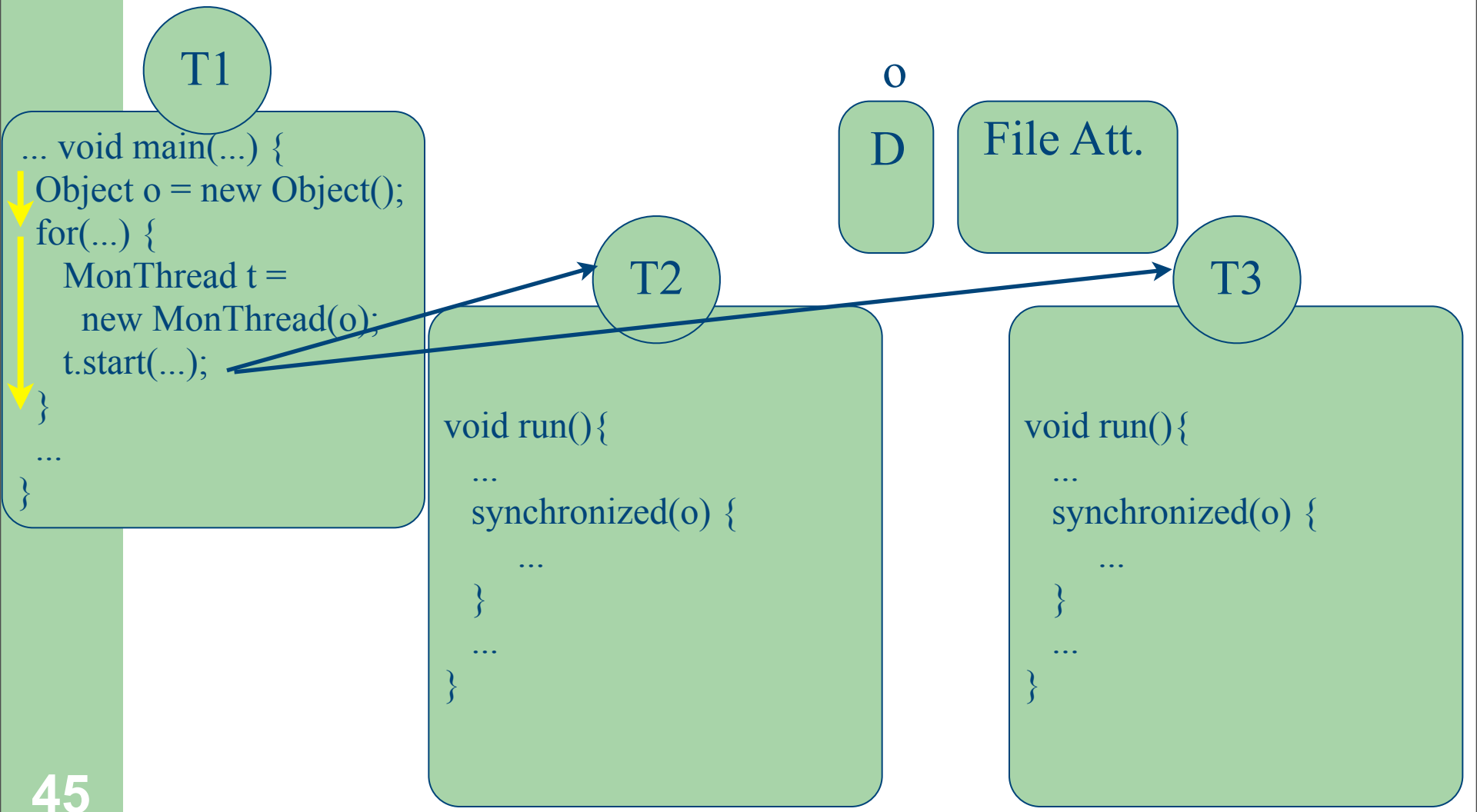
Synchronized



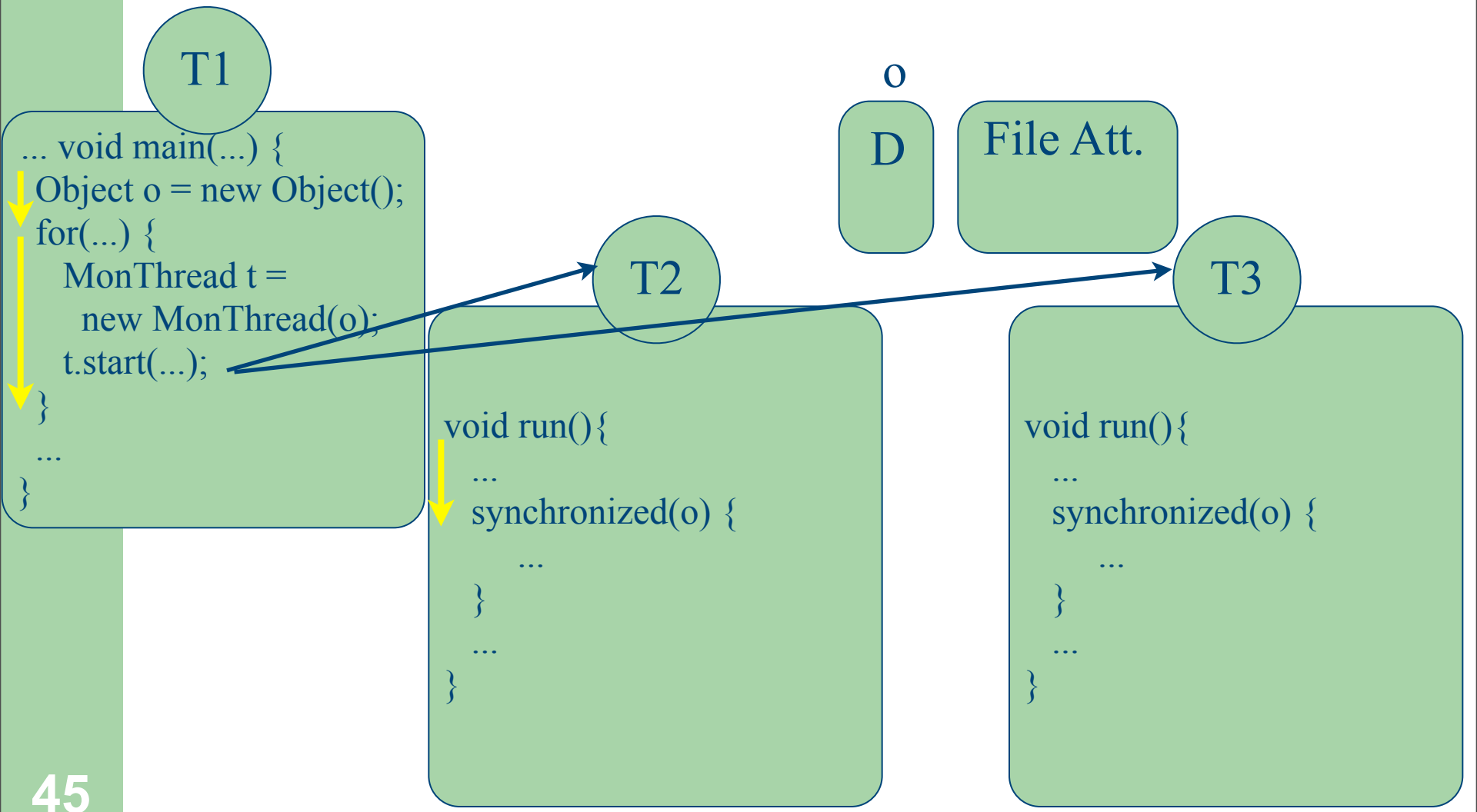
Synchronized



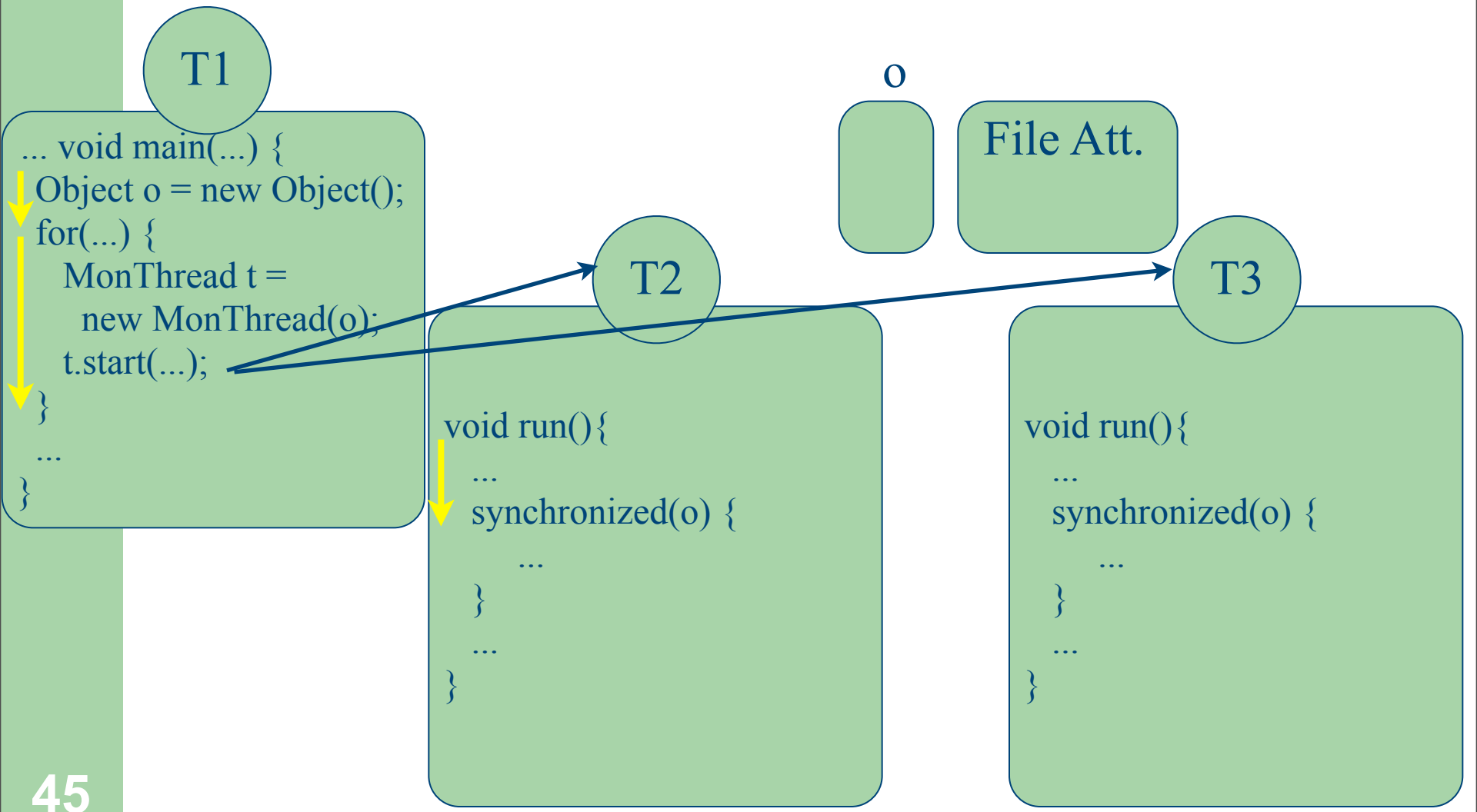
Synchronized



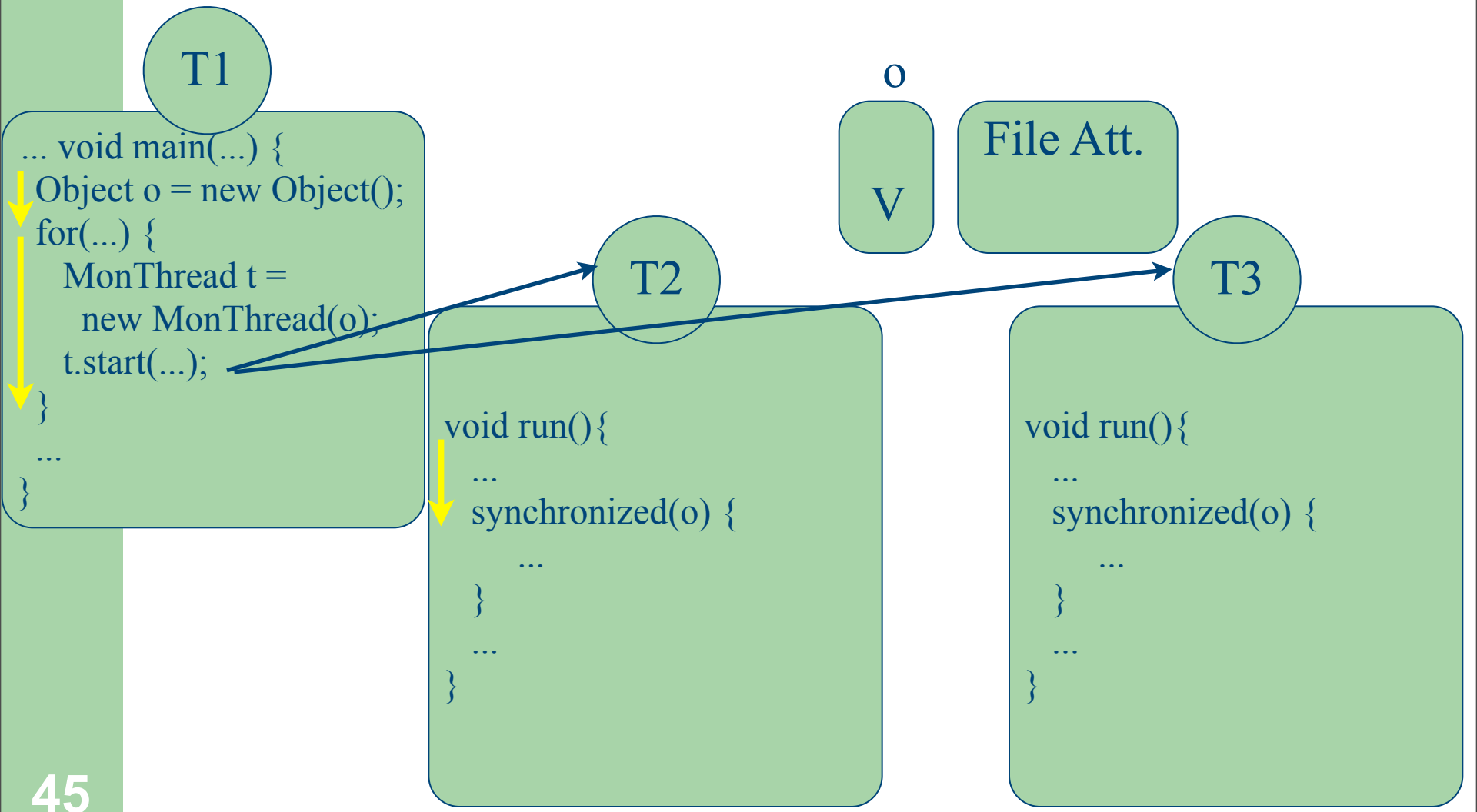
Synchronized



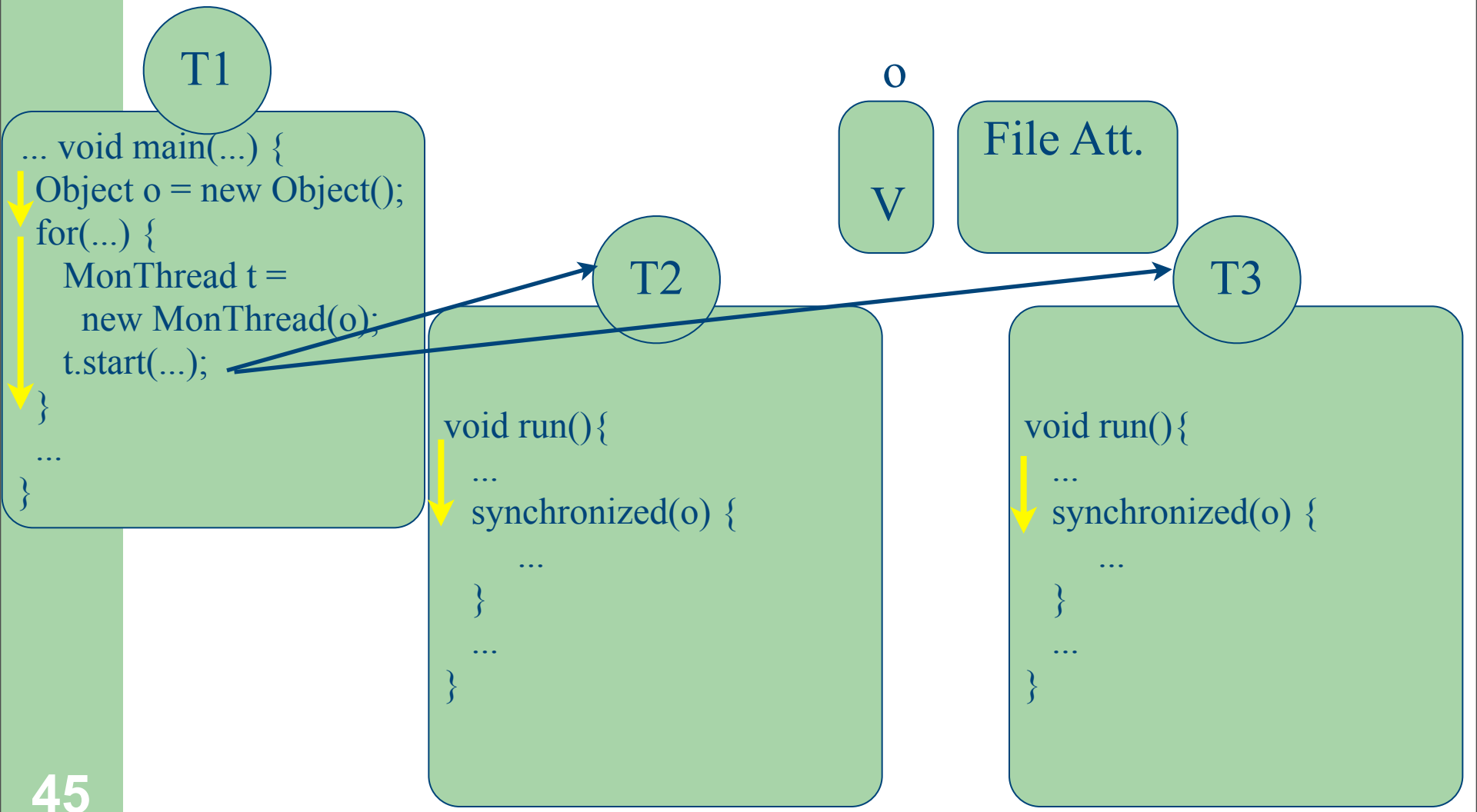
Synchronized



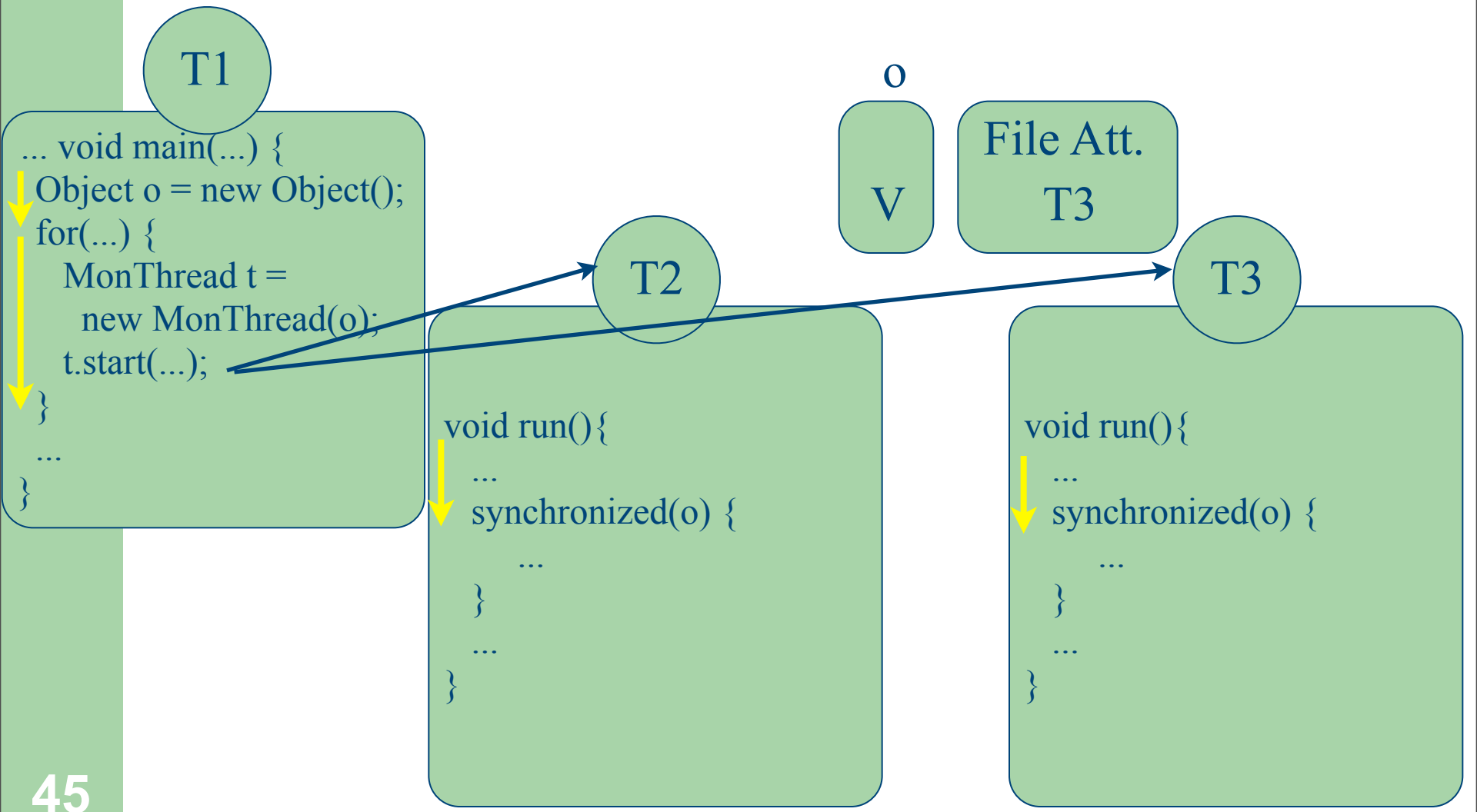
Synchronized



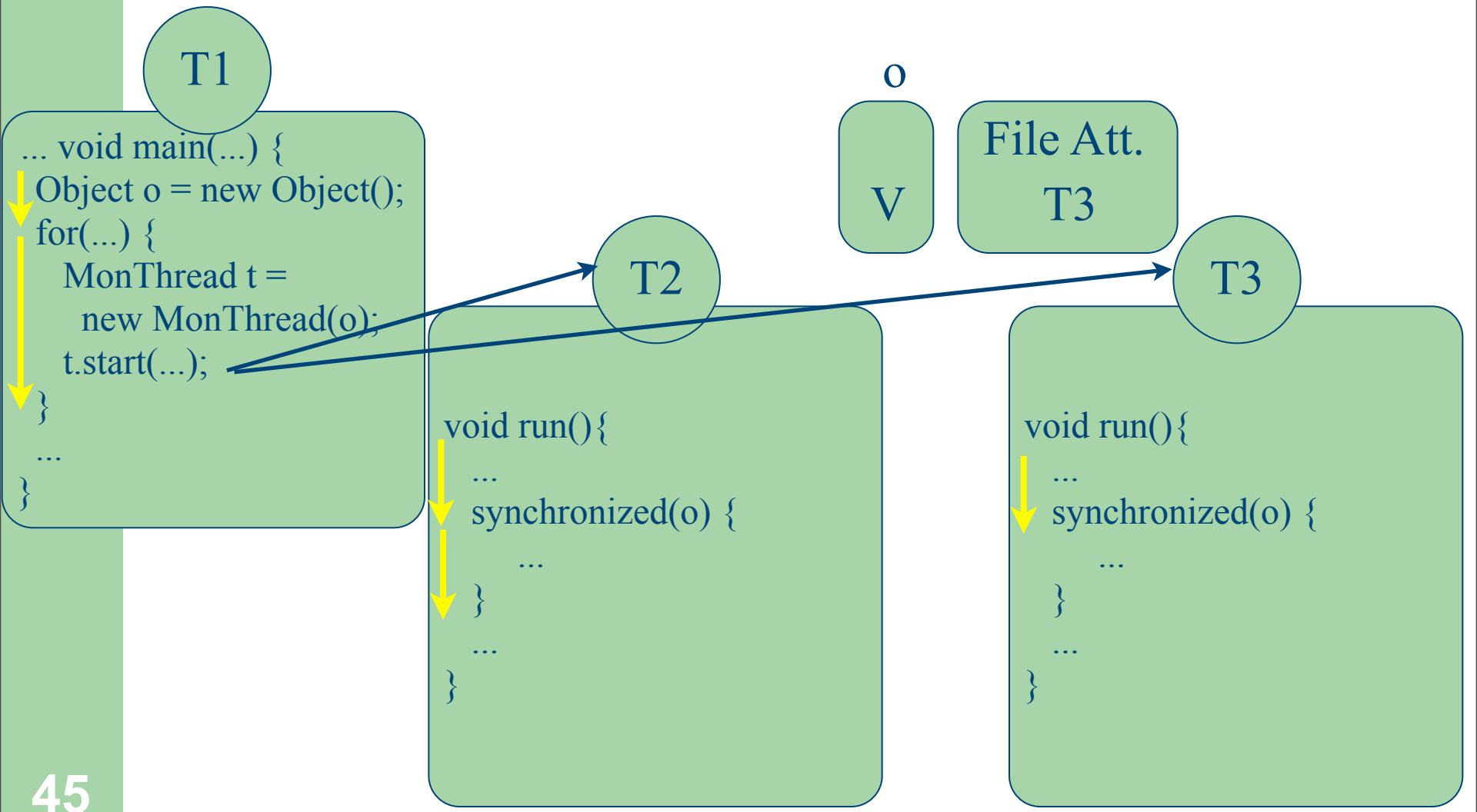
Synchronized



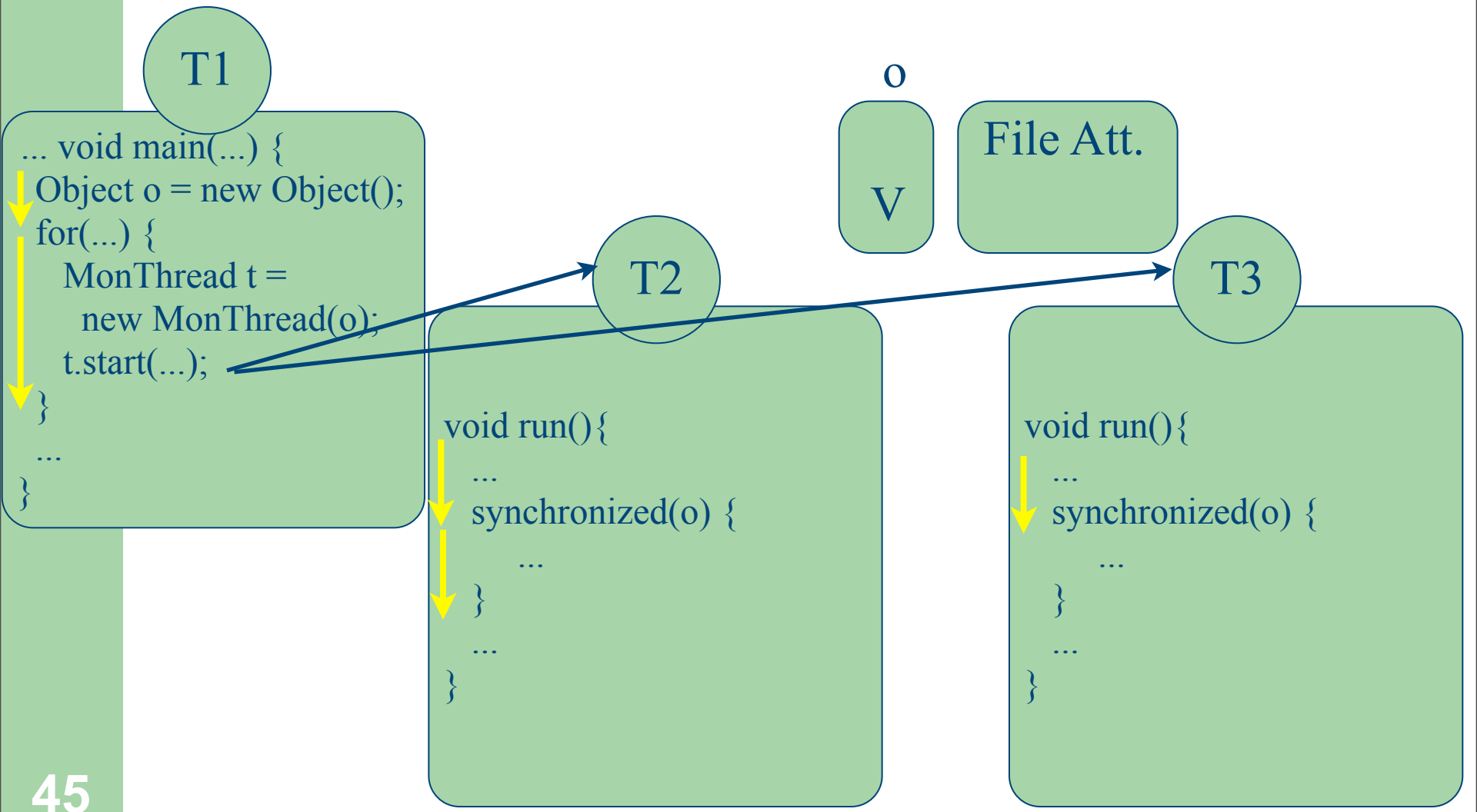
Synchronized



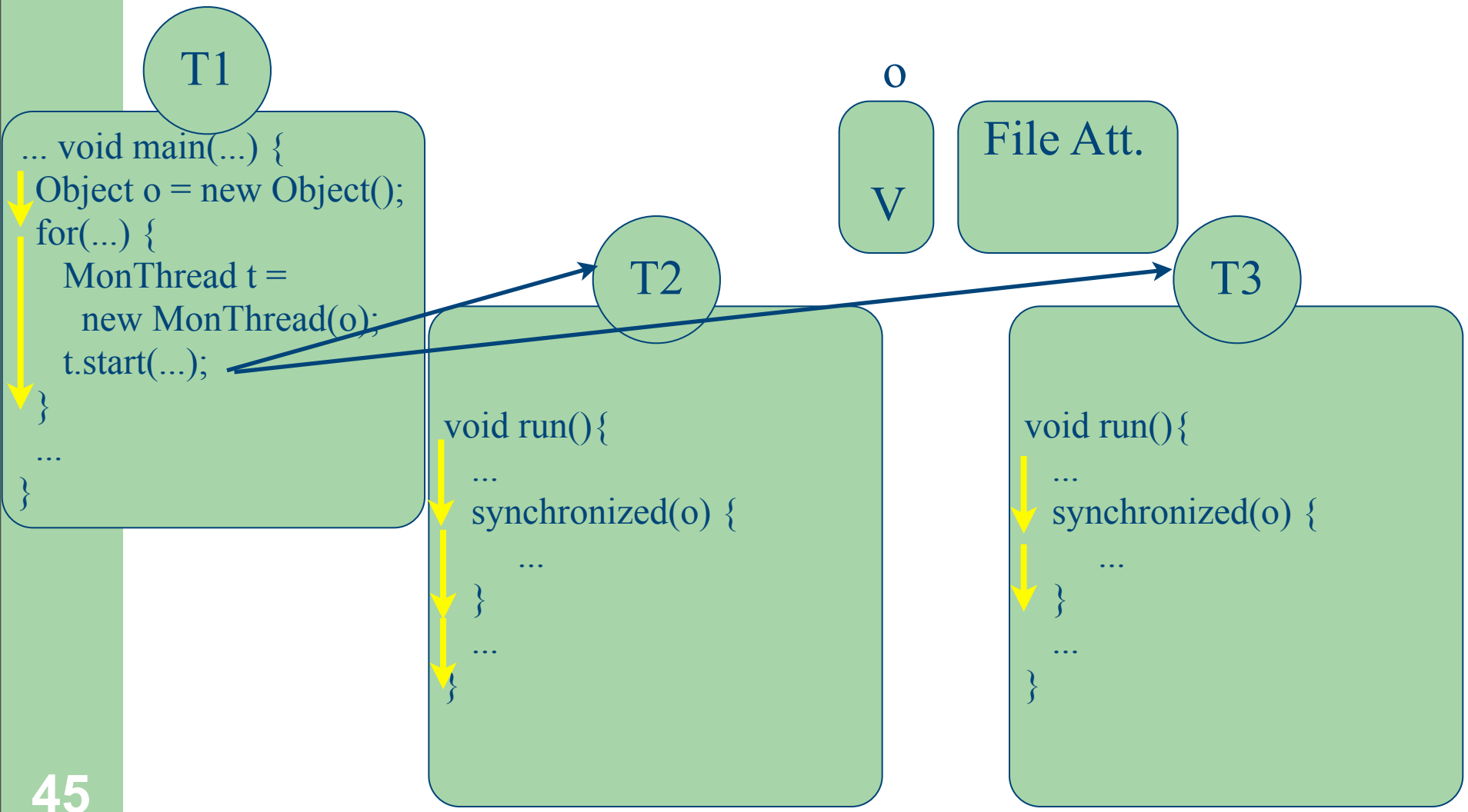
Synchronized



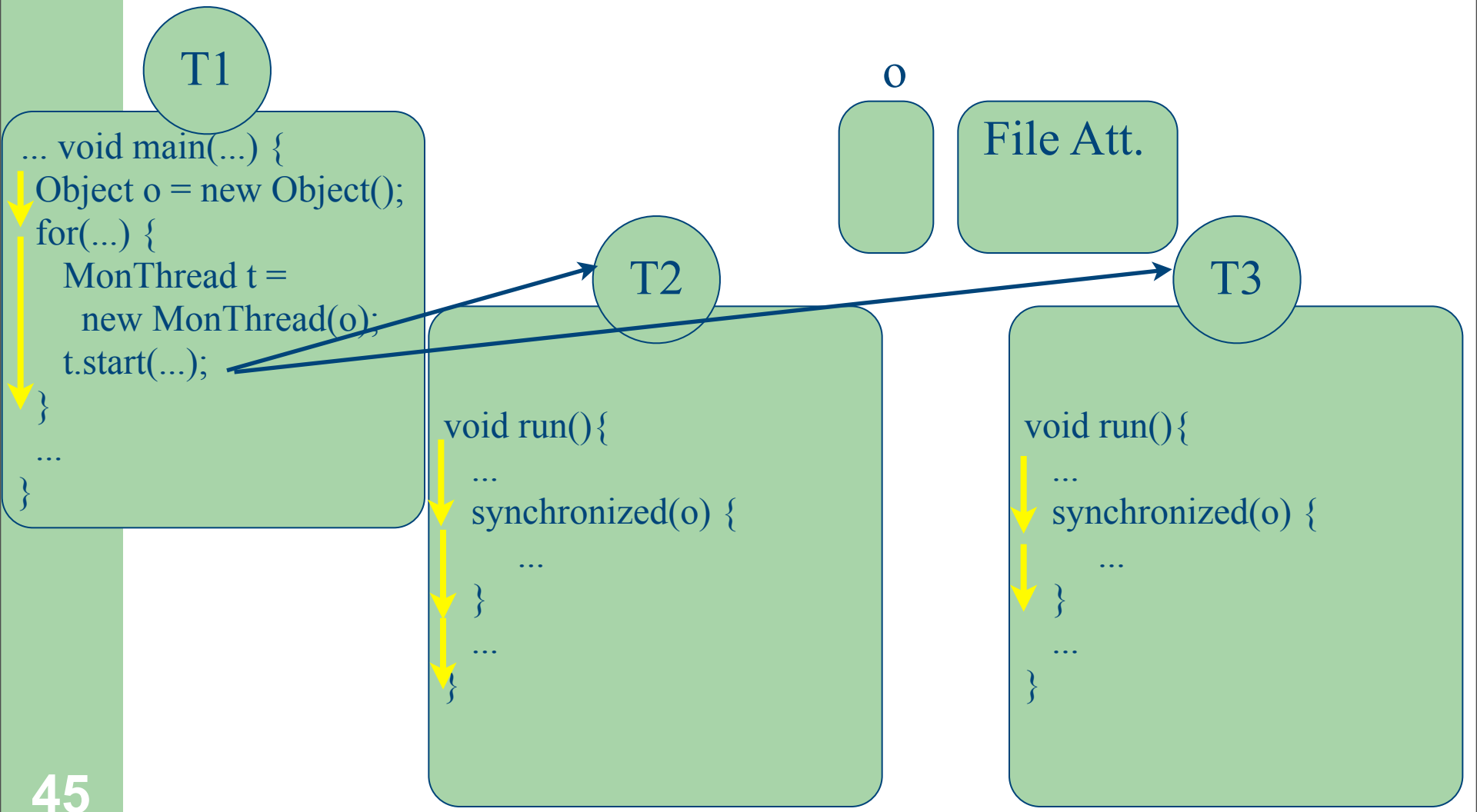
Synchronized



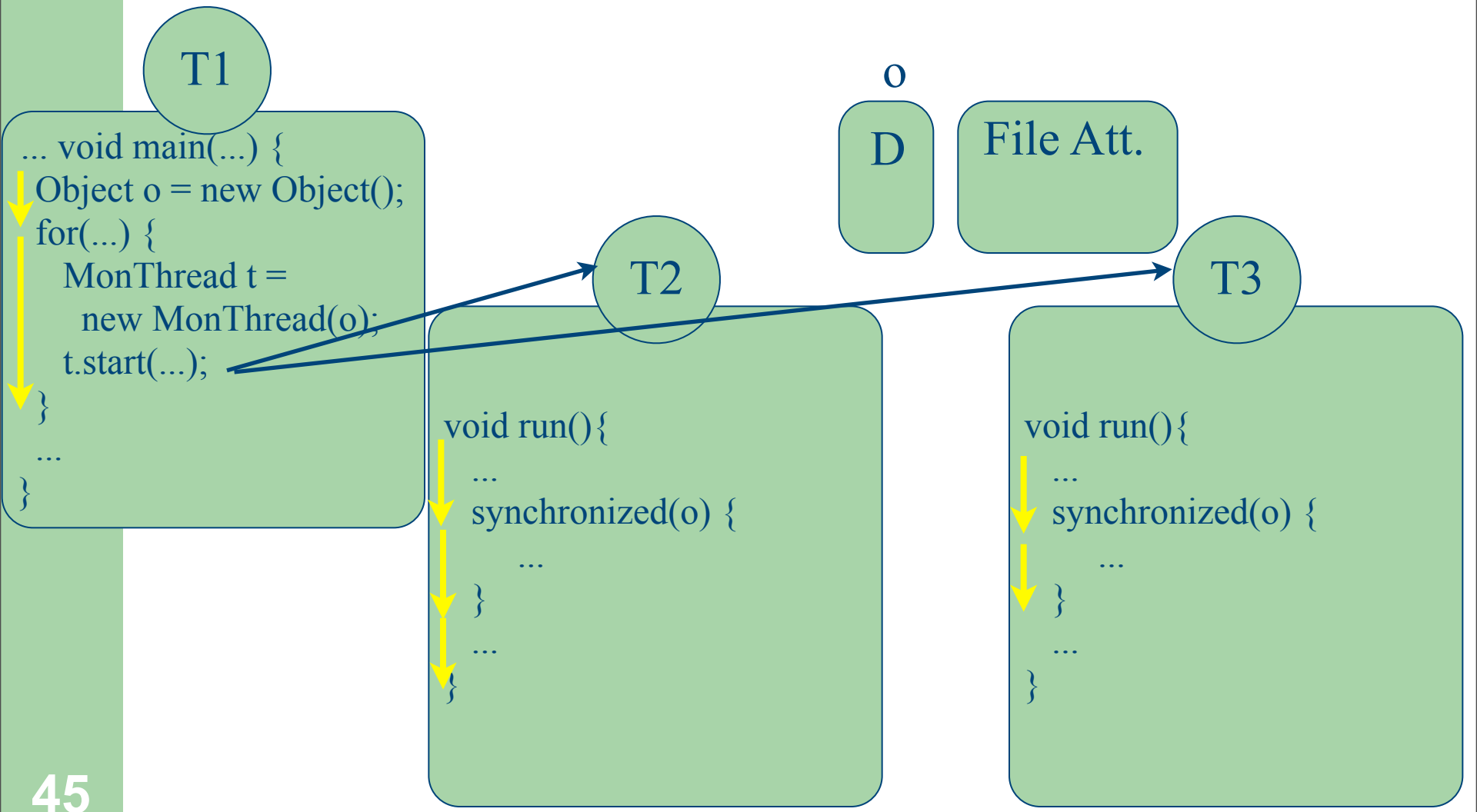
Synchronized



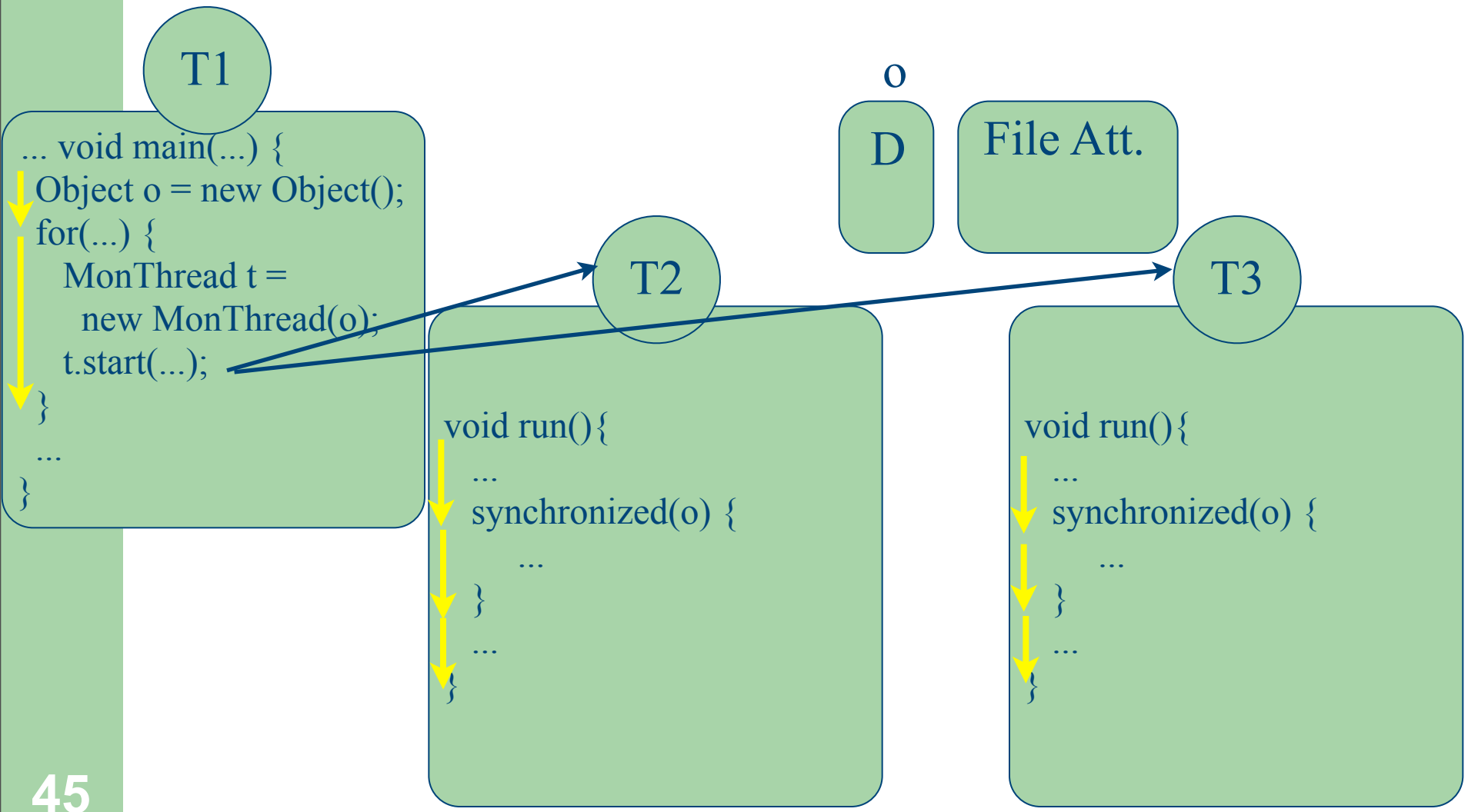
Synchronized



Synchronized



Synchronized



Moniteurs de Hoare

- Les moniteurs ont été inventés par Hoare et Hansen en 1972
- Ils permettent de synchroniser automatiquement plusieurs threads
- A un instant donné un seul thread peut exécuter une méthode du moniteur (on dit qu'il occupe le moniteur ou qu'il le possède), les autres threads qui veulent exécuter des méthodes sont bloqués jusqu'à que le premier thread sorte du moniteur (en quittant la méthode)

- Ce concept existe dans de nombreux langages

- En Java, chaque objet possède un moniteur

Acquisition et restitution d'un moniteur

- Un thread t acquiert automatiquement le moniteur d'un objet o en exécutant du code synchronisé sur cet objet o
- t rend le moniteur en quittant le code synchronisé (ou s'il se met en attente en appelant la méthode wait sur cet objet)
- Il peut quitter le code synchronisé normalement ou si une exception est lancée et non saisie dans le code synchronisé

Moniteurs réentrants

- Un thread qui a acquis le moniteur d'un objet peut exécuter les autres méthodes synchronisées de cet objet
- Il n'est pas bloqué en demandant à nouveau le moniteur

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

```
public class Compte {
  private double solde;
  public void deposer(double somme){
    solde = solde + somme;
  }
  public double getSolde() {
    return solde;
  }
}
```

Exemple

- On lance 3 threads du type suivant :

```
Thread t1 = new Thread() {
  public void run() {
    for (int i = 0; i < 100; i++) {
      compte.deposer(1000);
    }
  }
};
```

- A la fin de l'exécution, on n'obtient pas forcément 300 000

Exemple

- Pour éviter ce problème il faut rendre deposer synchronisée :

```
public synchronized
  void deposer(double somme)
```

- En fait, pour sécuriser la classe contre les accès multiples, il faudra aussi rendre getSolde synchronisée car, en Java, un double n'est pas nécessairement lu en une opération atomique. Il faut donc éviter que getSolde ne soit exécuté en même temps que deposer

Favoriser la détection des problèmes

- En fait, si on exécute le code précédent sans rendre deposer synchronisée, on obtiendra bien souvent (mais pas toujours...) le bon résultat

- Ca dépend du fonctionnement du multitâche du système d'exploitation sous-jacent, et de la JVM

- Pour repérer plus facilement les problèmes de multitâche, on peut ajouter des appels aux méthodes yield ou sleep qui forcent le thread à rendre la main, et permettre ainsi à un autre thread de pouvoir s'exécuter

Provoquer le problème

```
public class Compte {
  private double solde;
  public void deposer(double somme){
    double soldeTemp = solde;
    Thread.yield();
    solde = soldeTemp + somme;
  }
  public double getSolde() {
    return solde;
  }
}
```

Synchronisation et performances

- L'exécution de code synchronisé peut nuire aux performances
- Il peut provoquer des blocages entre threads ; peu de perte de performance s'il n'y a pas de blocage
- Du code synchronisé peut empêcher des optimisations (inlining) au moment de la compilation

Limiter la portée du code synchronisé

- Il faut synchroniser le moins de code possible pour faciliter les accès multiples (les performances seront meilleures s'il y a moins de threads en attente d'un moniteur)
- Attention : éviter d'appeler à l'intérieur d'une portion synchronisée des méthodes qu'un autre développeur peut redéfinir (dans une classe fille) ; en effet, une redéfinition non appropriée peut provoquer des pertes de performance ou même des inter-blocages

Affectations atomiques

- Il est inutile de synchroniser une partie de code qui ne fait qu'affecter (ou lire) une valeur d'une variable de type primitif de longueur 32 bits ou moins (int, short, ...)
- En effet, la spécification du langage Java dit qu'une telle affectation ne peut être interrompue pour donner la main à un autre thread
- Mais, cette spécification n'assure rien pour les affectations de double et de long !

Méthodes statiques

- Si on synchronise une méthode statique, on bloque le moniteur de la classe
- On bloque ainsi tous les appels à des méthodes statiques synchronisées de la classe (mais pas les appels synchronisés sur une instance de la classe)

Méthode synchronisée et héritage

- La redéfinition d'une méthode synchronisée dans une classe fille peut ne pas être synchronisée
- De même, la redéfinition d'une méthode non synchronisée peut être synchronisée

Synchronisation et visibilité de modifications

- La synchronisation permet d'assurer la visibilité par les autres threads de la modification d'une variable par un thread (voir « happens-before » dans la suite de ce cours)

Plan

- Introduction
- Notion de processus
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- **wait et notify**
- Les différents états d'un thread
- Difficultés liées au parallélisme
- Extensions de Java 5

Exécution conditionnelle

- Lorsqu'un programme est multi-tâche, la situation suivante peut se rencontrer :
  - Un thread t1 ne peut continuer son exécution que si une condition est remplie
  - Le fait que la condition soit remplie ou non dépend d'un autre thread t2
- Par exemple, t1 a besoin du résultat d'un calcul effectué par t2
- Une solution coûteuse serait que t1 teste la condition à intervalles réguliers
- Les méthodes **wait**() et **notify**() de la classe Object permettent de programmer plus efficacement ce genre de situation

Utilisation de wait et notify

- Cette utilisation demande un travail coopératif entre les threads t1 et t2 :
  1. Ils se mettent d'accord sur un objet commun **objet**
  2. Arrivé à l'endroit où il ne peut continuer que si la condition est remplie, t1 se met en attente :
     objet.wait();
  3. Quand t2 a effectué le travail pour que la condition soit remplie, il le notifie :
     objet.notify();
  4. ce qui débloque t1

Besoin de synchronisation

- Le mécanisme d'attente-notification lié à un objet met en jeu l'état interne de l'objet ; pour éviter des accès concurrents à cet état interne, une synchronisation est nécessaire
- Les appels **objet.wait()** et **objet.notify()** (et **objet.notifyAll()**) ne peuvent donc être effectués que dans du code synchronisé sur **objet**
- Autrement dit, **wait** et **notify** ne peuvent être lancés que dans une section critique

Méthode wait()

- `public final void wait()`
  `throws InterruptedException`
- `objet.wait()`
  - nécessite que le thread en cours possède le moniteur de objet
  - bloque le thread qui l'appelle, jusqu'à ce qu'un autre thread appelle la méthode objet.notify() ou objet.notifyAll()
  - libère le moniteur de l'objet
  - l'opération « blocage du thread – libération du moniteur » est atomique

Utilisation de wait

- Mauvaise utilisation :

```
if (!condition) objet.wait();
```

  - si on quitte l'attente avec le **wait**(), cela signifie qu'un autre thread a notifié que la condition était vérifiée (true)
  - mais, après la notification, et avant le redémarrage de ce thread, un autre thread a pu prendre la main et modifier la condition

- Le bon code (dans du code synchronisé) :

```
while (!condition) {
  objet.wait();
}
```

Méthode notifyAll()

- public final void notifyAll()
- objet.notifyAll()
  - nécessite que le thread en cours possède le moniteur de **objet**
  - débloque tous les threads qui s'étaient bloqués sur l'objet avec objet.wait()

Méthode notifyAll()

- Un seul des threads débloqués va récupérer le moniteur ; on ne peut prévoir lequel
- Les autres devront attendre qu'il relâche le moniteur pour être débloqués à tour de rôle, mais ils ne sont plus bloqués par un wait
- En fait, souvent ils se bloqueront à nouveau eux-mêmes (s'ils sont dans une boucle while avec wait)

Méthode notify()

- public final void notify()
- objet.notify()
  - idem notifyAll() mais
  - ne débloque qu'un seul thread
- On ne peut prévoir quel sera le thread débloqué et, le plus souvent, il vaut donc mieux utiliser notifyAll()

Déblocage des threads

- Le thread débloqué (et élu) ne pourra reprendre son exécution que lorsque le thread qui l'a notifié rendra le moniteur de l'objet en quittant sa portion de code synchronisé
- Le redémarrage et l'acquisition se fait dans une opération atomique

Notifications perdues

- Si un notifyAll() (ou notify()) est exécuté alors qu'aucun thread n'est en attente, il est perdu : il ne débloquera pas les wait()exécutés ensuite

Exemple avec wait-notify

- Les instances d'une classe Depot contiennent des jetons
- Ces jetons sont
  - déposés par un producteur
  - consommés par des consommateurs
  - Les producteurs et consommateurs sont des threads

Exemple avec wait-notify

```
public class Depot {
  private int nbJetons = 0;
  public synchronized void consomme() {
    try {
      while (nbJetons == 0) {
        wait();
      }
      nbJetons--;
    } catch (InterruptedException e) {}
  }
  public synchronized void produit(int n) {
    nbJetons += n; notifyAll();
  }
```

Wait/notify

T1

```
... void main(...) {
  Depot dep =
        new Depot();
  Thread2 t2 =
        new Thread2(dep);
  Thread3 t3 =
        new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```

Wait/notify

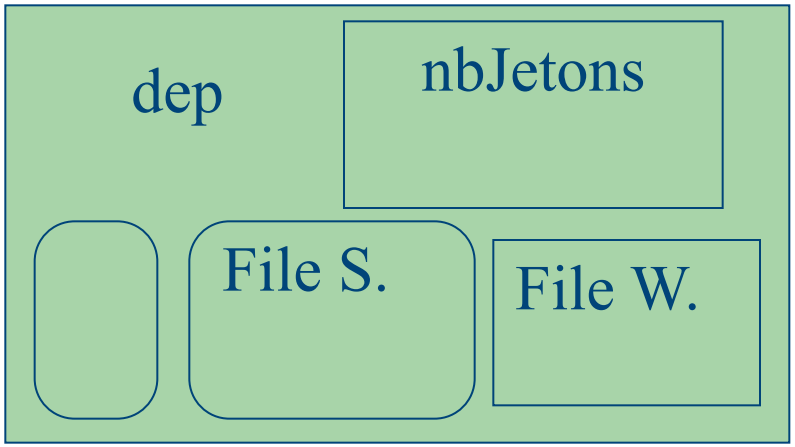
T1

```
... void main(...) {
  Depot dep =
        new Depot();
  Thread2 t2 =
        new Thread2(dep);
  Thread3 t3 =
        new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```

Wait/notify

T1

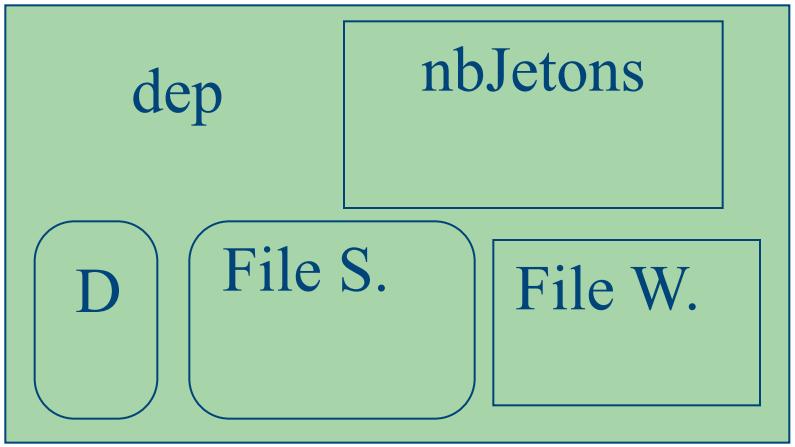
```
... void main(...) {
  Depot dep =
        new Depot();
  Thread2 t2 =
        new Thread2(dep);
  Thread3 t3 =
        new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```



Wait/notify

T1

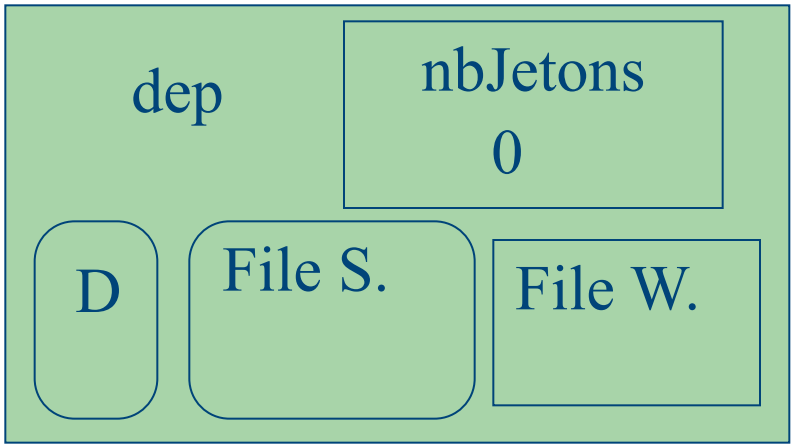
```
... void main(...) {
  Depot dep =
        new Depot();
  Thread2 t2 =
        new Thread2(dep);
  Thread3 t3 =
        new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```



Wait/notify

T1

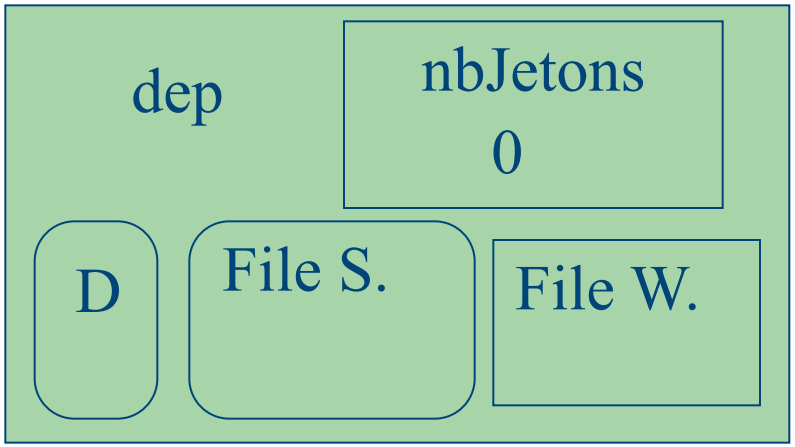
```
... void main(...) {
  Depot dep =
        new Depot();
  Thread2 t2 =
        new Thread2(dep);
  Thread3 t3 =
        new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```



Wait/notify

T1

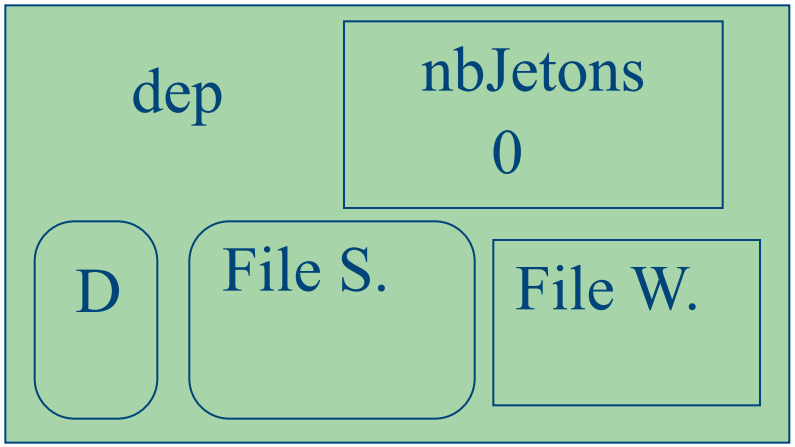
```
... void main(...) {
  Depot dep =
       new Depot();
  Thread2 t2 =
       new Thread2(dep);
  Thread3 t3 =
       new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```



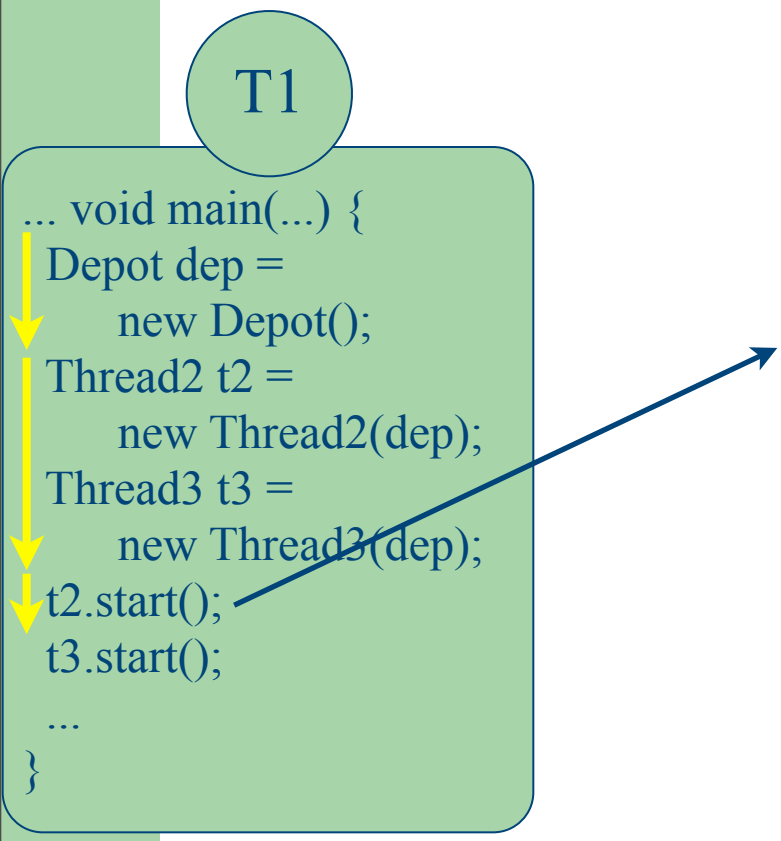
Wait/notify

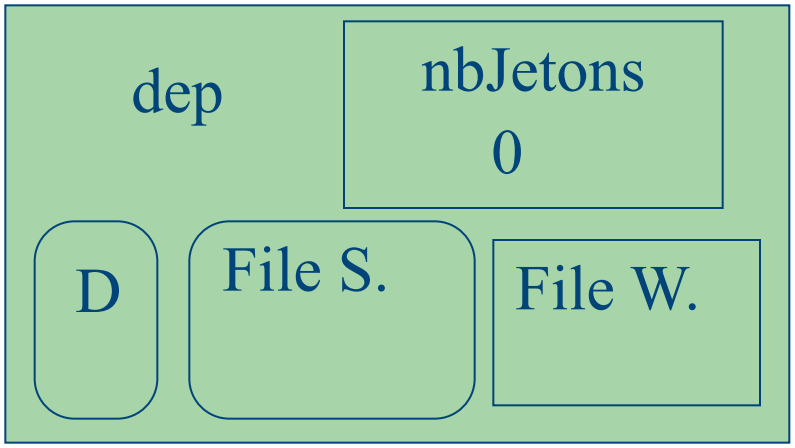
T1

```
... void main(...) {
  Depot dep =
       new Depot();
  Thread2 t2 =
       new Thread2(dep);
  Thread3 t3 =
       new Thread3(dep);
  t2.start();
  t3.start();
  ...
}
```

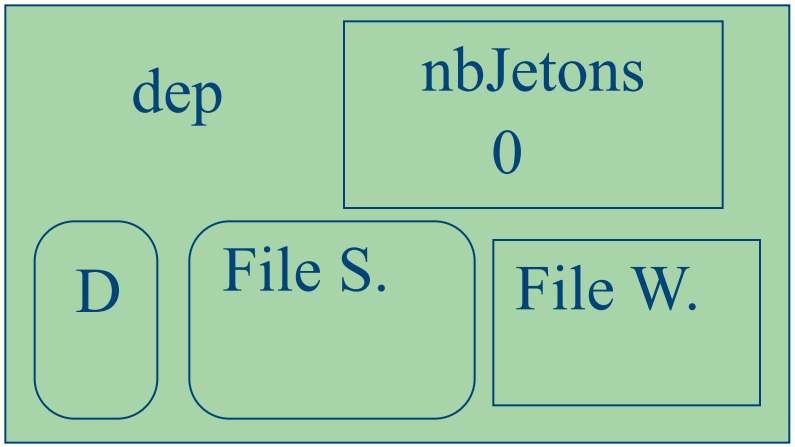


Wait/notify


T1
... void main(...) {
Depot dep =
new Depot();
Thread2 t2 =
new Thread2(dep);
Thread3 t3 =
new Thread3(dep);
t2.start();
t3.start();
...
}


dep
nbJetons
0
D
File S.
File W.

Wait/notify



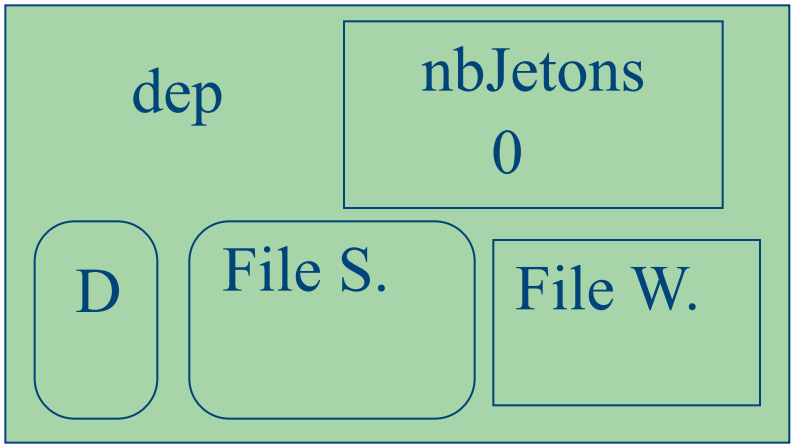
T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
 ...
}
```

T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

Wait/notify



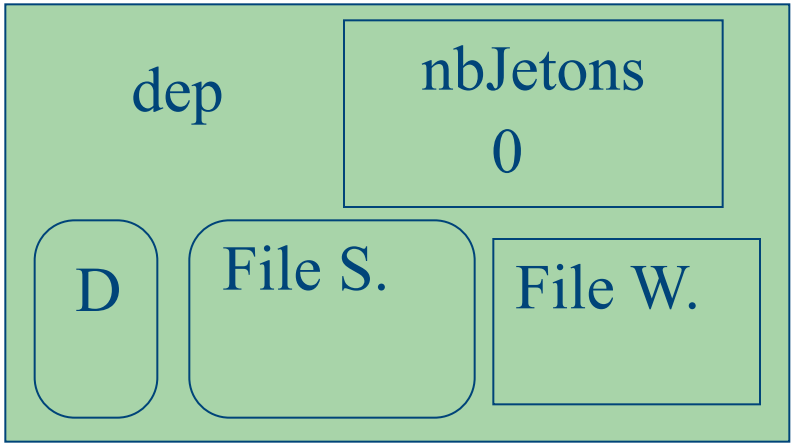
T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
 ...
}
```

T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

Wait/notify



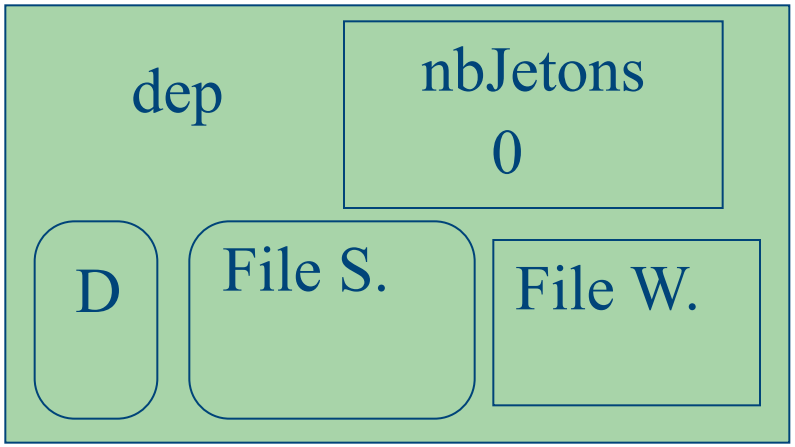
T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

Wait/notify



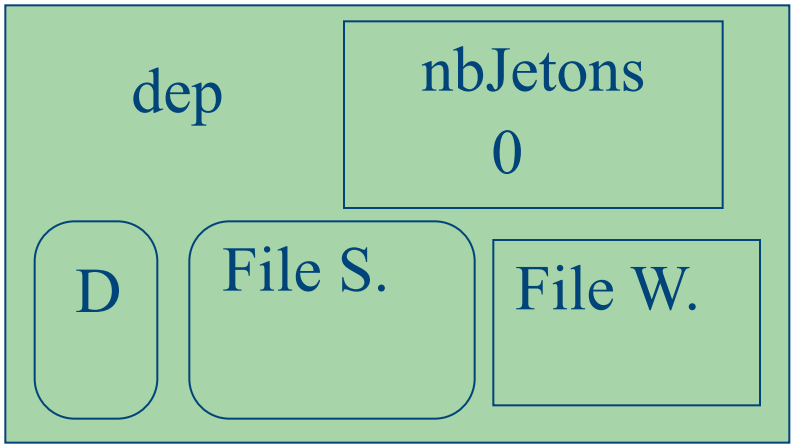
T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

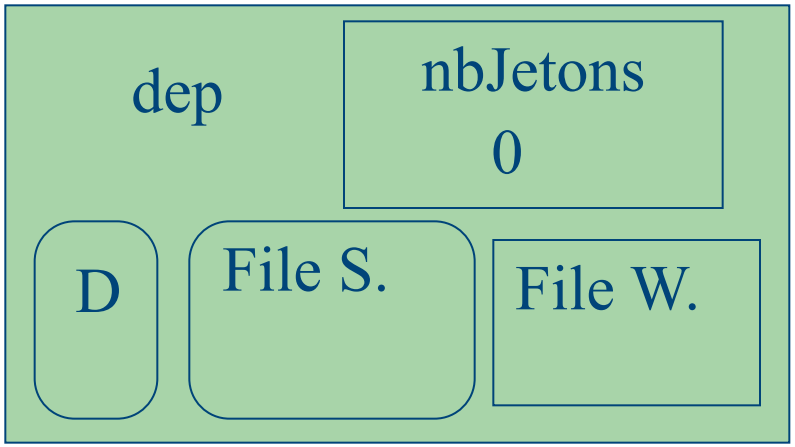
T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

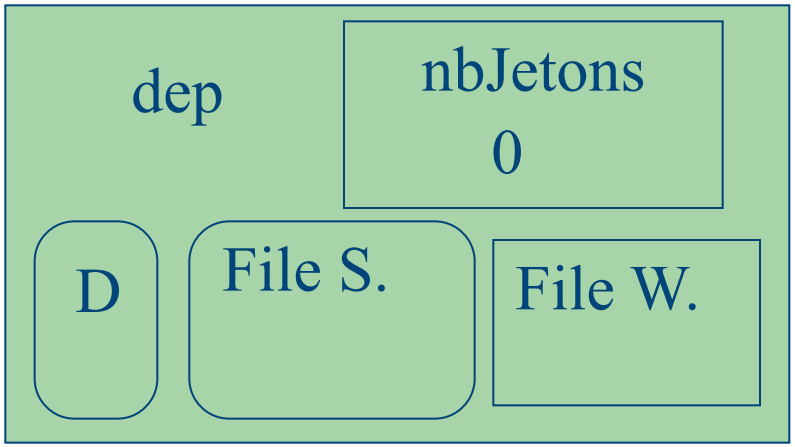
T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

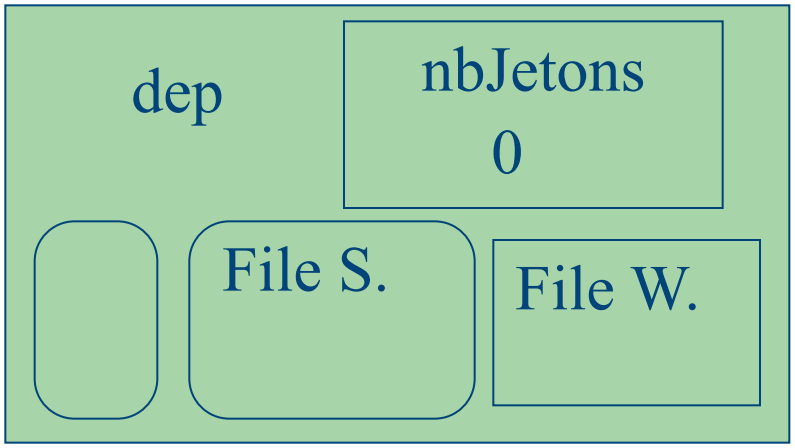
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

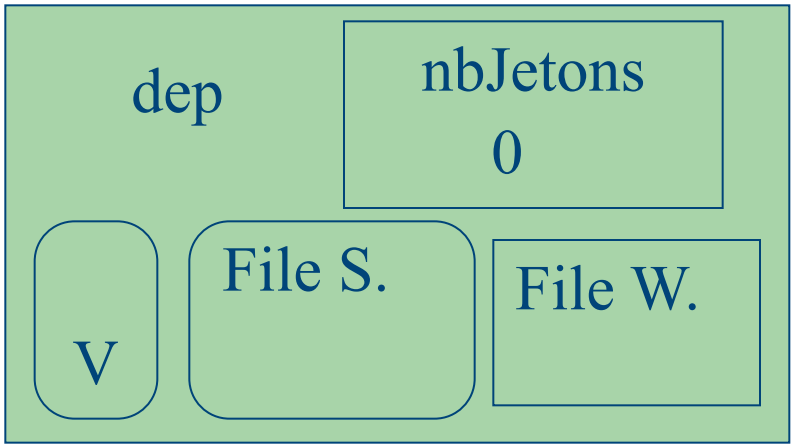
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

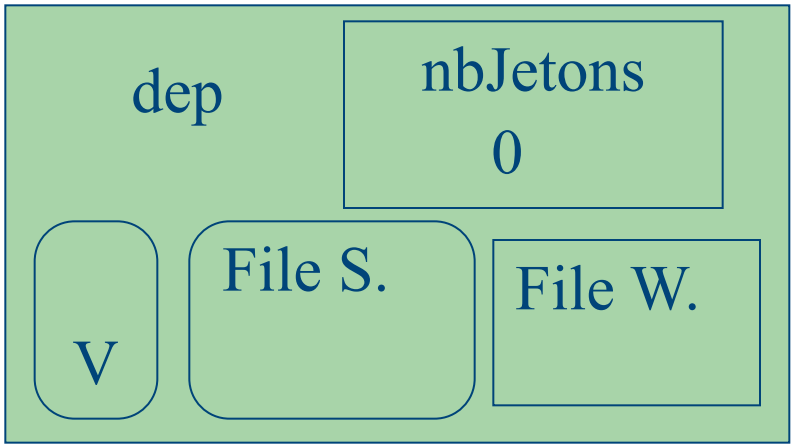
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

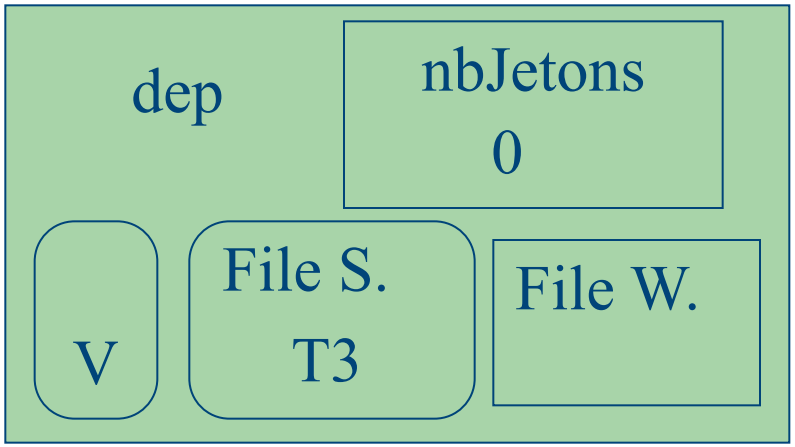
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

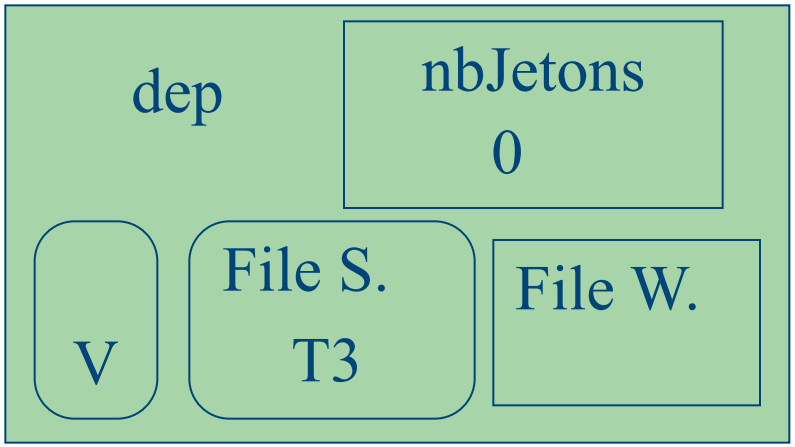
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
.   while(nbJetons == 0)
          wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

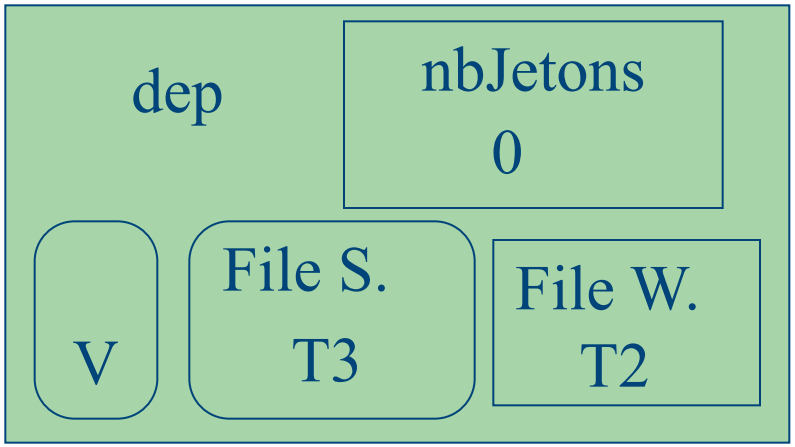
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
  Depot dep =
       new Depot();
  Thread2 t2 =
       new Thread2(dep);
  Thread3 t3 =
       new Thread3(dep);
  t2.start();
  t3.start();
   ...
}
```

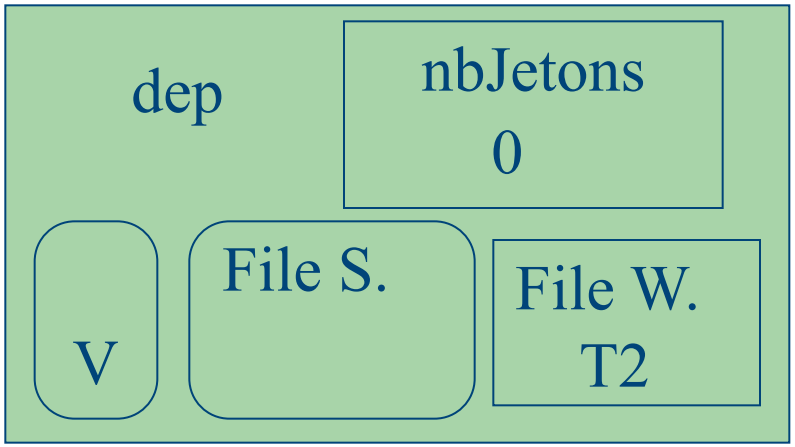
T2

```
void run(){
   while(true)
      consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
   nbJetons--;
}
```

T3

```
void run(){
   while(true)
      produit(1);
}

public sync... produit(int n)
{
   nbJetons+=n;
   notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

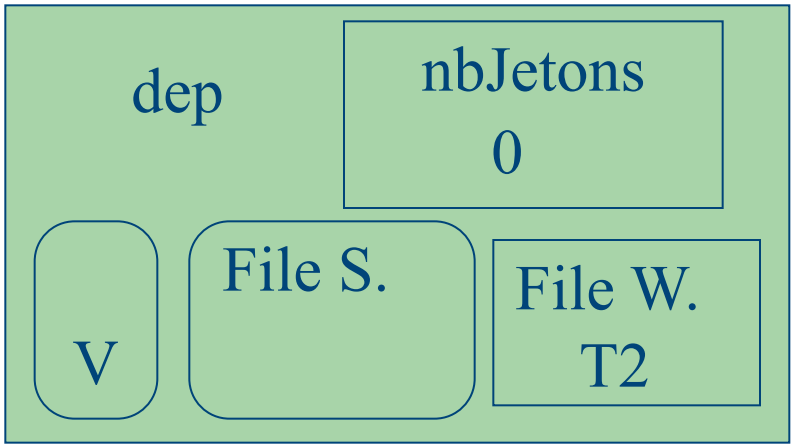
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

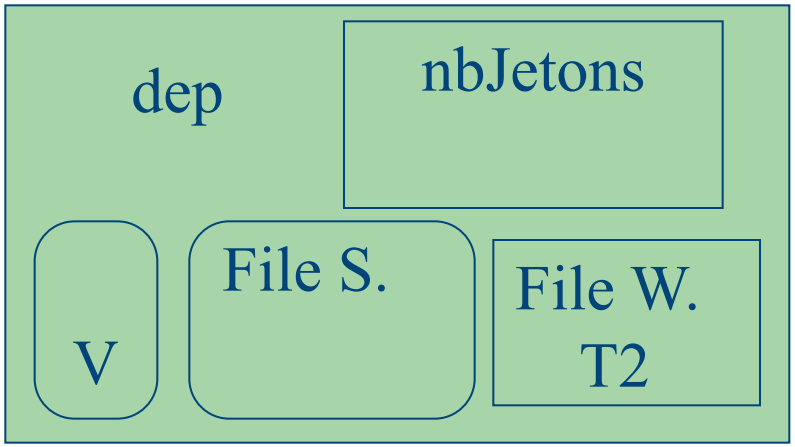
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

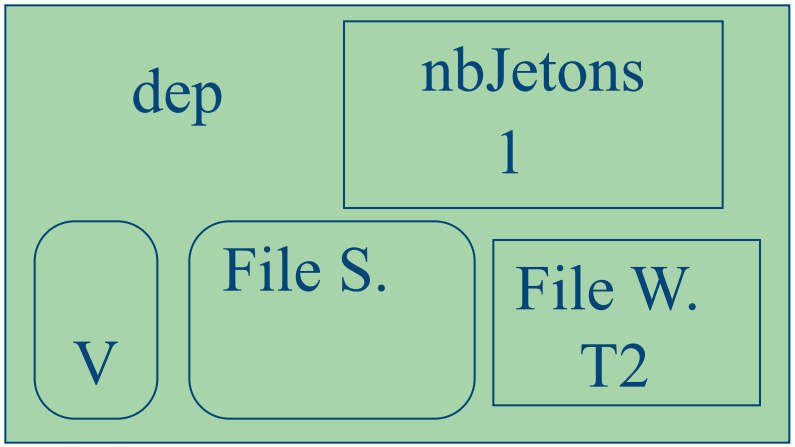
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
   nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

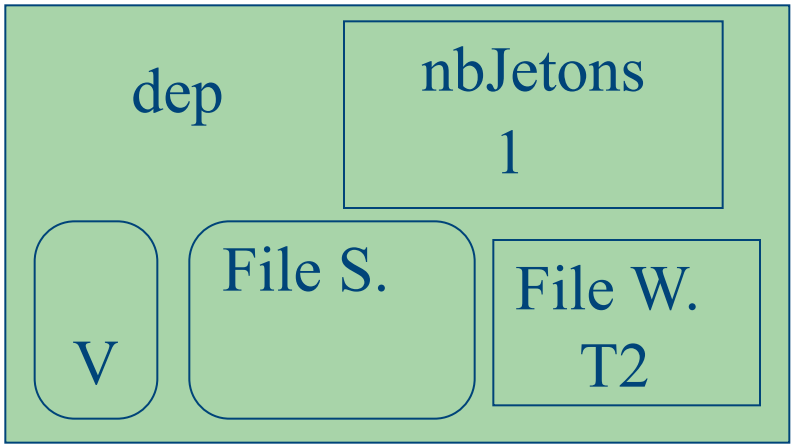
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

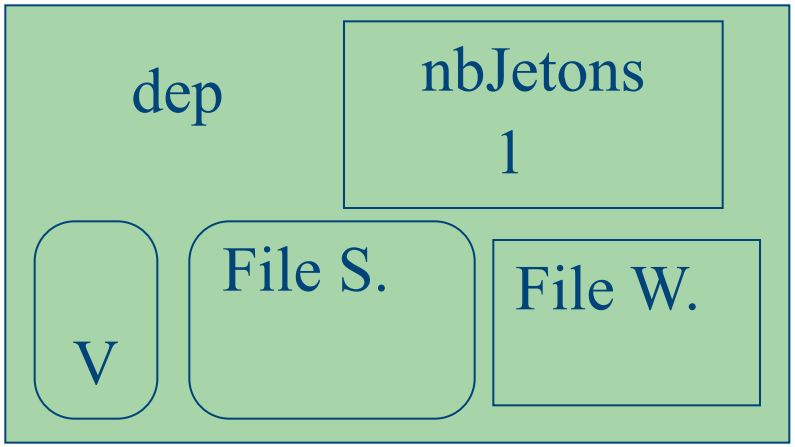
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
   nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
   nbJetons+=n;
   notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

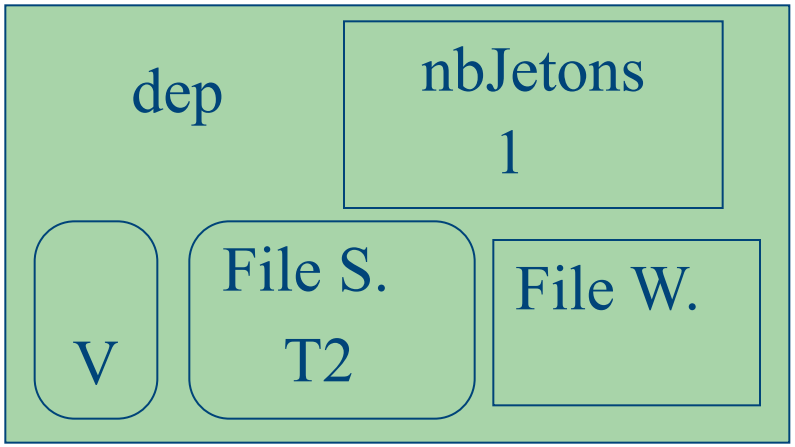
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

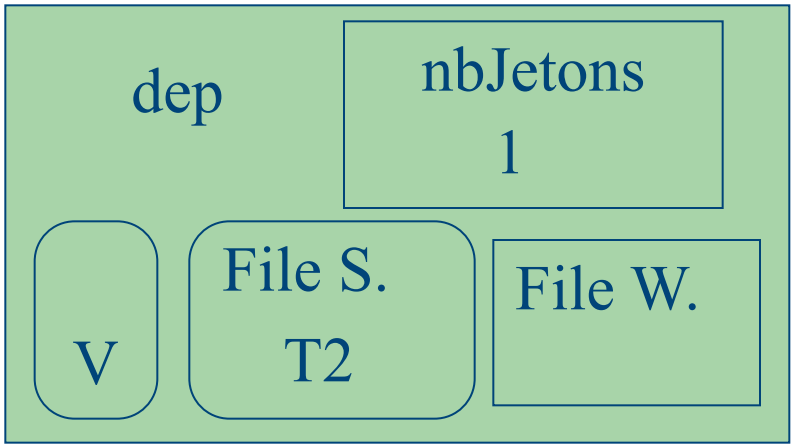
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

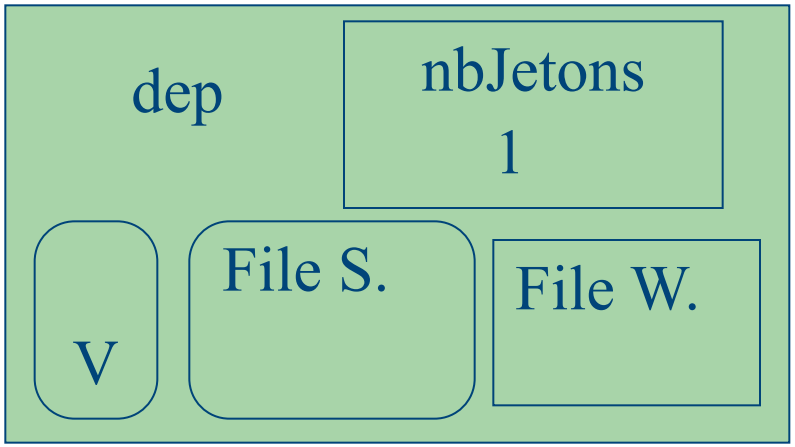
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

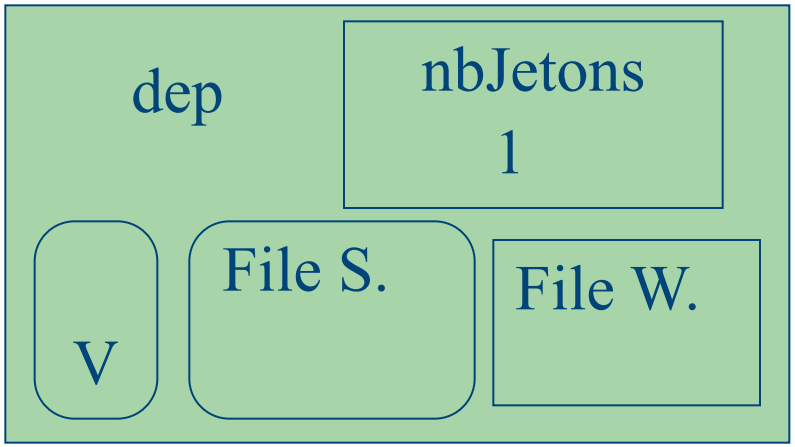
T2

```
void run(){
  while(true)
     consomme();
}

public sync... consomme() {
   while(nbJetons == 0)
         wait();
    nbJetons--;
}
```

T3

```
void run(){
  while(true)
     produit(1);
}

public sync... produit(int n)
{
    nbJetons+=n;
    notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

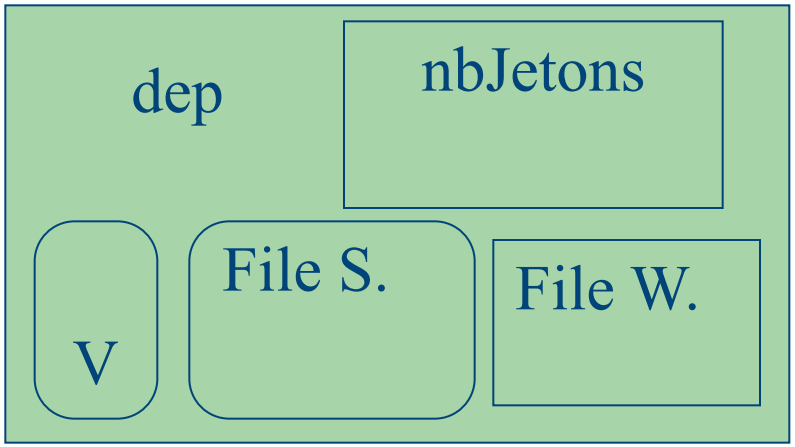
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

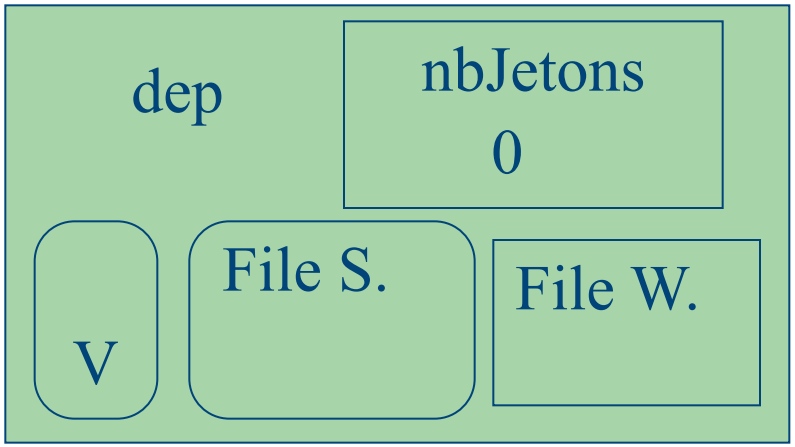
T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Wait/notify



T1

```
... void main(...) {
 Depot dep =
      new Depot();
 Thread2 t2 =
      new Thread2(dep);
 Thread3 t3 =
      new Thread3(dep);
 t2.start();
 t3.start();
  ...
}
```

T2

```
void run(){
  while(true)
    consomme();
}

public sync... consomme() {
  while(nbJetons == 0)
        wait();
  nbJetons--;
}
```

T3

```
void run(){
  while(true)
    produit(1);
}

public sync... produit(int n)
{
  nbJetons+=n;
  notifyAll();
}
```

Variantes de wait

- Si on ne veut pas attendre éternellement une notification, on peut utiliser une des variantes suivantes de wait :
  - public void wait(long timeout)
  - public void wait(long timeout, int nanos)
- Dans ce cas, le thread doit gérer lui-même le fait de connaître la cause de son déblocage (notify ou temps écoulé)

Stopper/Suspendre un thread

- Dans la première version de Java on pouvait stopper, suspendre et relancer un thread
- Désormais c'est impossible à cause du risque de problème de synchronisation :
  - lorsqu'on arrête un thread, tous les moniteurs qu'il possédait sont relâchés ce qui peut provoquer des problèmes de synchronisation
  - lorsqu'on suspend un thread, tous les moniteurs qu'il possède sont bloqués jusqu'à qu'on le relance ce qui peut bloquer l'application
- Aujourd'hui on doit utiliser des variables booléennes pour cela

Simuler stop()

- Pour rendre possible l'arrêt d'un thread T, on peut utiliser une variable arretThread visible depuis T et les threads qui peuvent stopper T :
  - T initialise arretThread à false lorsqu'il démarre
  - pour stopper T, un autre thread met arretThread à true
  - T inspecte à intervalles réguliers la valeur de arretThread et s'arrête quand arretThread a la valeur true
- arretThread doit être déclarée volatile si elle n'est pas manipulée dans du code synchronisé (volatile est étudié plus loin dans ce cours)

Simuler stop()

- Le mécanisme décrit précédemment pour stopper un thread ne fonctionne pas si le thread est en attente (wait , sleep, attente d'une entrée-sortie avec le paquetage java.nio depuis le JDK 1.4)
- Dans ce cas, on utilise interrupt() qui interrompt les attentes et met l'état du thread à « interrupted » (voir la section sur la classe Thread du début de ce cours)

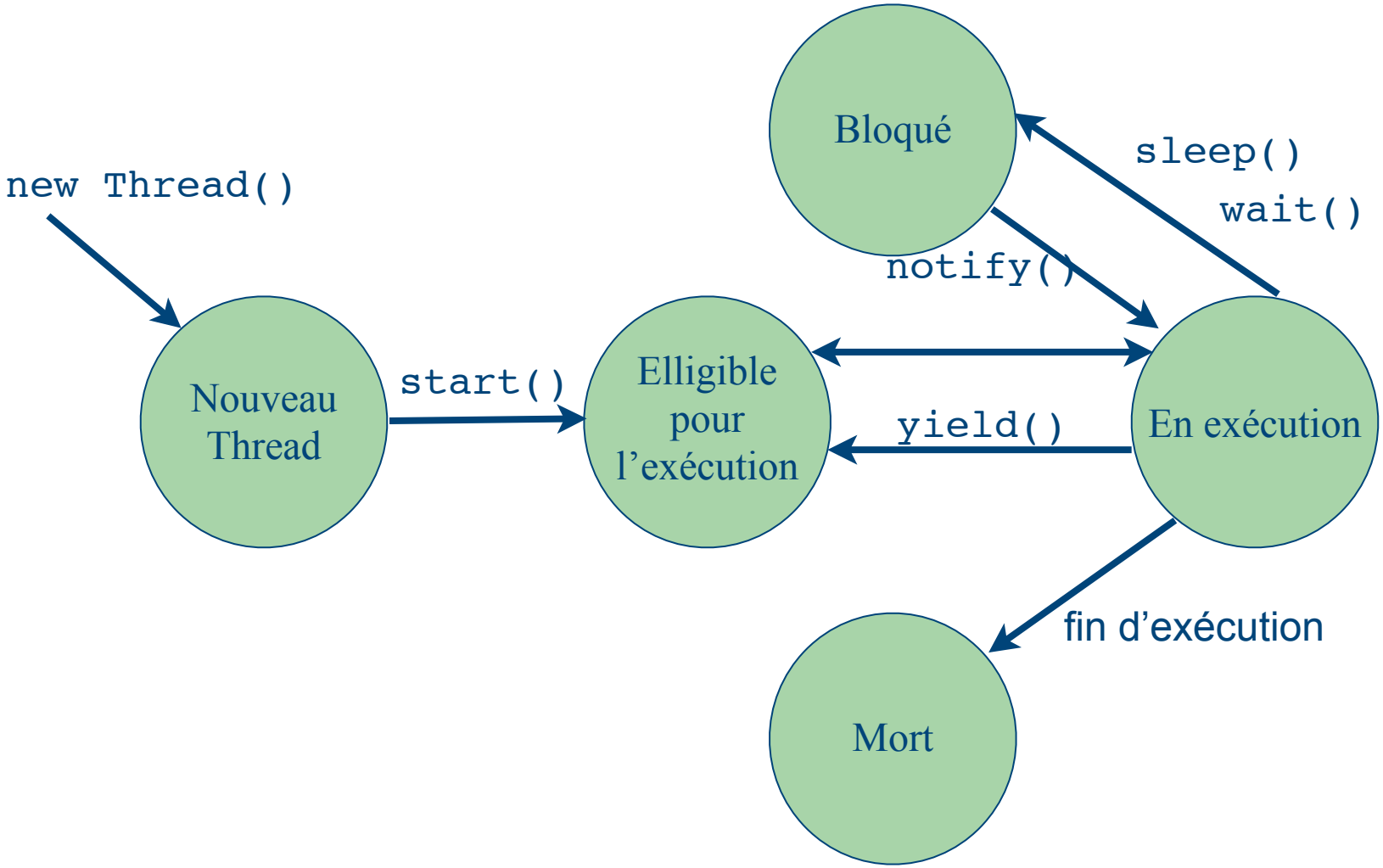
Simuler suspend/resume

- On peut les remplacer en utilisant une variable suspendreThread comme pour stop()
- Comme il faut pouvoir suspendre et reprendre l'exécution on doit en plus utiliser wait() et notify()
- Pendant son exécution le thread scrute la valeur de suspendreThread
- Si la valeur est true, le thread se met en attente avec wait sur un objet o
- Quand un autre thread veut relancer l'exécution du thread, il met la variable suspendreThread à false et il appelle notify sur l'objet o
- Rappel : les appels de wait et notify doivent se faire dans des sections synchronisées sur o

Plan

- Introduction
- Notion de processus
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- wait et notify
- **Les différents états d'un thread**
- Difficultés liées au parallélisme
- Extensions de Java 5

Cycle de vie d'un thread



Threads démons

- 2 types de threads
  - les threads utilisateur
  - les démons
- La différence :
  - la JVM fonctionne tant qu'il reste des threads utilisateurs en exécution
  - la JVM s'arrête s'il ne reste plus que des démons
- Les démons sont là seulement pour rendre service aux threads utilisateur. Exemple : ramasse-miettes
- La méthode void setDaemon(boolean on) de la classe Thread permet d'indiquer que le thread sera un démon (thread utilisateur par défaut)
- Elle doit être appelée avant le démarrage du thread par l'appel de start()

Plan

- Introduction
- Notion de processus
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- **Difficultés liées au parallélisme**
- Extensions de Java 5

Difficultés liées au parallélisme

- Si un thread t1 doit attendre 2 notify de 2 autres threads (t2 et t3), ce serait faux de coder :
  o.wait();
  o.wait();
- En effet, les 2 notify() peuvent arriver «presque en même temps» :
  - le thread t2 envoie un 1er notify() qui débloque le 1er wait() ; il relâche ensuite le moniteur et permet ainsi ...
  - ... au thread t3 d'envoyer le 2ème notify() avant que le 2ème wait() ne soit exécuté
  - le thread t1 reste bloqué éternellement sur le 2ème wait() (qui ne recevra jamais de notify())

Comment se débrouiller

- On compte le nombre de notify() avec une variable qui est incrémentée dans une section critique (synchronisée) qui contient le notify() (pour être certain que la variable représente vraiment le nombre de notify()) :

```
nbNotify++;
o.notifyAll();
```

- Et on se met en attente dans une boucle (dans une portion synchronisée) :

```
while (nbNotify < 2) {
    o.wait();
}
```

- Si on reçoit 1 notify() entre les 2 wait(), nbNotify sera égal à 2 et on sortira de la boucle sans faire le 2ème wait()

Eviter la synchronisation

- Comme la synchronisation a un coût non négligeable, il faut essayer de l'éviter quand c'est possible
- Par exemple, si un seul thread écrit une valeur de type int qui est lue par plusieurs autres threads, on peut se passer de synchronisation car les opérations de lecture/écriture de int sont atomiques
- Mais attention, il y a de nombreux pièges !

Partage de variables par les threads

- Soit v une variable partagée par plusieurs threads
- Si le thread T modifie la valeur de v, cette modification peut ne pas être connue immédiatement par les autres threads
- Par exemple, le compilateur a pu utiliser un registre pour conserver la valeur de v pour T
- La spécification de Java n'impose la connaissance de cette modification par les autres threads que lors de l'acquisition ou du relâchement du moniteur d'un objet (synchronized)

Volatile

- Pour éviter ce problème, on peut déclarer la variable v volatile
- On est ainsi certain que tous les threads partageront une zone mémoire commune pour ranger la valeur de la variable v
- De plus, si une variable de type long ou double est déclarée volatile, sa lecture et son écriture sont garanties atomiques

Attention !

- Dans les anciennes versions de Java, une variable volatile init ne pouvait servir à indiquer l'initialisation de variables non volatiles
- Par exemple, dans le code suivant :
  v1 = 8;
  v2 = 3;
  init = true;
- même si init est lue comme vraie par un autre thread, v1 et v2 n'ont peut-être pas été initialisées car le compilateur pouvait intervertir les instructions s'il le jugeait utile pour l'optimisation

« Happens-before »

- Les nouvelles versions de Java (à partir de la version 5) formalisent les cas où une modification effectuée par un thread est nécessairement connue par un autre thread avec la notion de « happens-before » (arrive-avant) définie dans le chapitre 17 de la spécification du langage
- Ce qui signifie que, dans tous les autres cas, cette modification n'est peut-être par visible par l'autre thread et donc qu'il faut utiliser des synchronized

Définition de « happens-before »

- Une modification effectuée par un thread T1 est assurée d'être visible par un thread T2 si et seulement si cette modification happens-before la lecture par le thread T2
- Cette notion de happens-before n'est pas nécessairement liée au temps ; il faut des conditions bien précises pour qu'elle ait lieu
- Le transparent suivant résume les cas de « bas niveau » ; l'API java.util.concurrent ajoute d'autres cas de plus haut niveau

Cas de « happens-before »

- Dans un seul thread, une action happens-before toute action qui arrive après dans le code (i.e. il ne peut pas y avoir de problèmes de causalité dans le thread mais le code peut tout de même avoir été ré-ordonné par le compilateur)
- La libération d'un moniteur happens-before l'acquisition future de ce moniteur
- L'écriture dans un champ volatile happens-before la lecture future de la valeur de ce champ
- t1.start() happens-before toutes les actions effectuées par le thread t1
- Les actions d'un thread t1 happen-before les actions d'un thread t2 qui suivent l'instruction t1.join()

Transitivité de « Happens-before »

- Si une action A happens-before une action B et si B happens-before une action C, alors A happens-before C
- Ce qui signifie, par exemple, que toute action effectuée par un thread t1 happens-before toute action effectuée par un thread t2, qui suit l'exécution de t1.join()
- Ce qui signifie aussi que le transparent précédent « Attention ! » sur l'utilisation erronée d'une variable volatile n'est plus d'actualité

Priorités des threads

- Si plusieurs threads de même priorité sont en exécution, on ne peut pas prévoir quel thread va prendre la main
- S'ils sont en attente d'exécution, un thread de plus grande priorité prendra toujours la main avant un autre thread de priorité plus basse
- Cependant, il peut arriver exceptionnellement qu'un thread continue son exécution alors que des threads de priorité supérieure sont en attente d'exécution

Niveaux de priorité

- Un nouveau thread a la même priorité que le thread qui l'a créé
- En général, tous les threads ont la même priorité (NORM_PRIORITY - typiquement 5)
- Il faut faire appel à la méthode setPriority si on veut modifier cette priorité par défaut
- Le paramètre de setPriority doit être inclus entre MIN_PRIORITY (typiquement 1) et MAX_PRIORITY (typiquement 10)

Groupes de threads

- ThreadGroup représente un ensemble de threads, qui peut lui-même comprendre un threadGroup ; on a ainsi une arborescence de threadGroup
- On peut ainsi jouer en une seule instruction sur la priorité des threads du groupe ou sur le fait que les thread soient des démons ou non
- Pour créer un thread d'un groupe, on doit utiliser le constructeur de la classe Thread qui prend un groupe en paramètre ; par exemple :

```
ThreadGroup tg = new ThreadGroup("monGroupe");
Thread t = new Thread(tg, "monThread");
t.start();
```

ThreadGroup et Exceptions

- La classe ThreadGroup contient la méthode `uncaughtException(Thread t, Throwable e)`
- qui est exécutée quand un des threads du groupe est stoppé par une exception non saisie
- On peut redéfinir cette méthode pour faire un traitement spécial sur les autres threads
- Depuis Java 5, on peut aussi utiliser la méthode static `setDefaultUncaughtException` ou d'instance `setUncaughtException` de la classe Thread pour traiter d'une façon particulière les exceptions non attrapées

ThreadGroup et Exceptions

```
class MyThreadGroup extends ThreadGroup {
   public MyThreadGroup(String s) {
      super(s);
   }
   public void uncaughtException(Thread t, Throwable e) {
      // On met ici le traitement qui doit être exécuté
      // si un des threads du groupe reçoit une exception
      // non attrapée
      System.err.println("uncaught exception: " + e);
   }
}
```

Exceptions non traitées

```
class ExceptionHandler implements
   Thread.UncaughtExceptionHandler
{
   void uncaughtException(Thread t, Throwable e) {
      // On met ici le traitement qui doit être exécuté
      // si un thread reçoit une exception
      // non attrapée
      System.err.println("uncaught exception: " + e);
   }
}

// au début du main :
ExceptionHandler eh = new ExceptionHandler();
Thread.setDefaultUncaughtExceptionHandler(eh);
```

ThreadLocal

- Cette classe de java.lang permet d'associer un état (typiquement une variable statique privée) à chaque thread sans créer d'instances différentes
- Puisque la valeur n'est connue que d'un seul thread on évite ainsi les problèmes liés aux accès concurrents
- C'est aussi un moyen commode d'utiliser une valeur tout au long du déroulement d'un thread sans utiliser une variable connue par toutes les lignes de code concernées

Classe ThreadLocal<T>
Méthodes de base

- public void set(T valeur)
  met une valeur de type T dans l'objet de type ThreadLocal
- public T get()
  récupère la valeur rangée dans l'objet de type ThreadLocal
- public void remove()
  enlève la valeur de l'objet

Classe ThreadLocal<T>
Initialisation de la valeur

- protected T initialValue()
  permet de préciser la valeur renvoyée par get si aucun set n'a été appelé avant (renvoie null dans la classe ThreadLocal)
- Pour donner une valeur initiale à la variable locale, il suffit de redéfinir cette méthode dans une sous-classe de ThreadLocal

Exemple

```
public class C {
   private static final ThreadLocal tlSession = new
   ThreadLocal<Session>();
   ...
   public static Session getSession() {
      Session s = tlSession.get();
      if (s == null) {
         s = new Session();
         tlSession.set(s);
      }
      return s;
   }
}
```

En appelant C.getSession(), tout le code parcouru par un thread pourra travailler avec la même session

InheritableThreadLocal

- Classe fille de ThreadLocal
- La différence avec ThreadLocal est que la valeur de ces variables (si elle a une valeur) est passée aux threads créés par le thread courant
- La méthode

```
protected T childValue(T valeurDuPère)
```

peut être redéfinie dans une classe fille pour donner au thread fils une valeur calculée à partir de la valeur du thread père (par défaut, cette méthode renvoie la valeur passée en paramètre)

Threads et GUI

- On ne doit pas modifier une GUI depuis un autre thread que le thread AWT qui gère les évènements.
- Il faut utiliser des Runnable et les faire exécuter par ce thread en utilisant soit :
  - SwingUtilities.invokeLater(Runnable r) qui demande au thread de l'interface graphique d'exécuter dès que possible le code de la méthode run de r
  - SwingUtilities.invokeAndWait(Runnable r) qui bloque le thread COURANT jusqu'à ce que la méthode run de r ait été exécutée par le thread de l'interface graphique

Classes Timer et TimerTask

- Ces 2 classes du paquetage java.util permettent de lancer l'exécution de tâches à des intervalles choisis
- TimerTask a une méthode run() qui détermine la tâche à accomplir
- Timer détermine quand seront exécutées les tâches qu'on lui associe
- Dans les 2 classes des méthodes cancel() permettent d'interrompre une tâche ou toutes les tâches associées à un timer

Configuration du timer

- Un timer peut déclencher une seule exécution, ou déclencher des exécutions à des intervalles réguliers
- Pour l'exécution à des intervalles réguliers, le timer peut être configuré pour qu'il se cale au mieux par rapport au début des exécutions (méthode `scheduleAtFixedRate`), ou par rapport à la dernière exécution (méthode `schedule`)

Exemple de Timer

```
final long debut = System.currentTimeMillis();

TimerTask afficheTemps = new TimerTask() {
   public void run() {
      System.out.println(
         System.currentTimeMillis()- debut);
   }};

Timer timer = new Timer();
Timer.schedule(afficheTemps, 0, 2000);
```

démarre tout de suite

intervalle de 2 s entre les affichages

Timers et swing

- Pour utiliser un timer qui modifie l'affichage en Swing, il faut utiliser la classe javax.swing.Timer
- Cette classe utilise le thread de distribution des événements pour faire exécuter les tâches

Classes non synchronisées

- Si c'est possible, synchroniser explicitement les accès aux objets partagés en construisant des classes qui enveloppent les classes non synchronisées, avec des méthodes synchronisées (illustré par les collections)
- Sinon, synchroniser les accès au niveau des clients de ces classes ; c'est plus difficile et moins pratique
- On peut aussi s'arranger pour que les méthodes non synchronisées ne soient appelées que par un seul thread (illustré par Swing/AWT et le thread de distribution des événements : voir plus haut)

Exemple des collections

- Les nouvelles collections (Java 2) ne sont pas sûres vis-à-vis des threads (ArrayList, HashMap, HashSet...)
- Ça permet
  - d'améliorer les performances en environnement mono-thread
  - davantage de souplesse en environnement multi-threads : par exemple, pour ajouter plusieurs objets, on peut n'acquérir qu'une seule fois un moniteur

Collections synchronisées

- L'API des collections permet d'obtenir une collection synchronisée à partir d'une collection non synchronisée, par exemple avec la méthode :
  static Collections.synchronizedList(List<T> l)

Utiliser les collections non synchronisées

- Il faut synchroniser explicitement les modifications des collections :

```
private ArrayList<Float> al;
...
synchronized(al) {
      al.add(...);
      al.add(...);
}
```

- Avantage sur Vector : une seule acquisition de moniteur pour plusieurs modifications
- Cette technique n'est pas toujours possible si on n'a pas accès au code des classes qui utilisent les collections

Collections thread-safe

- Le paquetage java.util.concurrent de Java 5 (voir section suivante) contient des collections « thread-safe » qui offrent sécurité et performance en environnement multi-tâche :
  `ConcurrentHashMap`,
  `CopyOnWriteArrayList`,
  `CopyOnWriteArraySet`
- Il ne faut les utiliser que lorsqu'il y a des risques dus à des modifications par plusieurs threads (moins performantes que les collections « normales »)

Plan

- Introduction
- Notion de processus
- Notion de thread
- Créations de threads
- Synchronisation entre threads
- wait et notify
- Les différents états d'un thread
- Difficultés liées au parallélisme
- **Extensions de Java 5**

Paquetage java.util.concurrent

- Java 5 a ajouté un nouveau paquetage qui offre de nombreuses possibilités, avec de bonnes performances
- Le programmeur n'aura ainsi pas à réinventer la roue pour des fonctionnalités standard telles que les exécutions asynchrones, les gestions de collections utilisées par plusieurs threads (telles que les files d'attentes), les blocages en lectures/écritures, etc.
- Ce cours ne fait que survoler quelques possibilités offertes par cette API
- Pour plus de précisions se reporter à la javadoc de l'API et à ses tutoriels

Considérations techniques

- De nouvelles instructions ont été ajoutées aux processeurs pour faciliter leur utilisation en environnement multi-cœurs
- Par exemple, dans les processeurs Intel, l'instruction « compare-and-swap » (CAS), en une seule opération atomique, (la main ne peut être donnée à un autre thread pendant son exécution) compare la valeur d'un emplacement mémoire à une valeur donnée et, selon le résultat de la comparaison, modifie l'emplacement mémoire
- La JVM a été adaptée pour bénéficier de ces nouvelles instructions
- La nouvelle API de java.util.concurrent s'appuie sur ces nouveautés pour améliorer la gestion du multitâche, en particulier pour améliorer les performances, par rapport à l'utilisation de synchronized

Problèmes avec Runnable

- La méthode run ne peut renvoyer une valeur et elle ne peut lancer d'exceptions contrôlées par l'appelant
- Les valeurs calculées par la méthode run doivent donc être récupérées par une méthode de type getValeur() et les exceptions contrôlées doivent être attrapées et signalées d'une façon détournée au code qui a lancé run

Callable

- Java 5 fournit un nouveau cadre pour faciliter et enrichir les possibilités lors du lancement de tâches en parallèle
- L'interface Callable améliore Runnable
- Future facilite la récupération des valeurs calculées par un autre thread
- Executor et Executors découplent la soumission de tâches et leur exécution, et offrent une gestion de pools de threads

Interface Callable<V>

- Elle représente une tâche à exécuter (par ExecutorService), qui renvoie une valeur de type V
- Elle définit une seule méthode qui exécute la tâche :

```
V call() throws Exception
```

- La classe Executors contient des méthodes pour envelopper les anciennes interfaces `Runnable`, `PrivilegedAction` et `PrivilegedExceptionAction`, et les transformer en `Callable`

Interface Executor

- Représente un objet qui exécute des tâches qu'on lui a soumis
- La soumission d'une tâche est effectuée par la seule méthode de l'interface :
  `void execute(Runnable tâche)`
- Les tâches soumises à un exécuteur ne sont pas nécessairement exécutées par des threads mais c'est le plus souvent le cas
- Au contraire de la classe Thread qui lance immédiatement un thread avec la méthode start(), un exécuteur peut choisir le moment et la manière d'exécuter les tâches qui lui ont été soumises

Interface Future<V>

- Représente le résultat d'une tâche exécutée en parallèle
- Les méthodes de l'interface permettent de retrouver le résultat du travail (get), d'annuler la tâche (cancel) ou de savoir si la tâche a terminé son travail (isDone) ou a été annulée (isCancelled)
- Méthodes get :
  - V get()
    récupère le résultat du travail ; bloque si le travail n'est pas encore terminée
  - V get(long délai, TimeUnit unité)
    idem get() mais ne bloque qu'au plus le temps du délai passé en paramètre sinon on récupère une TimeoutException
  - Exemple :

```
float x = future.get(50L, TimeUnit.SECONDS);
```

Interface ExecutorService

- ExecutorService est une sous-interface de Executor
- Elle ajoute des méthodes pour gérer la fin de l'exécution des tâches et récupérer un Future comme résultat de la soumission d'une tâche
- Méthodes submit pour soumettre une tâche (Callable ou Runnable) et récupérer un Future qui représente l'exécution de la tâche :

```
<T> Future<T> submit(Callable<T> task)
```

- Les méthodes avec Runnable en paramètre renvoient un Future dont la méthode get renvoie la valeur null ou une certaine valeur passée en paramètre après la fin de l'exécution («à la main» car un Runnable ne renvoie pas de valeur)

Interface ExecutorService

- Méthodes invokeAll et invokeAny pour exécuter une collection de tâches (Callable) et renvoyer le résultat (Future) de toutes les tâches (InvokeAll) ou d'une seule qui s'est exécutée correctement (invokeAny) ; un temps maximum d'exécution peut être passé en paramètre et les tâches non terminées dans le temps imparti sont arrêtées
- Par exemple :

```
<T> List<Future<T>> invokeAll(
    Collection<? extends Callable<T>> tasks)
    throws InterruptedException
```

Interface ExecutorService

- Méthodes shutdown et shutdownNow pour arrêter l'activité de l'exécuteur, d'une façon
  - « civilisée » (avec shutdown les tâches en cours se terminent normalement ; seules les tâches en attente d'exécution ne seront pas exécutées)
  - ou brutale (avec shutdownNow les tâches en cours d'exécution sont stoppées)
- List<Runnable> shutdownNow()
  - renvoie la liste des tâches qui étaient en attente d'exécution

Interface ExecutorService

- La méthode isShutdown indique si l'exécuteur a été arrêté
- La méthode isTerminated indique si toutes les tâches en cours d'exécution au moment de l'arrêt ont terminé leur exécution
- La méthode awaitTermination bloque en attendant la fin de l'exécution des tâches après un shutdown (un délai maximum d'attente est passé en paramètre)

Interface ScheduledExecutorService

- Hérite de ExecutorService
- 4 méthodes « schedule » qui créent un exécuteur et lancent les exécutions
- Permet de donner un délai avant l'exécution ou de faire exécuter périodiquement une tâche (comme les timers)
- On peut indiquer une périodicité moyenne pour les exécutions ou un intervalle fixe entre la fin d'une exécution et le début de la prochaine exécution

Classe Executors

- Contient des méthodes static utilitaires pour les interfaces du paquetage : Executor, ExecutorService, Callable,...
- Les méthodes callable renvoient un Callable qui enveloppe un Runnable, PrivilegedAction ou PrivilegedExceptionAction
- privilegedCallable(Callable) est réservée à un appel dans la méthode doPrivileged de la classe AccessController ; elle renvoie un Callable dont les autorisations ne dépendent pas du contexte d'appel (voir la documentation du paquetage java.security)

Les fabriques de threads/executors

- Méthodes de Executors qui commencent par « new »
- Une fabrique de Thread (java.util.ThreadFactory) peut être passée en paramètre si on veut des types spéciaux de threads (par exemple avec une certaine priorité, ou d'une certaine sous-classe de Thread)
- newCachedThreadPool : crée un pool de threads réutilisables
  - Les threads non utilisés depuis 1 minute sont supprimés
  - De nouveaux threads sont créés si nécessaire
- newFixedThreadPool : crée un pool contenant un nombre fixe de threads
  - Nombre de threads passé en paramètre

Les fabriques de threads/executors

- newSingleThreadExecutor : crée un exécuteur qui exécutera les tâches les unes après les autres
- newScheduledThreadPool : crée un pool de threads qui peuvent exécuter des commandes après un certain délai ou périodiquement
- newSingleThreadScheduledExecutor : crée un exécuteur qui peut exécuter des commandes après un certain délai ou périodiquement, une commande à la fois

Exemple d'utilisation de pools

```
ExecutorService pool = Executors.newFixedThreadPool(10);

for(int i = 0 ; i < 100 ; i++) {
     Runnable r = new Runnable() {
        public void run() {
           ...
        }
     };

     pool.execute(r);
} // à tout moment il n'y aura que 10 threads en parallèle
```

Exemple pools de Callable

```
ExecutorService pool = Executors.newFixedThreadPool(10);

Callable<Integer> c = new Callable<Integer>() {
   public Integer call() {
      ...
   }
};

Future<Integer> f = pool.submit(c);
...
int v = f.get();
```

Classe ConcurrentHashMap

- Au contraire de HashTable, les lectures ne bloquent ni les autres lectures ni même les mises à jour
- Une lecture reflète l'état de la map après la dernière mise à jour complètement terminée (celles en cours ne sont pas prises en compte)
- Les méthodes sont à peu près les mêmes que celles de HashMap

Classe CopyOnWriteArrayList

- Variante « thread-safe » de ArrayList dans laquelle toutes les opérations de mise à jour sont effectuées en faisant une copie du tableau sous-jacent
- Cette variante n'est intéressante que lorsque les parcours de la liste sont bien plus fréquents que les mises à jour
- Les itérateurs reflètent la liste au moment où ils ont été créés ; ils ne peuvent modifier la liste
- Idem pour CopyOnWriteArraySet

Interfaces pour files d'attente

- L'interface java.util.Queue, introduite par le J2SE 1.5, est semblable à List mais correspond à une file d'attente : elle ne permet des ajouts qu'à la fin et des suppressions qu'au début
- Cette contrainte permet plus d'optimisation que pour les implémentations de List
- L'interface java.util.concurrent.BlockingQueue
  - hérite de Queue et ajoute une méthode qui se bloque lorsqu'elle veut supprimer un élément dans une file vide (take) et une méthode qui se bloque lorsqu'elle ajoute un élément dans une file pleine (put)
  - Elle permet d'implémenter des files pour des producteurs et des consommateurs

Synchronisation des threads

- La nouvelle API de concurrence offre des objets pour faciliter le codage des programmes qui utilisent plusieurs traitements en parallèle
- La classe Semaphore implémente un sémaphore classique
- Les classes CountDownLatch et CyclicBarrier représentent des structures un peu plus complexes qui facilitent la synchronisation de plusieurs threads

Semaphore

- Un sémaphore gère des ressources limitées
- La méthode
  `public void acquire()`
  demande une ressource et bloque jusqu'à ce qu'il y en ait une disponible ; lorsque on sort de la méthode, une ressource en moins est disponible via le sémaphore
- La méthode
  `public void release()`
  rend une ressource ; lorsque on sort de la méthode, une ressource de plus est disponible via le sémaphore
- Plusieurs variantes des méthodes de base acquire et release sont disponibles (pour acquérir ou rendre plusieurs ressources en même temps ou pour ne pas se bloquer en attente de ressources)

CountDownLatch et CyclicBarrier

- Les classes CountDownLatch et CyclicBarrier facilitent la décomposition d'un traitement en plusieurs sous-traitements parallèles
- A certains points d'exécution le traitement ne peut se poursuivre que si les sous-traitements ont terminé une partie de leur travail

CyclicBarrier

- Représente une barrière derrière laquelle n threads (n est un paramètre du constructeur) qui représentent les sous-traitements, attendent par la méthode barriere.await()
- Quand les n threads y sont arrivés, la barrière « se lève » et les threads continuent leur exécution
- La barrière est cyclique car elle se rabaisse ensuite, et les threads y sont à nouveau bloqués
- Un constructeur permet de passer en paramètre un Runnable qui sera exécuté juste avant que la barrière ne se relève

CountDownLatch

- Rôle similaire à une barrière mais il ne peut servir qu'une seule fois et le fonctionnement est différent : l'arrivée au point d'attente est dissociée de l'attente elle-même
- Le constructeur prend en paramètre un nombre n
- Les sous-traitements peuvent décrémenter ce nombre par cdl.countDown()
- Des threads peuvent aussi attendre que n soit égal à 0 par cdl.await()
- Peut être utilisé pour le même usage qu'une barrière : les sous-traitements appellent countDown et await juste après
- Mais aussi comme un starter : n est initialisé à 1 et tous les threads commencent par attendre par await ; ensuite un thread « starter » appelle countDown pour faire démarrer tous les threads

D’autres cas de « happens-before »

- L’ajout d’un objet dans une collection « thread-safe » de cette API happens-before un accès ou une suppression de l’objet dans la collection
- Les actions effectuées pour le calcul d’un Future happens-before la récupération de la valeur du Future
- Une action exécutée par un thread avant une attente à une barrière cyclique (await) happens-before l’action (optionnelle) exécutée par la barrière au moment où elle se lève, qui elle-même, happens-before une action qui suit await

Remerciements

- Ce document est basé sur la Version 3.3 du support sur les threads de Richard Grin (Université de Nice - Sophia Antipolis)
- Ce document utilise aussi la documentation de l'API Java (versions 1.5 et suivantes) et ses tutoriels