
Les tâches et la synchronisation en langage Java

Les threads, les verrous, les sémaphores et les moniteurs en Java

D'après les cours de D. Genthial et B. Caylux



Généralités

- En java une tâche est un processus léger :
 - ➔ il partage l'espace mémoire de l'application qui l'a lancé.
 - ➔ il possède sa propre pile d'exécution.
- En Java, un processus léger peut être créé par :
 1. dérivation de la classe *Thread*.
 2. implémentation de l'interface *Runnable*.
- Documentation officielle de Sun :
<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html>



- Un *thread* est constitué essentiellement d'une méthode *run ()*
- Cette méthode est appelée par la méthode *start()*
- Un *Thread* vit jusqu'à la fin de l'exécution de sa méthode *run*.
 - ➔ Un thread peut être endormi par *sleep(nb de ms)*
 - ➔ Un thread peut passer son tour par *yield()*
 - ➔ Un thread peut se mettre en attente par *wait()*



Création de Thread par spécialisation de la classe Thread

Définition du thread

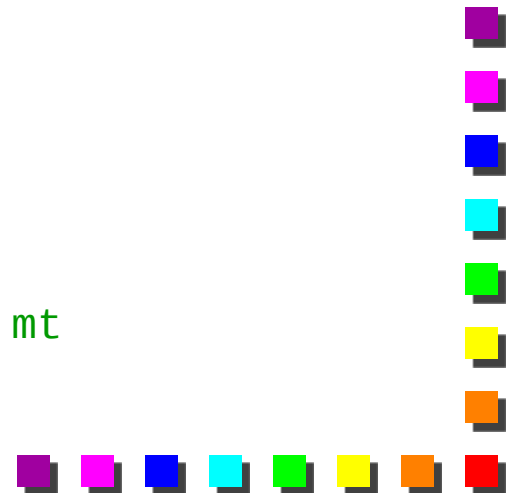
```
class MonThread extends Thread{  
    MonThread(. . .){  
  
    }  
    public void run(){  
        . . .  
    }  
    . . .  
}
```

Constructeur

Algorithme décrivant
le comportement du thread

Démarrage du thread

```
MonThread mt = new MonThread ();  
mt.start() // exécute la méthode run de mt
```



Création de Thread par spécialisation de la classe Thread

- Exemple

```
class MaPremiereTache extends Thread {  
    public MaPremiereTache() {  
        super();  
    }  
    public void run() {  
        System.out.println("Bonjour le monde !");  
    }  
}
```

...

```
MaPremiereTache t = new MaPremiereTache ();  
t.start(); // exécute la méthode run de t
```



Création de Thread par implémentation de l'interface Runnable

L'interface Runnable

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

Utilisation

```
public class MaPremiereTache implements Runnable {
    ...
    public void run() {
        System.out.println("Bonjour le monde !");
    }
}
```



Création de Thread par implémentation de l'interface Runnable

- **Création d'une tâche à partir d'un objet Runnable**

```
Runnable maTache = new MaPremiereTache(...);  
Thread leThread = new Thread(maTache);  
leThread.start(); // démarrage
```

- **ou simplement (tâche anonyme)**

```
Runnable maTache = new MaPremiereTache(...);  
new Thread(maTache).start();
```

- **ou encore...**

```
MaPremiereTache maTache = new MaPremiereTache(...);  
new Thread(maTache).start();
```



Méthodes statiques

- Elles agissent sur le thread appelant

- ➔ `Thread.sleep(long ms)`

- bloque le thread appelant pour la durée spécifiée;

- ➔ `Thread.yield()`

- Le thread appelant relâche le CPU au profit d'un thread de même priorité;

- ➔ `Thread.currentThread()`

- retourne une référence sur le thread appelant;



Méthodes d'instances

MonThread p = **new** MonThread ();

- p.start() démarre le thread p;
- p.isAlive() détermine si p est vivant (ou terminé);
- p.join() bloque l'appelant jusqu'à ce que p soit terminé;
- p.setDaemon() attache l'attribut « daemon » à p;
- p.setPriority(int pr) assigne la priorité pr à p;
- p.getPriority() retourne la priorité de p;
- ...(beaucoup d'autres)



Synchronisation

- **Blocage d'une tâche**

`void wait();`

→ sur un objet quelconque

- **Réveil d'une tâche**

`void notify();`

→ sur l'objet cause du blocage

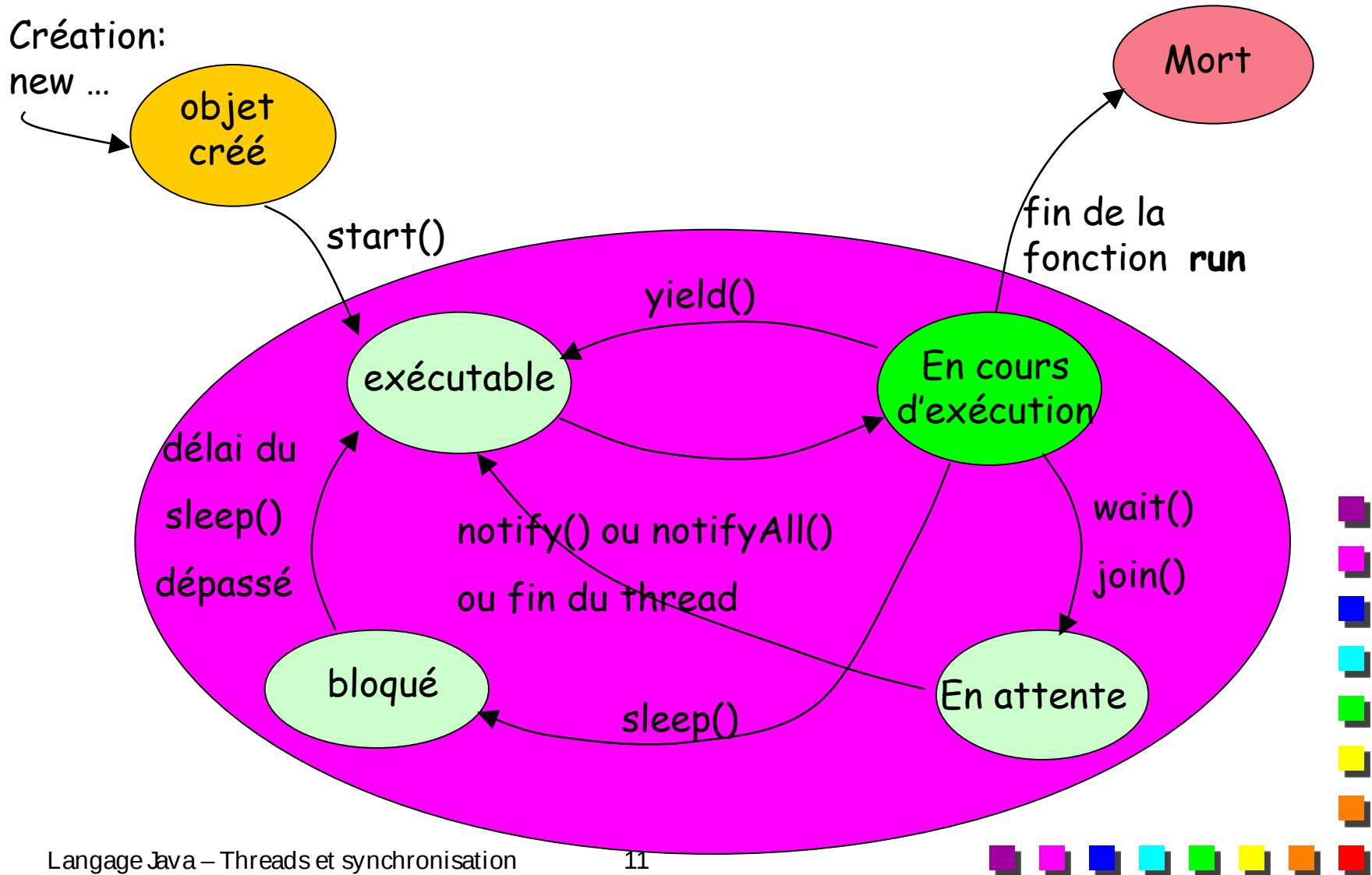
- **Réveil de toutes les tâches**

`void notifyAll();`

→ sur l'objet cause du blocage



Cycle de vie d'un thread



Arrêt d'une tâche

- Un *Thread* dans l'état bloqué ou en attente peut être arrêté par *interrupt()*

```
class MonThread extends Thread{
    . . .
    public void run(){
        try{
            while (true){
                . . .
                sleep(100);
            }
        }catch(InterruptedException ie){
            . . .
        }
    }
    . . .
}
```

```
MonThread t = new MonThread ();
t.start();

. . .
t.interrupt();
```



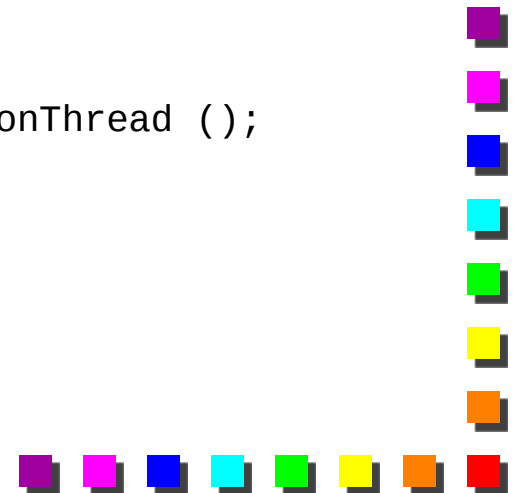
Arrêt d'une tâche

- L'itération peut être contrôlée par un booléen et arrêtée par une fonction qui rend vrai le booléen.

```
class MonThread extends Thread{
    boolean arret = false;

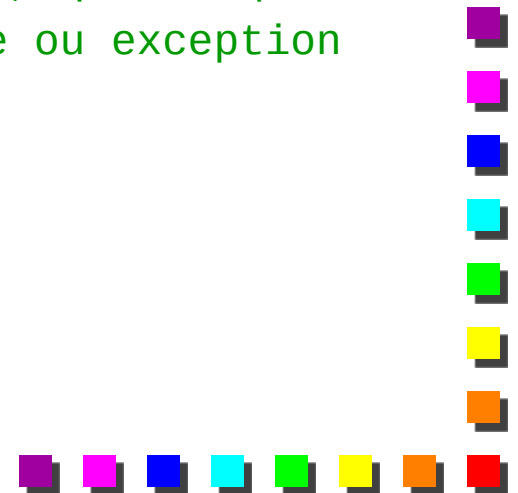
    public void run(){
        try{
            while(!arret){
                . . .
            }
        }catch(InterruptedException ie){
            . . .
        }
    }
    public void stop (){
        arret = true;
    }
    . . .
}

MonThread t = new MonThread ();
t.start();
. . .
t.stop();
```



Arrêt d'une tâche

```
public void run() {  
    try {  
        // Corps de la tâche  
        ...  
    } catch (InterruptedException e) {  
        // Terminer la tâche  
        ...  
        return;  
    } finally {  
        // Code exécuté avant la fin de la tâche, quelle que soit  
        // la cause qui l'a arrêtée (fin normale ou exception  
        // ou erreur)  
        ...  
    }  
}
```



Jointure de threads

- Un *thread* peut avoir à attendre la fin d'un ou plusieurs *threads* pour démarrer : *join()*

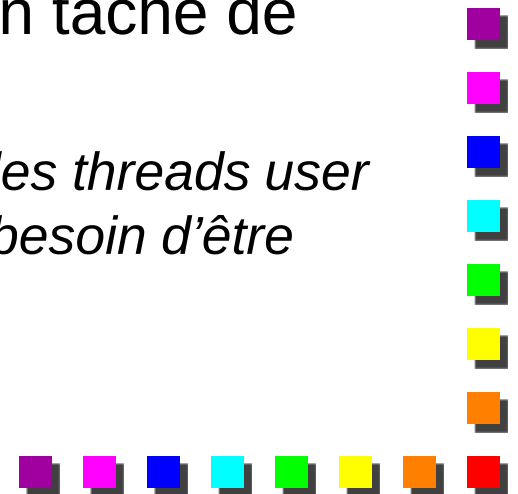
```
class MonThread extends Thread{
    Thread precedent;
    public MonThread( Thread p){
        précédent = p;
    }

    public void run(){
        try{
            precedent.join(); // le thread reste bloqué tant que
                             // precedent n'est pas terminé
            . . .
        }catch(InterruptedException ie){
            . . .
        }
    }
}
```



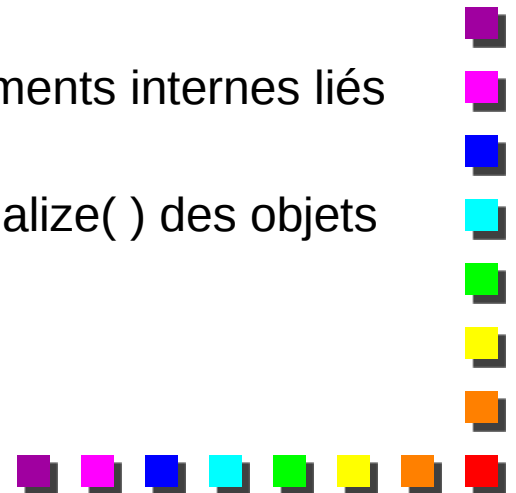
Priorité des threads

- Les Threads ont une priorité qui va de
 - ➔ Thread.MIN_PRIORITY (1)
 - ➔ Thread.NORM_PRIORITY (5)
 - ➔ Thread.MAX_PRIORITY (10)
- *int getPriority()* ;
- *void setPriority(int p)* ;
- *void setDaemon(boolean on)* permet de rendre un *thread* démon : *thread* qui s'exécute en tâche de fond.
- *Un programme Java se termine lorsque tous les threads user sont terminés (les threads daemon n'ont pas besoin d'être terminés).*



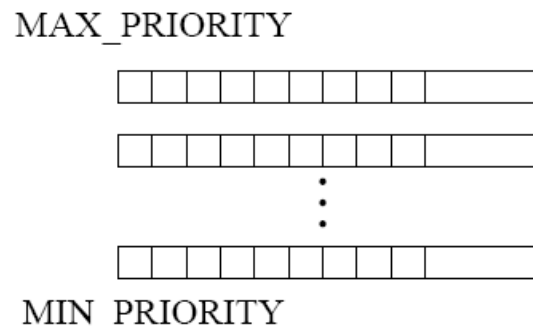
Threads de la JVM

- En plus des threads du programme Java, une machine virtuelle Java exécute des threads dédiés à des tâches de gestion :
 - ➔ **thread oisif** (priorité 0): exécuté si aucun autre thread ne s'exécute;
 - ➔ **ramasse-miettes** (garbage collector, priorité 1): récupère les objets qui ne sont plus accessibles (référéncés). Ce thread ne s'exécute que lorsqu'aucun autre thread (à part le thread oisif) ne s'exécute;
 - ➔ **gestionnaire d'horloge**: gère tous les événements internes liés au temps;
 - ➔ **thread de finalisation**: appelle la méthode `finalize( )` des objets récupérés par le ramasse-miettes.



Ordonnancement des tâches : la théorie

- Les threads exécutable se trouvent dans les queues de différentes priorités, gérées par le run-time de Java:
 - ➔ si plus d'un thread *exécutable*, Java choisit le thread de plus haute priorité;
 - ➔ si plus d'un thread de plus haute priorité, Java choisit un thread arbitrairement;
 - ➔ un thread *en cours d'exécution* perd le CPU au profit d'un thread de plus haute priorité qui devient *exécutable*;

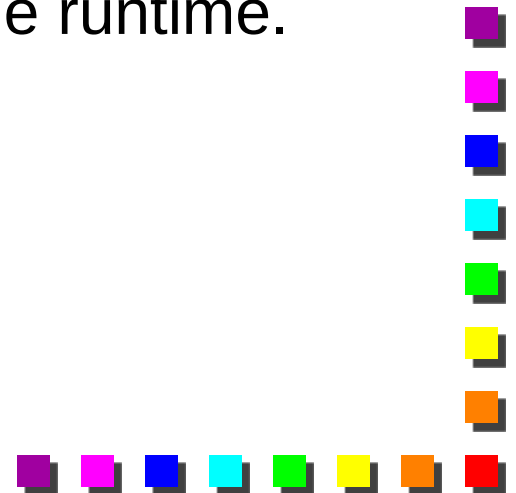


- par défaut, un thread a la même priorité que le thread qui l'a créé;
- la priorité peut être changée en appelant **Thread.setPriority()**.



Ordonnancement des tâches : la réalité

- La politique d'allocation du processeur aux threads de même priorité n'est pas fixée par le langage.
 - ➔ Elle peut être *avec* ou *sans* préemption !...
- Pas de traitement cohérent de la priorité:
exemple: la priorité n'est pas utilisée par la méthode **notify()** pour choisir le processus à débloquent;
- La spécification du langage n'exige pas que les priorités soient prises en compte par le runtime.



Synchronisation

- Comment empêcher plusieurs *threads* d'accéder en même temps à un objet ?
- Chaque objet Java possède un **verrou** d'exclusion mutuelle.

➔ Synchronisation d'un bloc d'instructions

```
synchronized (objet){  
    . . . // Aucun autre thread ne  
    . . . // peut accéder à objet  
}
```

➔ Synchronisation d'une méthode

```
synchronized void f(){  
    . . . // Aucun autre thread ne  
    . . . // peut accéder à l'objet  
    . . . // pour lequel est appelé la méthode f  
}
```



Rappel : les verrous

```
Type verrou = structure  
    ouvert : booléen;  
    attente : liste de tâches;  
fin verrou;
```

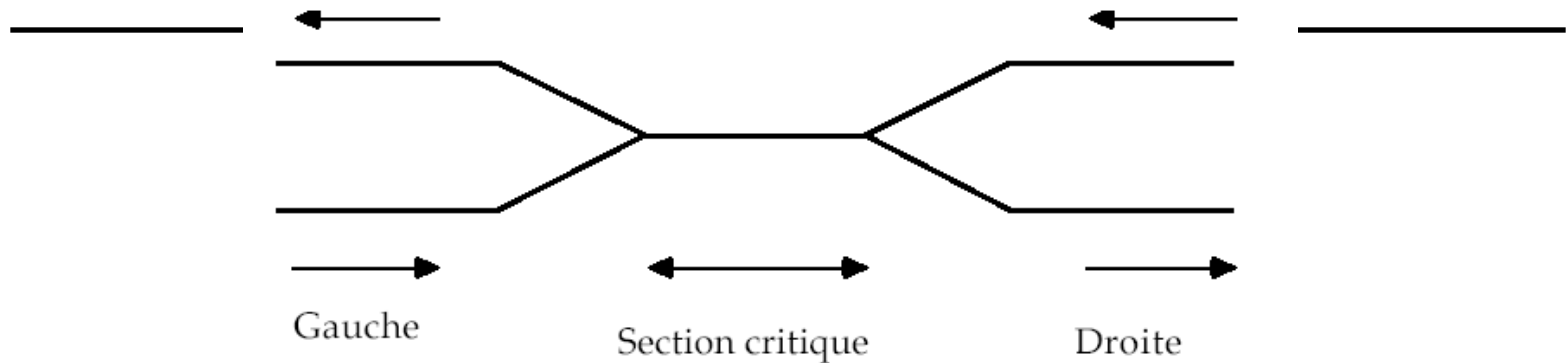
```
procédure verrouiller (modifié v : verrou);  
début  
    si v.ouvert alors  
        v.ouvert ← faux  
    sinon  
        Ajouter la tâche dans la liste v.attente  
        dormir  
    finsi;  
fin verrouiller;
```

```
procédure init_verrou (modifié v : verrou);  
début  
    v.ouvert ← vrai;  
    v.attente ← vide;  
fin init_verrou;
```

```
procédure déverrouiller (modifié v : verrou);  
début  
    si v.attente ≠ vide alors  
        Sortir la première tâche de v.attente,  
        La réveiller  
    sinon  
        v.ouvert ← vrai  
    finsi;  
fin déverrouiller;
```



Exemple : voie unique



Tâche TrainGauche (v : Voie);
début

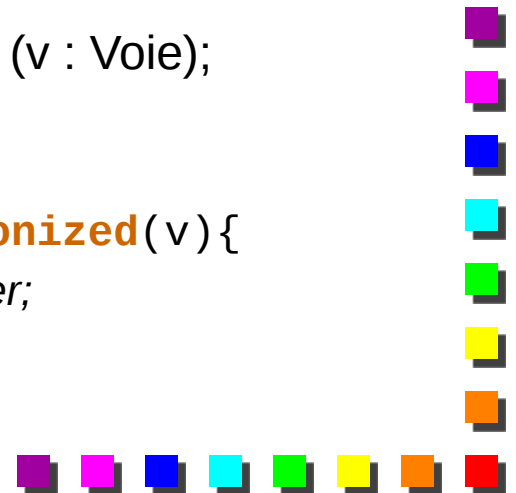
```
rouler;  
synchronized(v){  
    Passer;  
}  
rouler;
```

fin TrainGauche;

Tâche TrainDroit (v : Voie);
début

```
rouler;  
synchronized(v){  
    Passer;  
}  
rouler;
```

fin TrainDroit;



Exemple de bloc *synchronized*

Comment garantir une exécution atomique des commandes suivantes ?

```
System.out.print (new Date ( )) ;  
System.out.print ( " : " ) ;  
System.out.println (...) ;
```

(les méthodes print et println de la classe Printstream sont synchronisées)

```
Synchronized (System.out){  
    System.out.print (new Date ( ) ) ;  
    System.out.print ( " : " ) ;  
    System.out.println (...) ;  
}
```

lorsqu'une méthode synchronized est appelée, le ***thread appelant*** doit obtenir le verrou associé à l'objet



Synchronisation

- La synchronisation par verrou peut provoquer des « interblocages » (dead locks) amenant l'arrêt définitif de un ou plusieurs *threads*.

```
class MonThread1 extends Thread{  
    public void run(){  
        synchronized(a){  
            . . .  
            synchronized(b){  
                . . .  
            }  
        }  
    }  
}
```

```
class MonThread2 extends Thread{  
    public void run(){  
        synchronized(b){  
            . . .  
            synchronized(a){  
                . . .  
            }  
        }  
    }  
}
```



Les verrous Java : utilisation (1)

- **Synchronisation (verrouillage) sur un objet**

```
public class Compte {  
    private double xSolde;  
    public Compte(double DépôtInitial) {  
        xSolde = DépôtInitial;  
    }  
    public synchronized double solde() {  
        return xSolde;  
    }  
    public synchronized void dépôt(double montant) {  
        xSolde = xSolde + montant;  
    }  
}
```

➔ Solde **et** Dépôt **sont exécutées en exclusion mutuelle sur un objet donné** (xSolde)



Les verrous Java : utilisation (2)

```
public class Chèques
    implements Runnable {

    private Compte xCompte;

    public Chèques(Compte c) {
        xCompte = c;
    }

    public void run() {
        double montant;
        // Lecture du montant
        . . .
        xCompte.dépôt(montant);
    }
}
```

```
public class Liquide
    implements Runnable {

    private Compte xCompte;

    public Liquide(Compte c) {
        xCompte = c;
    }

    public void run() {
        double montant;
        // Lecture du montant
        . . .
        xCompte.dépôt(montant);
    }
}
```



Les verrous Java : utilisation (3)

```
public class Banque {  
  
    public static void main(String [] arg) {  
        Compte monCompte = new Compte(2000.00);  
  
        Chèques c = new Chèques(monCompte);  
        Liquide l = new Liquide(monCompte);  
  
        new Thread(c).start();  
        new Thread(l).start();  
    }  
}
```



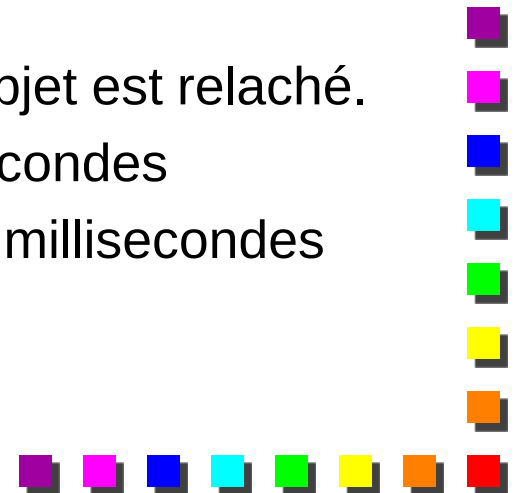
Synchronisation

La méthode `wait()` est définie dans la classe `Object`.

- On ne peut utiliser cette méthode que **dans un code synchronisé**.
- Elle provoque l'attente du thread en cours d'exécution.
- Elle doit être invoquée sur l'instance verrouillée.

```
synchronized(unObjet){  
    ...  
    try{  
        unObjet.wait(); // unObjet bloque l'exécution du thread  
    }catch(InterruptedException ie){  
        ...  
    }  
}
```

- Pendant que le thread attend, le verrou sur `unObjet` est relâché.
 - ➔ `wait(long ms)` attend au maximum `ms` millisecondes
 - ➔ `wait(long ms, int ns)` attend au maximum `ms` millisecondes + `ns` nanosecondes



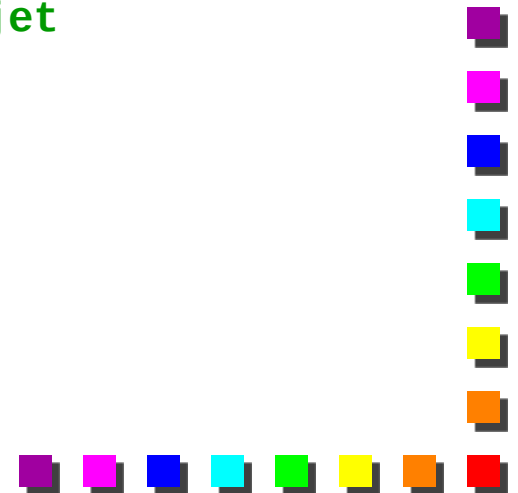
Synchronisation

- La méthode *notify()* permet de débloquent un thread.

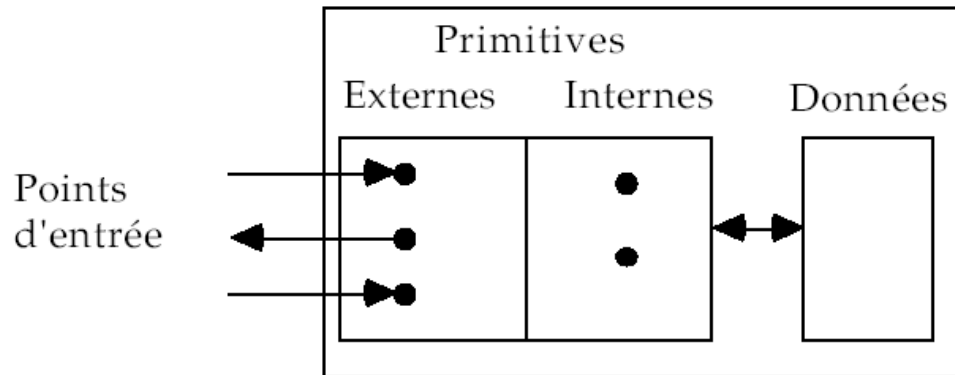
```
synchronized(unObjet){  
    ...  
    unObjet.notify(); // débloquent un thread bloqué par unObjet  
    ...  
}
```

- La méthode *notifyAll()* permet de débloquent tous les threads.

```
synchronized(unObjet){  
    ...  
    unObjet.notifyAll(); // débloquent tous les threads  
                        // bloqués par unObjet  
    ...  
}
```



Rappel : les moniteurs



Module avec des propriétés particulières

- on n'a accès qu'aux primitives externes (publiques), pas aux variables (privées) ;
- les procédures sont exécutées en exclusion mutuelle et donc les variables internes sont manipulées en exclusion mutuelle.
- on peut bloquer et réveiller des tâches. Le blocage et le réveil s'exprime au moyen de *conditions*.



Rappel : les moniteurs

Conditions = variables, manipulables avec :

procédure attendre (**modifié c : condition**);

début

Bloque la tâche qui l'exécute et l'ajoute dans la file d'attente associée à c

Fin attendre;

fonction vide (**c : condition**) → un booléen;

début

si la file d'attente associée à c est vide alors

renvoyer(VRAI)

sinon

renvoyer(FAUX)

finsi

Fin vide;

procédure signaler (**modifié c : condition**);

début

si non vide(c) alors

extraire la première tâche de la file associée à c
 la réveiller

finsi

Fin signaler;

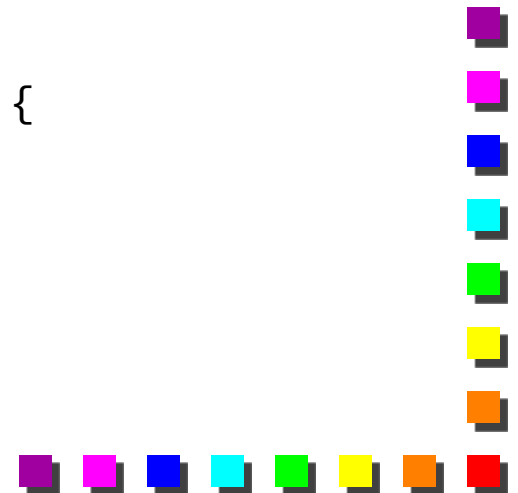


Moniteurs en Java (1)

- En java on dispose de moniteurs simplifiés
- Définition d'une classe

```
public class Moniteur {
    private type uneVariable;

    public synchronized unePrimitive(. . .)
        throws InterruptedException {
        . . .
        if (. . .)
            wait(); // le thread se met en attente sur la cible
        . . .
    }
    public synchronized unePrimitive(. . .) {
        . . .
        notify(); // réveil de la cible
        . . .
    }
}
```



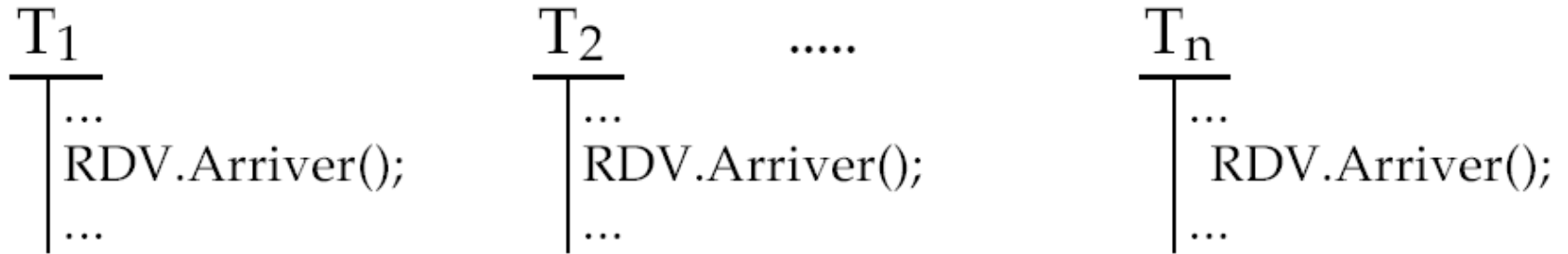
Moniteurs en Java (2)

- Implémentation (réduite) des conditions

```
public class Moniteur {  
    private type uneVariable;  
    private Object uneCondition = new Object();  
    public unePrimitive(. . .) // pas synchronized  
        throws InterruptedException {  
        . . .  
        if (. . .)  
            synchronized (uneCondition) {uneCondition.wait();}  
        . . .  
    }  
    public synchronized unePrimitive(. . .) {  
        . . .  
        synchronized (uneCondition) {uneCondition.notify();}  
        . . .  
    }  
}
```



Rendez-vous avec un moniteur (rappel)



Moniteur **RDV**;

Point d'entrée

Arriver

Lexique

n : un entier constant = ??

i : un entier;

tous_là : une condition;

Début {Initialisations}

i $\leftarrow$ 0;

Fin **RDV**;

procédure Arriver();

début

i $\leftarrow$ i + 1;

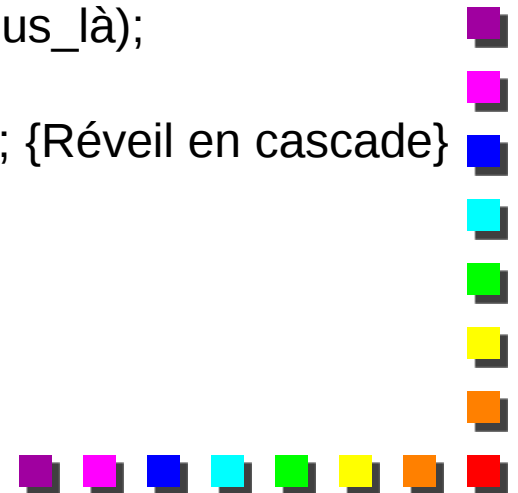
si i < n **alors**

attendre(tous_là);

finsi

signaler(tous_là); {Réveil en cascade}

fin Arriver;



Rendez-vous en Java : moniteur simplifié

```
class RendezVous {  
    private int nbattendus;           // nombre attendus  
    private int i;                   // nombre arrivés  
  
    // constructeur  
    public RendezVous(int n) {  
        nbattendus = n;  
        i = 0;  
    }  
  
    public synchronized void arriver() throws  
        InterruptedException {  
        i++;  
        if (i < nbattendus) {  
            wait();  
        };  
        notify();  
    }  
}
```



Rendez-vous en Java avec condition

```
class RendezVous {  
  
    private int nbattendus;           // nombre attendus  
    private int i;                   // nombre arrivés  
    private Object tousla = new Object(); // condition  
  
    // constructeur  
    public RendezVous(int n) {  
        nbattendus = n;  
        i = 0;  
    }  
  
    public void arriver() throws InterruptedException {  
        i++;  
        if (i < nbattendus) {  
            synchronized(tousla) {tousla.wait();}  
        }  
        synchronized(tousla) {tousla.notify();}  
    }  
}
```



Les Sémaphores en Java

- Il est inutile de définir des sémaphores en Java, car les mécanismes fournis par le langage pour la synchronisation et le partage de ressources sont suffisamment puissants pour résoudre tous les problèmes qu'on peut avoir à résoudre avec des sémaphores.
- Il est cependant intéressant de montrer avec quelle facilité ces mécanismes peuvent être utilisés pour définir des sémaphores.



Rappel : Sémaphores

- Mécanisme de synchronisation entre processus.
- Un sémaphore S est une variable entière, manipulable par l'intermédiaire de deux opérations :

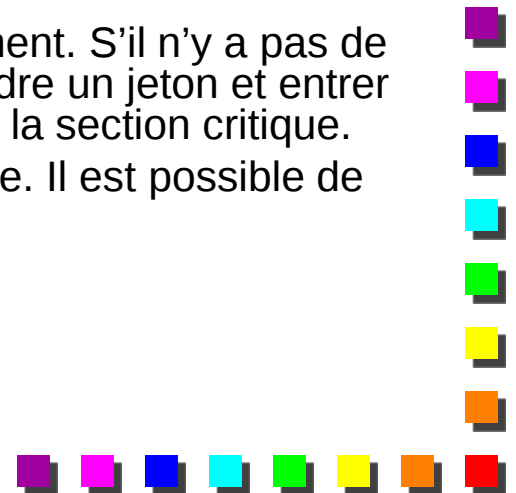
P (proberen) et V (verhogen)

acquire release

acquire (S) $S \leftarrow (S - 1)$
si $S < 0$, alors mettre le processus en attente ;
sinon on poursuit l'exécution

release (S) $S \leftarrow (S + 1)$; *réveil d'un processus en attente.*

- **acquire (S)** correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la section critique alors attendre, sinon prendre un jeton et entrer dans la section (S), puis rendre son jeton à la sortie de la section critique.
- Chaque dépôt de jeton **release (S)** autorise un passage. Il est possible de déposer des jetons à l'avance



Réalisation de la classe Semaphore

```
public class Semaphore {
    private int xVal;    // la valeur du sémaphore
    // En java, l'initialisation du sémaphore vient naturellement
    // dans le constructeur

    Semaphore (int valeur_initiale)
    { xVal = valeur_initiale; }

    // Acquire : noter l'indispensable synchronized peut
    // générer une InterruptedException (wait).
    public synchronized void acquire() throws InterruptedException {
        xVal = xVal - 1;
        if (xVal < 0) wait();
    }

    // Release
    public synchronized void release() {
        xVal = xVal + 1;
        if (xVal <= 0) notify();
    }
}
```



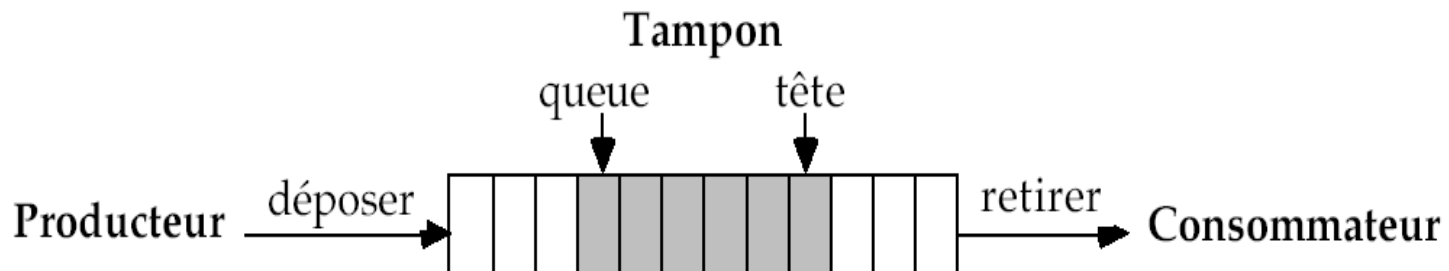
Exemple : Producteur/Consommateur

Partage de mémoire commune + exclusion mutuelle

Modèle général : producteur / consommateur

Producteur → **Données** → Consommateur

- **La communication est en général asynchrone** : une des deux tâches travaille en général plus vite que l'autre.
- **Pour limiter les effets de l'asynchronisme, on insère un tampon** entre le producteur et le consommateur :



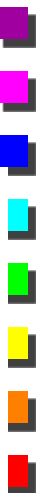
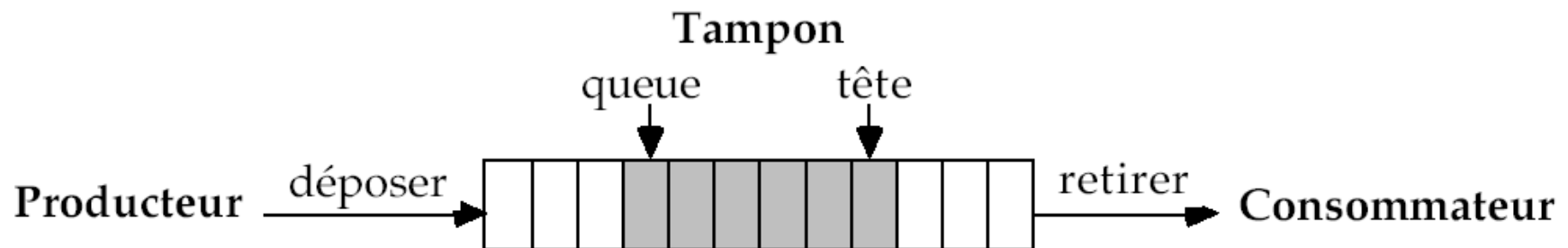
Rappel: Producteur/Consommateur avec tampon

Producteur

```
tantque vrai faire  
    produire(m);  
    Tampon.déposer(m);  
fintantque
```

Consommateur

```
tantque vrai faire  
    Tampon.retirer(m);  
    consommer(m);  
fintantque
```



Rappel : Producteur/Consommateur avec tampon

Moniteur Tampon; {Suppose Taille et Message connus}

Points d'entrée

déposer, retirer;

Lexique

nb : entier; {Nbre d'élts. dans le tampon (0..Taille)}

non_plein, non_vide : conditions; {Pour le blocage et le réveil}

tamp : tableau[0..Taille-1] de Messages;

tête, queue : entier; {0..Taille-1}

procédure **déposer** (consulté m:Message)

début

si **nb** = **Taille** alors

attendre(**non_plein**)

finsi

{Ajouter m au tampon}

nb ← **nb** + 1;

tamp[queue] ← m;

queue ← (**queue**+1) mod **Taille**;

signaler(**non_vide**);

fin **déposer**;

procédure **retirer** (consulté m:Message)

début

si **nb** = 0 alors

attendre(**non_vide**)

finsi

{ Enlever m du tampon }

m ← **tamp[tête]**;

tête ← (**tête**+1) mod **Taille**;

nb ← **nb** - 1;

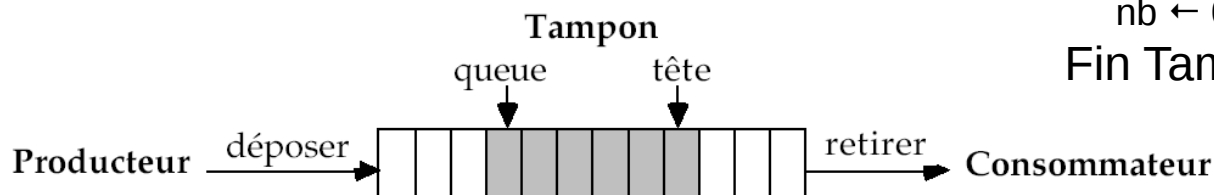
signaler(**non_plein**);

fin **retirer**;

Début

nb ← 0; **tête** ← 0; **queue** ← 0;

Fin Tampon;



- Démo en java...

