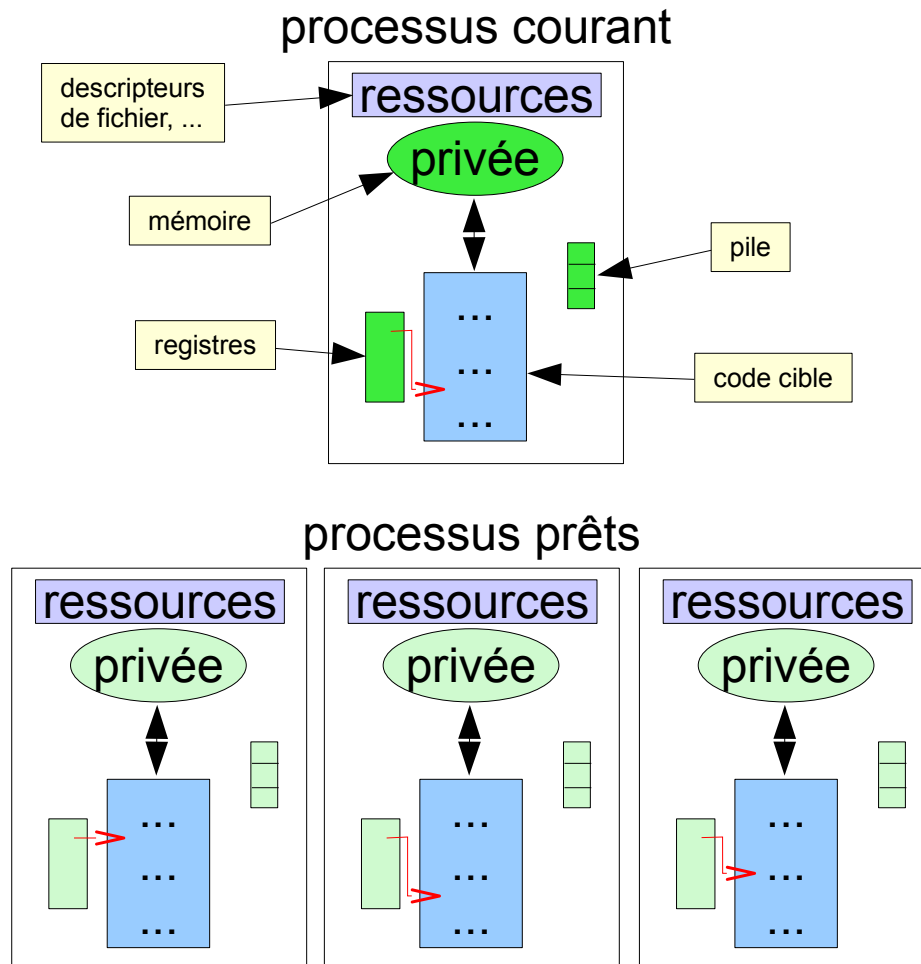


Threads et MMJ

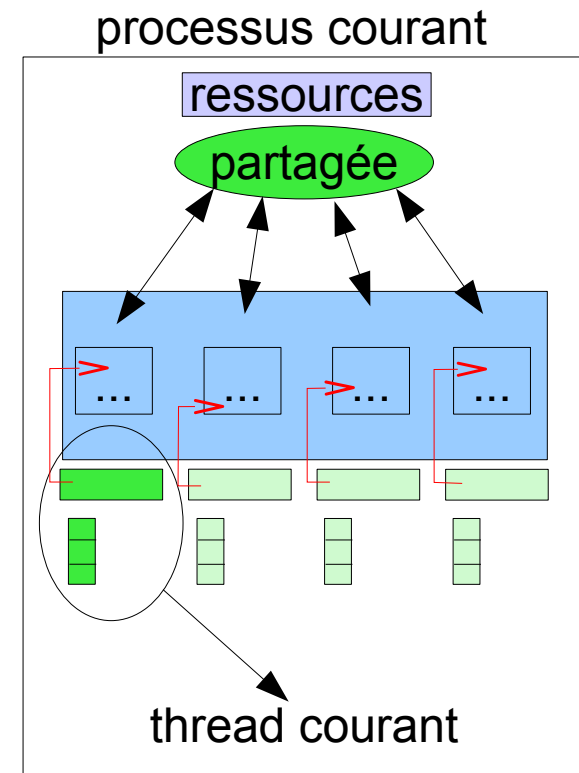
- Généralités sur les threads
- Cycle de vie d'un thread
- Swing et les threads
- Timer Swing
- SwingWorker
- Concurrency
- MMJ
- Thread safety
- Un exemple de manipulation des threads

Thread

système multi-processus

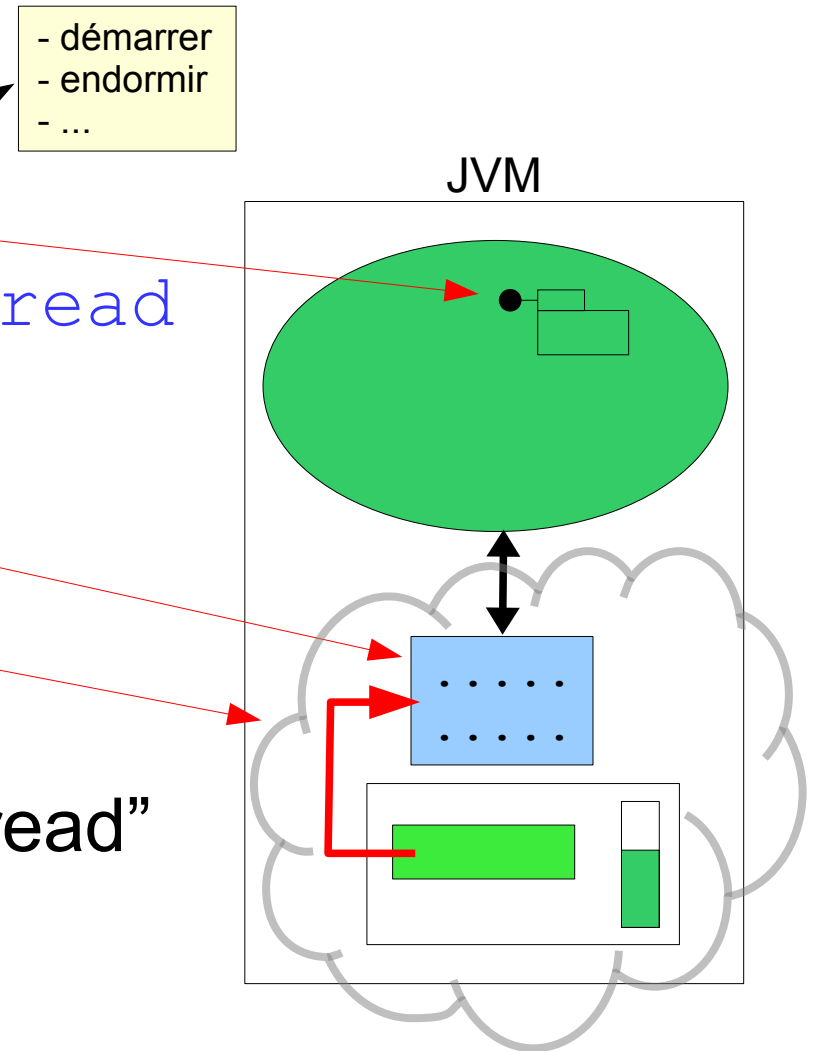


système multi-threads

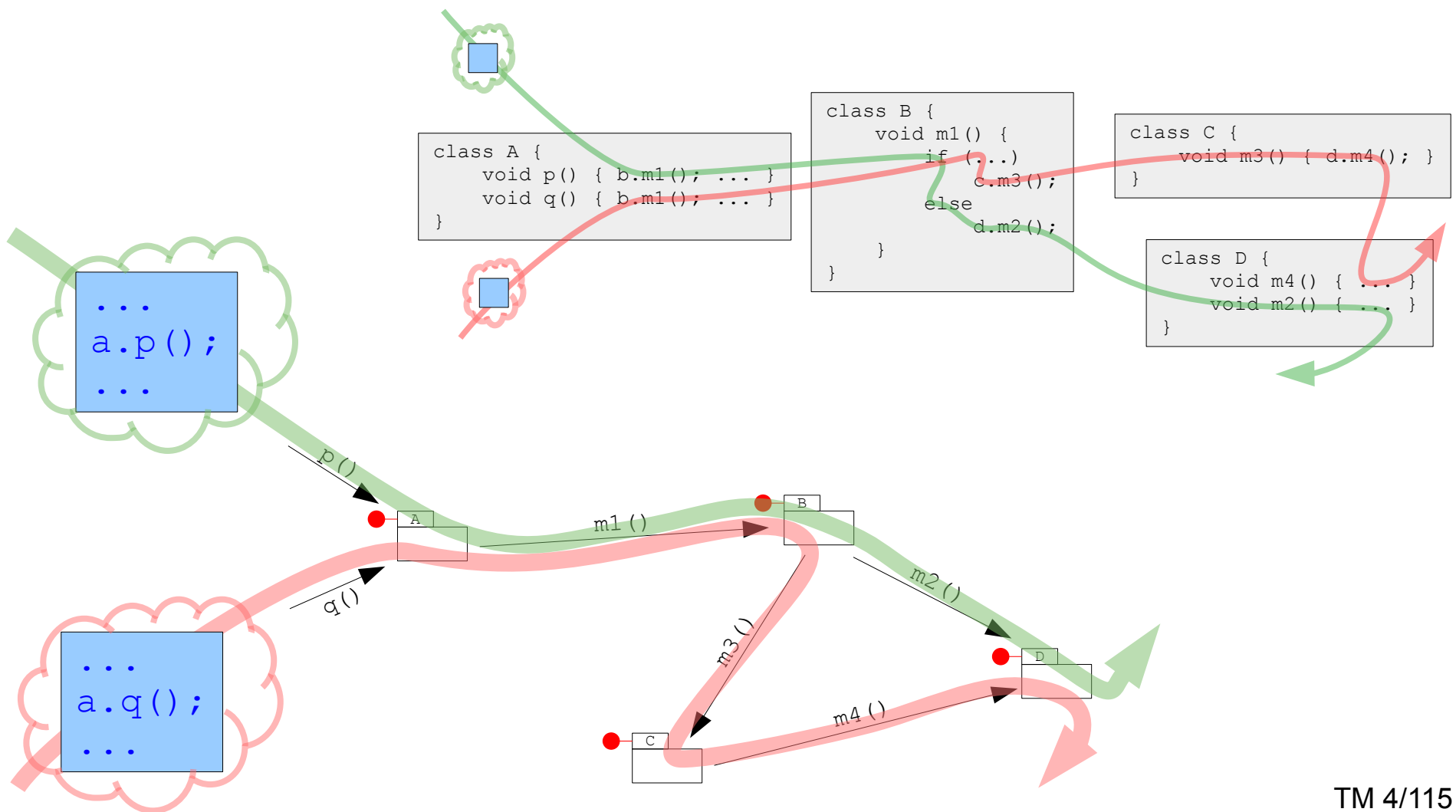


Contrôleur de thread

- Il faut distinguer :
 - le contrôleur du thread
instance de `java.lang.Thread`
 - le code cible du thread
 - le thread
- /!\ abus de langage /!\
 - “thread” ↔ “contrôleur de thread”



« Traces » de threads



Threads présents à l'exécution

- Dans le processus d'exécution d'une JVM :
 - « *main thread* » : exécution de la méthode `main`
 - « *event dispatch thread* » : traitement des événements + affichage des composants graphiques
 - « *toolkit thread* » : création des événements bas-niveau
 - « *user threads* » : définis par le programmeur
 - ...

Créer un contrôleur de thread

```
public class Thread ... {
    // code cible du thread
    private Runnable target;
    public Thread() {
        ...
        target = null;
    }
    public Thread(Runnable r) {
        ...
        target = r;
    }
    public void run() {
        // comportement du thread
        if (target != null) {
            target.run();
        }
    }
    ...
}
```

Forme 1

```
// Concept de « bloc de code exécutable »
public interface Runnable {
    // code à exécuter
    void run();
}
```

Forme 2

```
class MonCode implements Runnable {
    public void run() {
        <bloc de code à exécuter>
    }
}

Thread t = new Thread(new MonCode());
```

forme adaptée à la définition
du code cible d'un thread

```
class MonThread extends Thread {
    public void run() {
        <nouveau comportement du thread>
    }
}

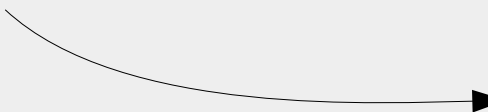
Thread t = new MonThread();
```

forme adaptée à la redéfinition
du comportement d'un thread

Notion de thread courant

- **static** Thread `currentThread()`
 - dans la classe `Thread`
 - contrôleur du thread en cours d'exécution

```
class X {  
    ...  
    public void m() {  
        ...  
        System.out.print("Contrôleur du thread qui exécute le corps de m :");  
        System.out.println(Thread.currentThread());  
        ...  
    }  
    ...  
}
```



par exemple : `Thread[main,5,main]`
ou peut-être : `Thread[AWT-EventQueue-0,6,main]`

Démarrer un thread

- **void** `start()`
 - déclenche la méthode `run` du contrôleur
 - exécute le code cible du thread
 - Précondition :
 - `getState() == Thread.State.NEW`
(sinon `IllegalMonitorStateException`)
 - Postcondition :
 - `isAlive() == true`
 - `getState() == Thread.State.RUNNABLE`

état du contrôleur
rien à voir avec le SE !

Endormir le thread courant

- `static void sleep(long)` endort `currentThread()`

qui est endormi ?
→ le thread qui exécute cette ligne de code

```
...  
try {  
    Thread.sleep(100);  
} catch (InterruptedException e) {  
    // gérer la situation  
}  
...
```

- Remarque :
 - un thread **peut** s'endormir
 - un thread **ne peut pas** en endormir un autre

Terminer un thread

- Terminaison d'un thread = fin de l'exécution de son code cible
- Thread terminé (thread « mort »)
 - `getState() == Thread.State.TERMINATED`
 - `isAlive() == false`
- Impossible de terminer un thread, à la fois :
 - de manière sûre **ET** immédiate

Méthodes `stop` de `Thread`

- Terminent immédiatement un thread
- Méthodes déconseillées (*deprecated*) car
 - arrêt préemptif du code cible
 - pas le temps de « faire le ménage » → état instable
- Donc ne **jamais utiliser** ces méthodes

Méthodes ~~déconseillées~~ *interdites*

- Pour les mêmes raisons que pour `stop` :
 - `void suspend()` est déconseillée
 - `void resume()` est déconseillée
- Forte probabilité d'interblocage
 - (car conservent les verrous)
- Donc ne **jamais utiliser** ces méthodes

Interrompre un thread

- **void** `interrupt()`

- drapeau d'interruption passe à **true**
- ... et c'est tout ! 

`interrupt` traduit un souhait :
on demande au thread de bien
vouloir préparer sa terminaison

- **boolean** `isInterrupted()`

- valeur du drapeau d'interruption

- **static boolean** `interrupted()`

- retourne la valeur du drapeau d'interruption de `Thread.currentThread()`
- **ET** réinitialise le drapeau à **false**

pas terrible
mais seul
moyen de
réinitialiser
le drapeau
à la demande

Politiques d'interruption

- À définir pour chaque thread
- À mettre en place dans le code cible

```
// politique d'interruption :  
// arrêter au plus tôt l'activité  
public void run() {  
    Thread t = Thread.currentThread();  
    int n = 0;  
    while (!t.isInterrupted() && n < Integer.MAX_VALUE) {  
        System.out.println(n++);  
    }  
}
```

```
// politique d'interruption :  
// ignorer l'interruption  
public void run() {  
    int n = 0;  
    while (n < Integer.MAX_VALUE) {  
        System.out.println(n++);  
    }  
}
```

```
// politique d'interruption :  
// réinitialiser l'activité  
public void run() {  
    Thread t = Thread.currentThread();  
    int n = 0;  
    while (n < Integer.MAX_VALUE) {  
        System.out.println(n++);  
        if (t.isInterrupted()) {  
            n = 0;  
        }  
    }  
}
```

Méthodes bloquantes

- *Méthode bloquante* : méthode capable
 - de suspendre l'exécution du thread courant
 - en attendant qu'une condition soit réalisée
 - puis de poursuivre l'exécution du thread courant
- Exemples :
 - Les opérations de lecture bloquante sur un flux
 - Dans la classe `Thread` : `sleep` et `join`
 - Dans la classe `Object` : `wait`

Interruption de méthode bloquante

- Thread bloqué par `wait`, `sleep` ou `join` :
 - la détection de l'interruption se fait "le plus rapidement possible" (*sic*)
 - /!\ drapeau d'interruption réinitialisé à **false**
 - levée d'une `InterruptedException`
- Thread bloqué sur une opération d'entrée/sortie bloquante (flux classique `java.io`)
 - /!\ drapeau d'interruption réinitialisé à **false**
 - levée d'une `IOException`

Exemple d'interruption avec méthode bloquante

```
class Action1 implements Runnable {  
    // politique d'interruption :  
    // arrêter au plus tôt l'activité  
    public void run() {  
        boolean interrupted = false;  
        while (!interrupted) {  
            if (<condition>) {  
                unObjet.appelNonBloquant();  
            } else {  
                try {  
                    Thread.sleep(delay);  
                } catch (IE e) {  
                    interrupted = true;  
                }  
            }  
        }  
    }  
}
```

```
class Action2 implements Runnable {  
    private Thread t;  
    Action2(Thread t) {  
        this.t = t;  
    }  
    public void run() {  
        if (t != null) {  
            t.interrupt();  
        }  
    }  
}
```

```
Thread t = new Thread(new Action1());  
Thread s = new Thread(new Action2(t));  
t.start();  
s.start();
```

- Que se passe-t-il si ... ?
 - `t` interrompu par `s` pendant `sleep`
 - `t` interrompu par `s` pendant `appelNonBloquant`
- Corrigez `Action1.run()` conformément à la politique d'interruption

Rendre volontairement la main

- **static void yield()**
 - `currentThread()` retourne dans le pool des threads exécutables
 - augmente la probabilité d'élection des autres threads

Attendre la fin d'un autre thread

- **void join([long millis])**
 - le thread courant attend la mort du thread cible
 - méthode bloquante et interruptible (IE)

Affichage :
mort du dormeur
mort du veilleur

```
class Action1 implements Runnable {  
    public void run() {  
        try {  
            Thread.sleep(longDelay);  
        } catch (IE ignore) {  
            // rien  
        }  
        sysout.println("mort du dormeur");  
    }  
}
```

```
Thread t = new Thread(new Action1());  
Thread s = new Thread(new Action2(t));  
t.start();  
s.start();
```

```
class Action2 implements Runnable {  
    private Thread t;  
    Action2(Thread t) { this.t = t; }  
    public void run() {  
        while (t.isAlive()) {  
            try {  
                t.join();  
            } catch (IE ignore) {  
                // rien  
            }  
        }  
        sysout.println("mort du veilleur");  
    }  
}
```

Priorité

- *Priorité* : mesure de la précédence d'un thread
 - `int getPriority()`
 - `NORM_PRIORITY` (5) par défaut
 - `void setPriority(int newPriority)`
 - entre `MIN_PRIORITY` (1) et `MAX_PRIORITY` (10)
- Influence la fréquence d'attribution du CPU à chaque thread...
 - ... dépend du SE

Démons

- *Démon* : processus qui tourne en tâche de fond
- `void setDaemon(boolean)`
 - bascule l'état du thread entre démon et non démon
 - doit être appelée avant `start`
- Arrêt automatique de la JVM après la mort de tous les threads non démon
 - les démons sont interrompus brutalement
 - ne pas laisser les démons accéder aux fichiers, ...

Cycle de vie d'un thread (de contrôleur t)

