Le multi-threads

Les threads

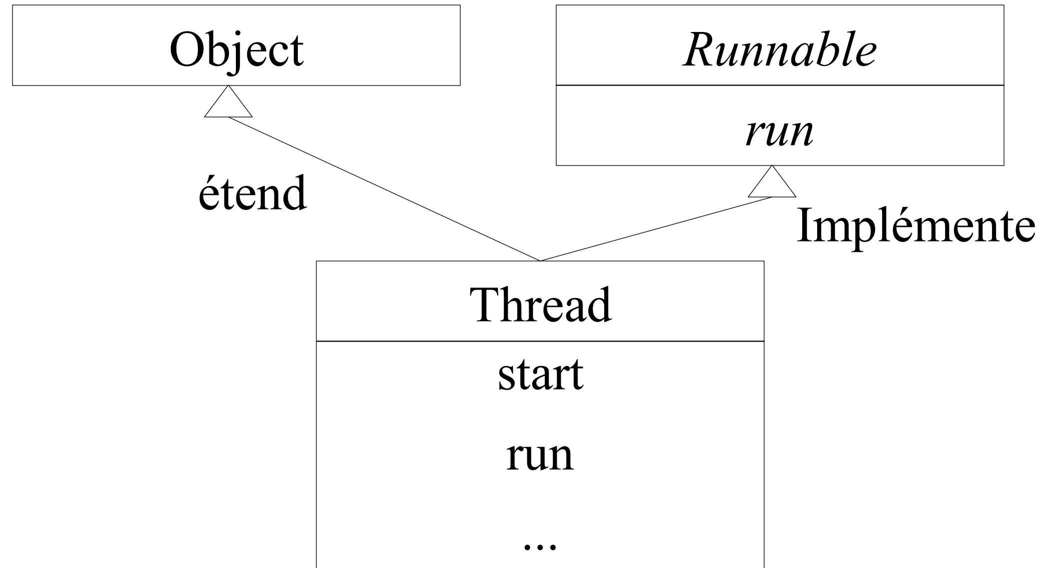
Définition :

- un thread est un flux d'instructions séquentielles s'exécutant dans un processus ;
- une application multi-thread supporte plusieurs flux d'instructions concurrents ;
- les threads d'un processus partagent le même espace d'adressage.

- Les threads sont abondamment utilisés par la machine virtuelle Java :
  - le ramasse-miettes est un thread de très base priorité qui récupère la mémoire qui n'est plus utilisée ;
  - la méthode paint d'une applet n'est pas appelée par l'applet mais par un thread créé par le navigateur.

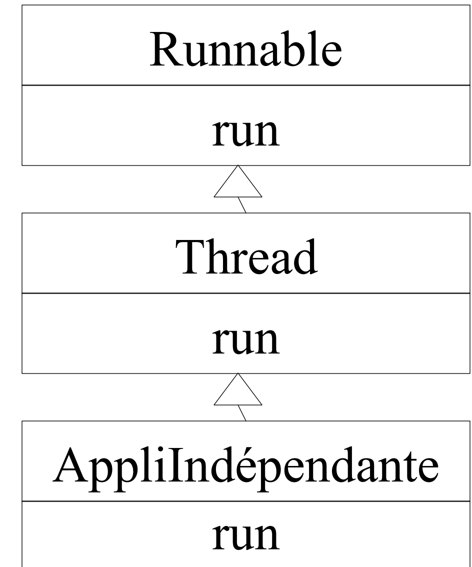
Créer des threads

- pour créer un thread, il faut construire un objet de type Thread ;
- pour lancer un thread, il faut appeler la méthode start ;
- le code à exécuter séparément de celui des autres threads doit être défini dans la méthode run.



Les classes java pour le multi-threads

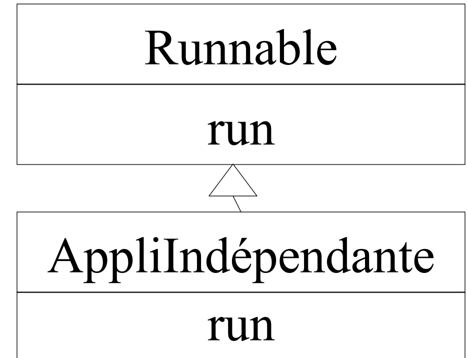
Réaliser une application multi-threads : 1ére méthode




```
class AppliIndépendante extends Thread {
      public void run() {       // ce qui doit s'exécuter
            // …                // de façon concurrente
      }
      public static void main( String [] argv ){
            AppliIndépendante monAppli = new AppliIndépendante () ;
            monAppli.start() ;                  // exécution séparée de run
      }
}
```

Inconvénient : la classe AppliIndépendante ne peut pas étendre une autre classe que Thread.

Réaliser une application multi-threads : 2ième méthode




```
class AppliIndépendante extends Bidon implements Runnable {
    private Thread monThread ;
    public AppliIndépendante() {
        monThread = new Thread( this ) ;
        monThread.start() ;          // exécution séparée de run
    }
    public void run() {       // ce qui doit s'exécuter
        // …                   // de façon concurrente
    }
    public static void main( String [] argv ){
        AppliIndépendante monAppli = new AppliIndépendante () ;
    }
}
```

Avantage : la classe AppliIndépendante peut étendre une classe quelconque.

Contrôler les threads

Thread

Lance l'exécution d'un thread (ne peut être appelée qu'une seule fois)

start

sleep

Mise en sommeil d'un thread (méthode statique)

Endormir un thread

Un thread ne consomme pas de temps CPU pendant qu'il est endormi.

```
class MonAppli implements Runnable {
    private Thread monThread ;
    public MonAppli () {
        monThread = new Thread( this ) ;
        monThread.start() ;
    }
    public void run() {
        while( true ){
            try {
                Thread.sleep( 1000 ) ;      // endort le Thread courant
            } catch( InterruptedException e ) {}
            // …
        }
    }
}
```

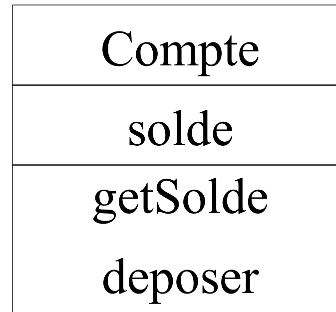
Arrêter un thread

Ne pas utiliser la méthode stop (de la classe Thread) qui est boguée !

```
public class MonAppli implements Runnable {
    private volatile Thread monThread ;
    public MonAppli () {
        monThread = new Thread( this ) ;
        monThread.start() ;
    }
    public void run() {
        Thread thisThread = Thread.currentThread() ;
        while( monThread == thisThread ){
            // …
        }
    }
    public void stop() {
        monThread = null ;
    }
}
```

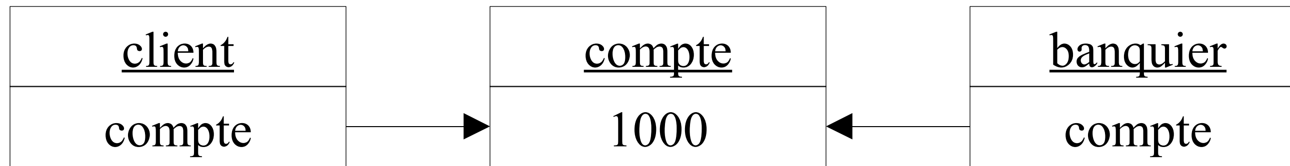
Synchroniser les threads

Comment conserver l'intégrité d'un objet quand plusieurs threads souhaitent changer son état ?



classe compte bancaire

Deux objets, client et banquier, font référence à un unique objet compte :

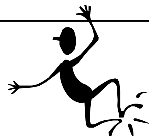


compte.deposer(100) ;

int solde = compte.getSolde() ;

```
public int getSolde() {
    return solde ;
}
```

Si le dépôt intervient avant la fin de la consultation ?!?



Mot clef synchronized

Le mot clef synchronized permet la pose d'un verrou transparent au programmeur qui garanti qu'un seul thread à la fois peut accéder à la partie synchronisée.

```
class CompteBancaire {
    private int solde ;

    public CompteBancaire( int soldeInitial ) {
        solde = soldeInitial ;
    }
    public synchronized int getSolde() {
        return solde ;
    }
    public synchronized void deposer( int somme ) {
        solde += somme ;
    }
}
```

Mot clef synchronized

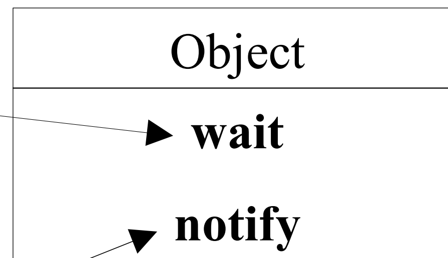
- ce sont les objets qui sont verrouillés (il y a 1 verrou par objet même si plusieurs méthodes de la classe sont synchronisées)
- les méthodes de classe (statiques) peuvent être synchronisée (c'est alors un verrou de classe qui est posé ; ce verrou ne verrouille pas d 'éventuels objets de cette classe) ;
- la redéfinition dans une sous-classe d'une méthode verrouillée dans une super-classe peut être synchronisée ou pas ;
- le mot clef synchronized peut s'appliquer sur un objet pour une suite d'instructions à exécuter :

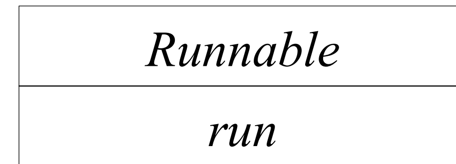
```
synchronized( objet ) {
// manipule objet
}
```

La communication entre threads : wait et notify

Place le thread courant en attente d'un événement

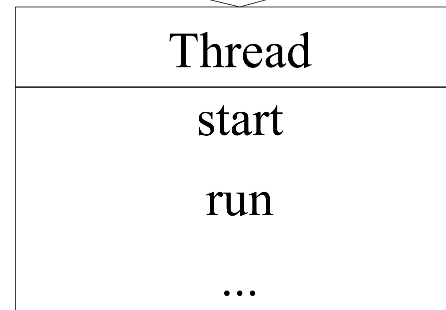
Signale à un thread en attente qu'un événement c'est produit





étend

Implémente

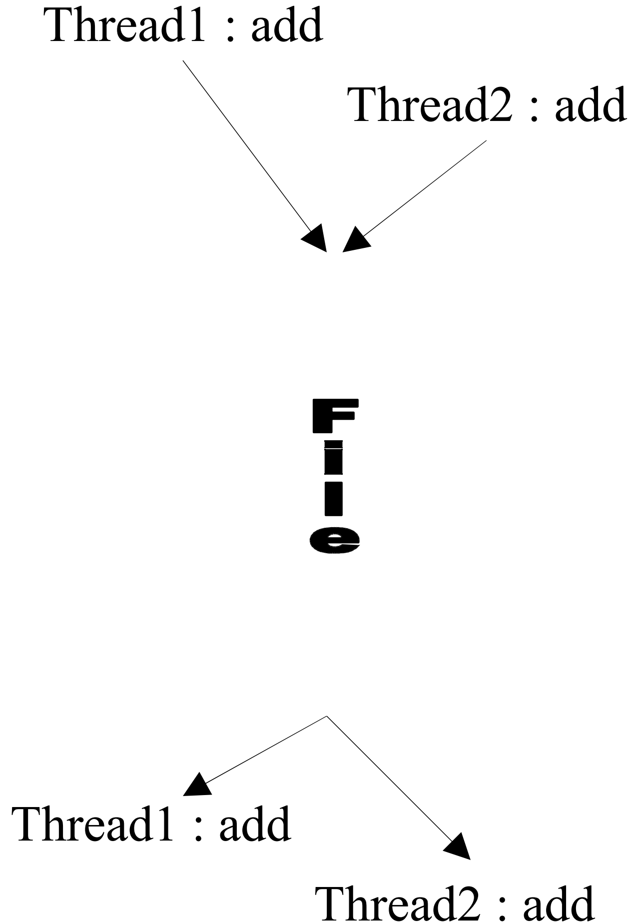


La communication entre threads : wait et notify

```
public class MonAppli implements Runnable {
    private volatile Thread monThread ;
    private boolean go ;
    public MonAppli () {
        monThread = new Thread( this ) ;
        monThread.start() ;
        go = false ;
    }
    public void run() {
        try {
            while( go == false ) wait() ;
        } catch( InterruptedException e ) { return ; }
        // …
    }
    public synchronized void démarre() {
        go = true ;
        notify() ;
    }
}
```

La communication entre threads : wait et notify (1/2)

- Gestion d'une file de messages accédée concurremment par plusieurs threads :




```
public class MonAppli implements Runnable {
    private Thread monThread ;
    private File file ;
    public MonAppli ( File file ) {
        this.file = file ;
        monThread = new Thread( this ) ;
        monThread.start() ;
    }
    public void run() {
        String message = Reseau.getMessage() ;
        file.add( message ) ;
        // …
        System.out.println( (String)file.get() ) ;
    }
}
```

La communication entre threads : wait et notify (2/2)

- Gestion d'une file avec mise en attente des threads voulant obtenir un élément quand la file est vide.

```
public class File {
        private Vector file = new Vector() ;
        boolean vide = true ;                          // événement à signaler

        public synchronized void add( Object o ) {
                file.addElement( o ) ;          // ajoute un élément
                vide = false ;                  // événement à signaler
                notify() ;                      // notifie qu'un élément est présent
        }
        public synchronized Object get() {
                try {
                        while( vide ) wait() ;  // attend qu'un élément soit présent
                } catch( InterruptedException e ) { return null ; }
                if( file.size() == 1 ) vide = true ;       // ré-initialise l'événement
                return file.firstElement() ;               // retourne l'élément
        }
}
```

La communication entre threads : wait et notify

- ces méthodes s'appliquent à un objet partagé par plusieurs threads ;
- elles ne peuvent être invoquées que dans une partie synchronisée de code, sinon l'exception IllegalMonitorStateException est levée ;
- notify() signale qu'un événement c'est produit à **un seul thread** (celui qui attend depuis le plus longtemps) ;
- notifyAll() signale à tous les threads en attente qu'un événement c'est produit ;
- il est possible d 'attendre pendant un certain temps :
  - wait(long timeout) ; // time out en millisecondes
  - wait(0) <=> wait() ;
  - wait(long timeout, int nanos) ; // time out en milli et nanosecondes
- écrire « if(condition) wait() ; » n'est pas correct, car cela suppose qu'être réveillé signifie que la condition a été satisfaite.

L'ordonnancement des threads

Thread
MIN_PRIORITY NORM_PRIORITY MAX_PRIORITY
setPriority getPriority currentThread yield

Champs statiques : MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY

```
Thread thread = thread.currentThread() ;
thread.setPriority( Thread. MIN_PRIORITY ) ;
```

- Quand un thread se bloque sur une fonction système, ou s'endort (sleep), Java fait exécuter un des autres threads de plus haute priorité => donner aux threads d'interface avec l'utilisateur la plus haute priorité pour qu'ils répondent rapidement aux sollicitations et qu'ils rendent la main dès qu'ils se bloquent ;
- yield() indique à l'ordonnanceur que l'exécution du thread courant peut être remplacée par celle d'un autre thread ;

Terminer une application multi-threads

- Un thread s’arrête quand sa méthode run se termine, ou qu’il y a appel à la méthode stop (méthode dépréciée) ;
- chaque application est composé d’au moins un thread (celui qui exécute la méthode main) ;
- si l’application créée des threads (threads utilisateur), ceux-ci continus leurs exécutions même après que la méthode main est terminée, et ce, tant que leur méthodes run se poursuit ;
- pour qu’un thread se termine dès que les threads utilisateurs sont finis, il faut en faire des threads démon : utilisez la méthode setDaemon(on) de la classe Thread.

Les groupes de threads

- Les threads peuvent être regroupés en ThreadGroup ;
- un thread peut accéder à des propriétés d'autres threads (changer leur priorité par exemple) uniquement s'ils font parties du même groupe ;
- par défaut, un thread est placé dans le groupe qui l'a créé ;
- le choix du groupe auquel appartient un thread se fait lors de la construction du thread : public Thread(ThreadGroup group, String name) ;