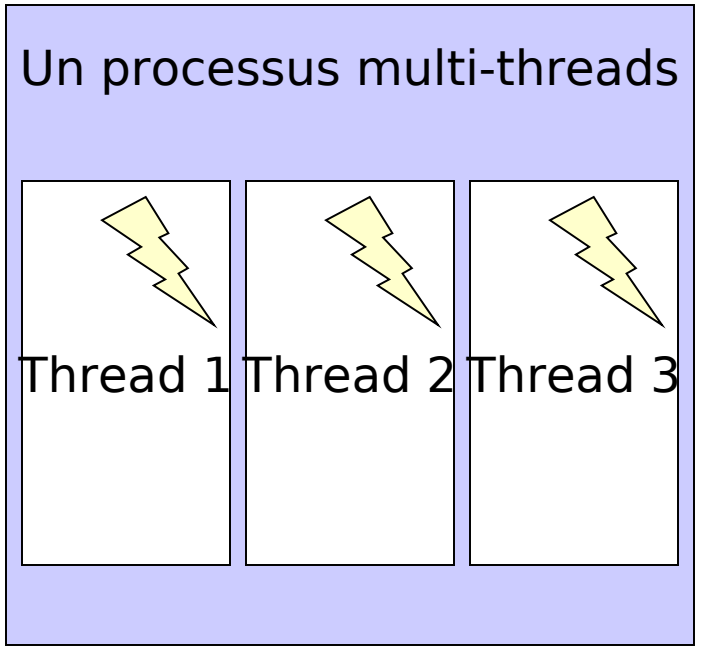
Les threads

- Dans un environnement mutlti-tâches, plusieurs processus peuvent s'exécuter en parallèle.
- Dans une application multi-threads, plusieurs activités peuvent s'exécuter en parallèles.
  - Par exemple plusieurs fonctions d'une application peuvent s'exécuter en même temps.
- Un « thread » est également appelé activité ou processus léger.

Notion de threads



Il faut voir une processus comme le code correspondant au programme.
Le thread est l'entité qui exécute le code.

Toute application comporte au moins un thread appelé « thread pincipal ».

Définir un thread

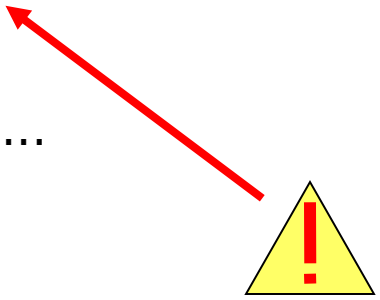
- Définir un thread revient à créer une activité d'exécution pour un processus.
- La définition d'un thread revient à créer une classe qui hérite de « ***java.lang.Thread*** ».
- Le fait de créer une instance de la classe qui implante « java.lang.Thread » n'entraîne pas la création d'un thread.
- Pour créer un thread, on doit appeler la méthode « ***start*** »
- L'appel à « start » entraîne la création du thread et le début de son traitement qui commence par la méthode « ***run*** ».

Un exemple de thread

```
public class monThread extends java.lang.Thread
{
    // ...

    public void run()
    {
        // Traitement du thread.
    }

    // ...
}
```



La sortie de la méthode « run » met fin à la vie du Thread.

Exemple d'utilisation d'un thread

```
public class ExempleThread extends java.lang.Thread
{
    public static int threadCompteur = 0;
    public int numThread = 0;
    public int count = 5;
    public ExempleThread()
    {
        numThread = ThreadCompteur++;
        System.out.println("Création du thread n°" + numThread );
    }
    public void run()
    {
        while ( count != 0 )
        {
          System.out.println("Thread n°" + numThread + " , compteur = " +
    count-- );
        }
    }
    public static void main( String [] args )
    {
        for ( int i=0; i<3; i++ )
            new ExempleThread().start();
        System.out.println("Tous les threads sont lancés");
    }
}
```

A l'exécution...

Création du thread n°1
Création du thread n°2
Création du thread n°3
Thread n°1, compteur = 5
Thread n°2, compteur = 5
Thread n°2, compteur = 4
Thread n°2, compteur = 3
Thread n°3, compteur = 5
Thread n°1, compteur = 4
Tous les threads sont lancés
Thread n°3, compteur = 4
...

L'ordonnancement est imprévisible.

Interruption et reprise d'un Thread

- On peut interrompre un thread par l'intermédiaire de l'opération « ***suspend*** ».
- Pour relancer l'exécution d'un thread, on fait appel à la méthode « ***resume*** ».
- On peut également marquer une pause dans l'exécution d'un thread en employant l'opération « ***sleep*** ».
- Enfin, un thread peut attendre la fin d'un autre thread en appliquant l'opération « ***join*** » sur le thread en question.

Autre opérations d'un thread

- Pour arrêter un thread on utilise l'opération « ***stop*** » :

  ```
  public final void stop();
  ```

- Pour connaître la priorité d'un thread, on emploi la méthode « ***getPriority*** » :

  ```
  public final int getPriority();
  ```

- De plus, pour fixer la priorité d'un thread, on utilise « ***setPriority*** » :

  ```
  public final void setPriority(int newPriority);
  ```

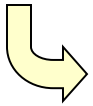
Comment récupérer le thread courant ?

- Lorsqu'une méthode est exécutée, elle peut l'être par plusieurs threads.

- Pour connaître le thread courant, elle peut utiliser l'opération « ***currentThread*** » :

```
public static Thread currentThread();
```

- A partir de la référence vers le thread récupéré, on peut appliquer toutes les opérations traditionnelles aux threads.



L'opération « currentThread » peut être également utilisée pour récupérer le thread principal.

Accès concurrents

- Que se passe t'il si plusieurs threads accèdent à la même méthode ou à la même ressource au même instant ?
  - Comportement imprévisible selon les applications
  - problématique des accès concurrents

Accès concurrents

```
class ListeTab {

private String[] tab = new String[50];
    private int index = 0;
    void ajoute(String s) {
        tab[index] = s;
        index++;
    }
}
```

Accès concurrents

Thread a

```
void ajoute(String s) {
    tab[index] = s; //(a1)
    index++;        //(a2)
}
```

Thread b

```
void ajoute(String s) {
    tab[index] = s; //(b1)
    index++;        //(b2)
}
```

(a1) (a2) (b1) (b2), est une exécution possible, cohérente ;

(b1) (b2) (a1) (a2), est une exécution possible, cohérente ;

(a1) (b1) (b2) (a2), est une exécution possible, mais incohérente :

le tableau ne contient pas la chaîne de caractères ajoutée par T1,

et une case de la liste est vide.

Accès concurrents

Thread a

```
void ajoute(String s) {
    tab[index] = s; //(a1)
    index++;        //(a2)
}
```

Thread b

```
void ajoute(String s) {
    tab[index] = s; //(b1)
    index++;        //(b2)
}
```

(a1) (a2) (b1) (b2), est une exécution possible, cohérente ;

(b1) (b2) (a1) (a2), est une exécution possible, cohérente ;

(a1) (b1) (b2) (a2), est une exécution possible, mais incohérente :

le tableau ne contient pas la chaîne de caractères ajoutée par T1,

et une case de la liste est vide.

Accès concurrents

- Pour qu'une méthode ne soit pas utilisée par plus d'un thread à la fois, il faut la spécifier « synchronized » :

synchronized type_de_retour nom_methode (liste des paramètres)

Un même thread pourra tout de même appeler récursivement cette opération.

Les verrous

- Un verrou (en anglais « ***mutex*** ») est un concept qui lorsqu'il est activé empêche les threads qui n'ont pas activés le verrou d'utiliser le code verrouillé.

- Tant que le verrou n'est pas levé, seul un thread peut être actif dans le code verrouillé.

- Chaque objet java peut servir de verrou.

  (Les méthodes de classes peuvent aussi être synchronisées (le verrou est alors sur la classe))
- Comment créer une zone verrouillée ?
  - On applique « ***synchronized*** » sur un objet.

Exemple de verrou

```
java.lang.Object verrou = new java.lang.Object();
synchronized ( verrou )
{
    // Zone verrouillée.
}
```

On pourrait très bien écrire :

```
synchronized ( this )
{
    // Zone verrouillée.
}
```

Ecritures équivalentes

```
void methode() {
    synchronized(this) {
        //section critique
    }
}
```

```
synchronized void methode()
 {
    //section critique
}
```

L'interface « java.lang.Runnable »

- Pour définir un thread, on peut également implanter l'interface « ***java.lang.Runnable*** » plutôt que d'hériter de « java.lang.Thread »

- Cette interface définie l'opération « ***run*** » qui doit être implantée et qui correspond à la méthode appelée au lancement du thread.

- Pour créer un thread à partir d'une classe qui implante « Runnable », on doit créer une instance de « java.lang.Thread » qui prenne en paramètre une référence vers cette classe.

```
public Thread(Runnable target);
```

On ne peut pas appliquer les opération de « Thread » directement sur une classe qui implémente « Runnable ».

Exemple de thread utilisant « Runnable »

```
public class monThread implements Runnable
{
    public void run()
    { // ... }

    public static void main( String [] args )
    {
        Thread t = new Thread( new monThread() );
        t.start();
    }
}
```

Le JDK 1.2 et les opérations sur les threads

- Plusieurs opérations sont notés « deprecated » dans le JDK 1.2.
- En particulier les opérations « suspend » et « resume » ne doivent plus être utilisées.
  - Utiliser à la place une synchronisation à partir d'un verrou.
- De plus, l'opération « stop » est également déconseillée au profit de l'utilisation d'une variable :

```
public void run()
{
   while ( stop != true )
   { // ... }
}
public void stop()
{ stop = true; }
```

Les threads démons

- Lorsqu'une application créée un thread, celle-ci reste bloquée tant que le thread ne meurt pas.
- Pour éviter cela, il est possible de signaler qu'un thread joue le rôle de démon.
- Lorsqu'une application termine, tous les threads démons sont alors stoppés.
- Pour signaler le fait qu'un thread est un démon, on doit lui appliquer l'opération « ***setDaemon*** » :

```
public final void setDaemon(boolean on);
```

Exercice applicatif

Proposez une implémentation de l'application suivante :

Soit une pâtisserie proposant des gâteaux. Chaque client qui se présente achète 1 à 5 gâteaux. La serveuse met 20 secondes (10s sont simulées 1 seconde) pour servir chaque gâteau.

- Identifier les classes à développer
- Quelles classes doivent avoir leur propre Thread ?
- Y a-til une section critique à protéger ? Pourquoi ?
- Proposer une implémentation

Synchronisation coopérative

- Méthodes de la classe Objet :

  wait() : endort le thread qui l'exécute sur l'objet

  notify() : réveille aléatoirement un thread endormi sur l'objet

  notifyAll() : réveille tous les thread endormis sur l'objet

Synchronisation coopérative

- Méthodes de la classe Objet :

  wait() : endort le thread qui l'exécute sur l'objet

  notify() : réveille aléatoirement un thread endormi sur l'objet

  notifyAll() : réveille tous les thread endormis sur l'objet

  Toutes ces opérations doivent être synchronisées sur l'objet

  Pourquoi ?

Synchronisation coopérative

- Ne pas confondre les Threads et les Objets

  C'est le thread qui execute la fonction wait() qui s'endort

Synchronisation coopérative

```
Class C extends Thread {
        public void run() {
                test();
        }
        public synchronized void test()
        {
                wait(); // + gestion
                   //exception
        }
}
```

Main()

C c = new C();

c.start(); => ??

c.notify(); => ??

c.test(); => ??

Synchronisation coopérative

```
Class C extends Thread {
        public void run() {
                test();
        }
        public synchronized void test()
        {
                wait(); // + gestion
                   //exception
        }
}
```

Main()

C c = new C();

c.start(); => *le thread c se lance et s'endort*

c.notify(); => *le thread c se réveille et se termine*

c.test(); => *le thread du main s'endort*

Exemple

```
Class ListeTab {

    private String[] tab = new String[50];
    private int index = 0;

    synchronized void ajoute(String s) {
        tab[index] = s;
        index++;
        notify();
        System.out.println("notify() exécuté");
    }

    synchronized String getPremierElementBloquant() {
        //tant que la liste est vide
        while(index == 0) {
            try {
                //attente passive
                wait();
            } catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        return tab[0];
    }

}
```

Exercice

- Proposer une solution permettant de conserver l'ordre des clients en attente dans l'exercice précédent.