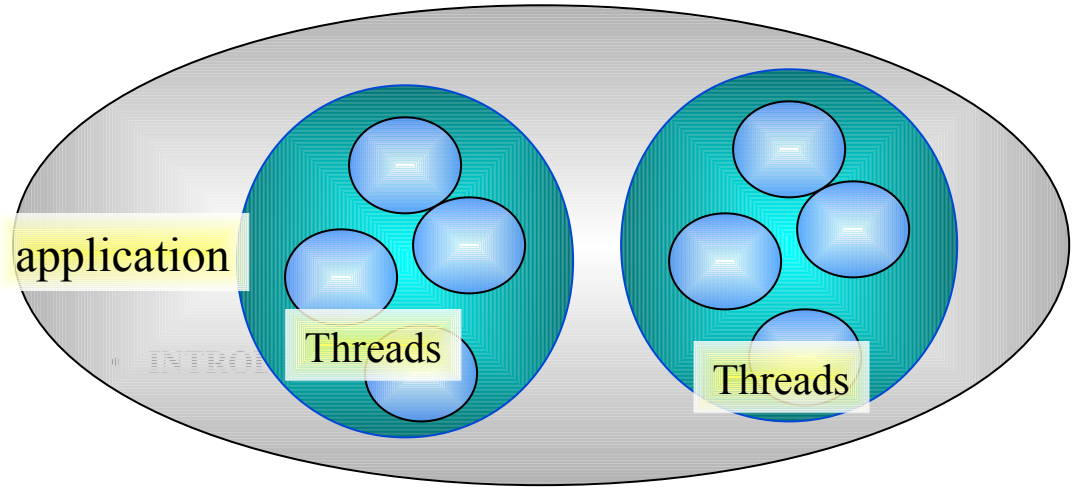
# PROGRAMMATION CONCURRENTE

# LES THREADS EN JAVA




- **PROCESSUS LÉGERS (THREADS)**
- **EXCLUSION MUTUELLE**
- **SYNCHRONISATION**

# Bases des fils d'exécution Java

***d'après l'excellent livre de Yakov Fain:***

***"Programmation Java pour les enfants, les parents et les grands-parents"***

- **Exécution d'un programme en séquence**: une action après l'autre.
  Si un programme appelle deux méthodes, la seconde méthode attend que la première se soit terminée.
  Autrement dit, chacun de nos programmes n'a qu'un ***fil d'exécution* (*thread of execution*).**

- Cependant, dans la vraie vie, nous pouvons faire plusieurs choses en même temps, comme manger, parler au téléphone, regarder la télévision et faire nos devoirs. Pour mener à bien toutes ces actions ***en parallèle***, nous utilisons plusieurs ***processeurs*** : les mains, les yeux et la bouche.
- Certains des ordinateurs les plus chers ont aussi deux processeurs ou plus.
- Un ordinateur mono-processeur effectue les calculs, envoie les commandes à l'écran, au disque, aux ordinateurs distants, etc.
- Même un unique processeur peut exécuter plusieurs actions à la fois si le programme utilise des ***fils d'exécution multiples* (*multiple threads*).**
- Une classe Java peut lancer plusieurs fils d'exécution qui obtiennent chacun à leur tour des **tranches du temps** de l'ordinateur (***time slicing***)
- Un bon exemple de programme capable de créer de multiples fils d'exécution est un navigateur web. Il est possible de naviguer sur Internet tout en téléchargeant des fichiers : un seul programme exécute deux fils d'exécution.

# I Introduction

• Un **système d'exploitation multitâches** permet de traiter plusieurs tâches ***"en même temps"*** au sein d'une même application.

• Une application peut exécuter de **manière concurrente** plusieurs **processus légers**, appelés **threads**. L'application fonctionne en **multithreading**.

• On différencie généralement les **processus légers** des **processus lourds**.

• Les premiers sont internes à une application alors que les processus lourds sont vus par le système d'exploitation comme plusieurs processus avec chacun leur propre espace d'adressage.

• Les processus légers partagent un **espace d'adressage commun**.

• Une fois crée, un thread peut être démarré, exécuté, stoppé.

• **La programmation concurrente sous-entend mécanismes de synchronisation et d'exclusion mutuelle et de coopération.**

## Définition du multi-threading

- C'est la division d'un programme en **plusieurs unités d'exécution évoluant en parallèle**.

  Chaque thread exécute comme un programme indépendant.

  Mais peut partager un espace mémoire avec d'autres threads.
- C'est un **technique très puissante**, permettant de
  - Augmenter le taux d'utilisation de ressources précieuses
  - Améliorer la réactivité des interfaces utilisateurs
  - Faire de l'animation
  - Communiquer en réseau
  - …
- **Demande un effort de conception supplémentaire pour bien gérer l'accès concurrent aux ressources du programme.**

## La classe Thread et l'interface Runnable

• Java fournit la **classe Thread,** abstraction du thread physique dans la JVM. Cette classe contient toutes les **méthodes nécessaires pour gérer des threads**.

• Java fournit également l'**interface Runnable** dont la seule méthode abstraite est **run()** :

**run()** est la méthode "main" d'un thread, appelée au démarrage du thread.

son exécution peut être interrompue.

***Tout objet qui implémente l'interface Runnable peut être exécuté comme un thread.***

## Méthodes et classes Java concernées

| java.lang.Thread | java.lang.Runnable | java.lang.Object |
|---|---|---|
| Thread() | run() | wait() |
| Thread(Runnable) | | notifyAll() |
| start() | | notify() |
| ~~stop()~~ | | |
| run() | | |
| ~~suspend()~~ | | |
| ~~resume()~~ | | |
| sleep() | | |
| ~~destroy()~~ | | |
| setDaemon() | | |
| setPriority() | | |
| join() | | |

**Les méthodes barrées sont dépréciées par des raisons de sécurité.**

**Remarque:** la méthode ~~stop()~~ est qualifiée de **final**. On ne peut donc pas la redéfinir.

**Détruire (destroy) un thread**

• La classe **Thread** déclare une méthode ~~**destroy()**~~.

• Totalement inutile : N'a jamais été implémenté

• Depuis Java 2, appeler cette méthode ne fais que lancer une **NoSuchMethodException**.

## Priorité et ordonnancement

• **Pré-empti**f, le processus de **plus forte priorité** devrait avoir le processeur

• ***Arnold et Gosling96 :*** *When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to implement mutual exclusion.*

• **Priorité de 1 à 10 (par défaut 5)**.

**Un thread adopte la priorité de son processus créateur**

***setPriority(int p)*** permet de changer la priorité

• **Ordonnancement dépendant des plate-formes***

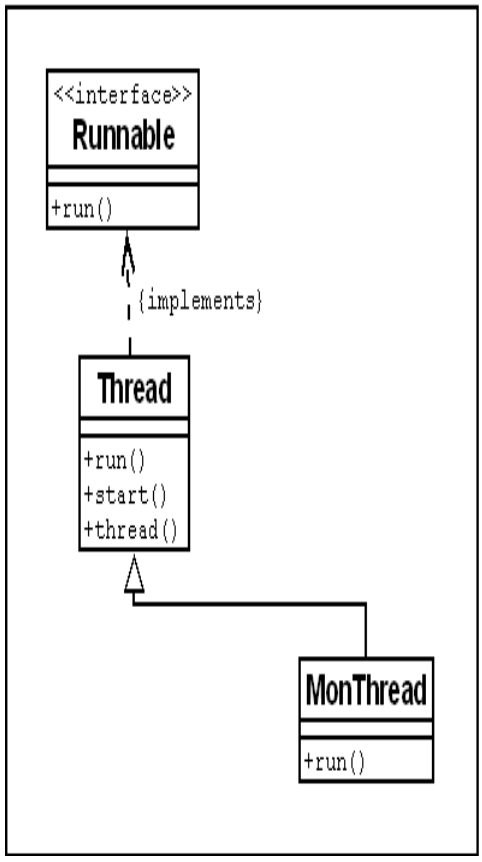
- **Tourniquet** facultatif pour les processus de **même priorité**,
- Le choix du **processus actif** parmi les **éligibles** de même priorité est **arbitraire**
- La sémantique de la **méthode *yield() (passage au suivant)*** n'est pas définie, certaines plate-formes peuvent l'ignorer ( en général les plate-formes implantant un tourniquet)

*** L'implémentation et le comportement de l'ordonnanceur de processus (scheduler) n'est pas spécifié par Sun. Cela veut dire que les différentes machines virtuelles Java (JVM) n'auront pas forcément le même comportement.**

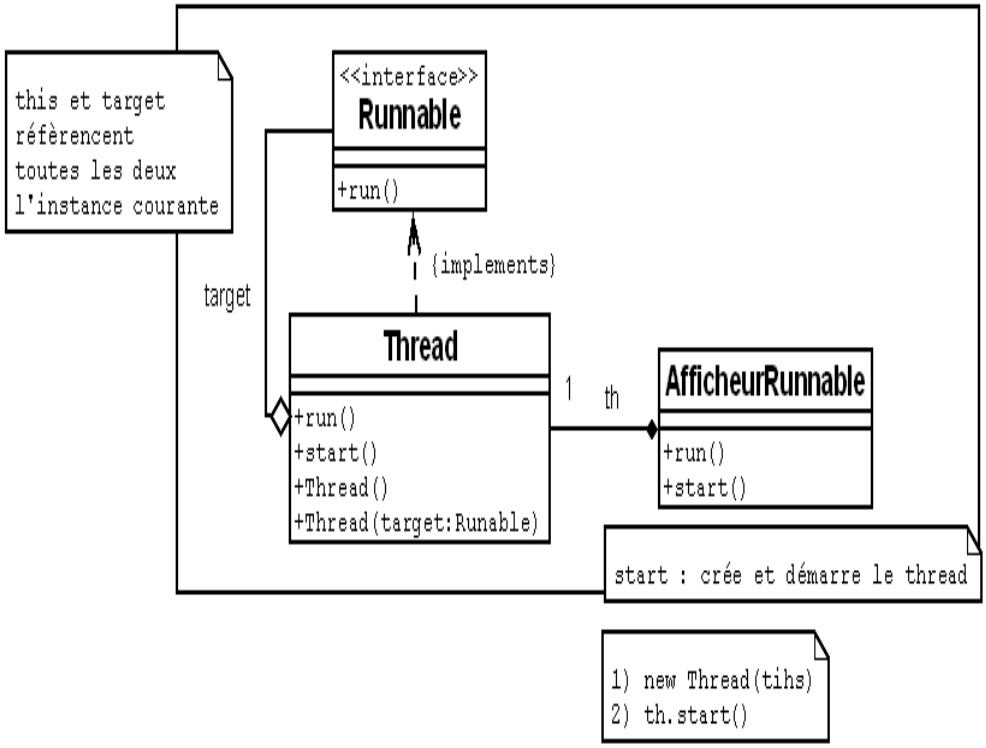
## II Processus légers (threads)

### La Classe Thread et l'interface Runnable


modèle 1




modèle 2



# MODELE 1: Dériver de Thread

C'est l'approche la plus simple pour créer un thread.

• **Méthode :**

définir une **sous-classe de Thread**

implémenter la méthode **run()** par notre propre algorithme

démarrer le thread avec **start()**

la méthode **run()** est alors déclenchée, et elle s'arrête lors d'un appel à **~~stop()~~.**

• **run()** c'est le "main()" d'un thread, appelé par la méthode **start()**.

**En résumé:**

```
// création du thread
Thread t = new Thread () ;

// démarrage du thread
t. start () ;

// lancement du thread: appel automatique
// t. run () ;

// exécution interrompue
t. ~~stop()~~ ;
```

**Exemple 1**

```
public class Afficheur extends Thread
{
	/** constructeur permettant de nommer le processus */
	public Afficheur ( String s )
	{
		super(s);
	}

	/** affiche son nom puis passe le contrôle au suivant */
	public void run ()
	{
		while (true)
		{
			System.out.println("je suis le processus " + getName());
			Thread.yield();    // passe le contrôle au suivant
		}
	}


	public static void main ( String args[] )
	{
		Afficheur thread1 = new Afficheur("1");
		Afficheur thread2 = new Afficheur("2");

		thread1.start();
		thread2.start();

		while (true)
		{
			System.out.println("je suis la tache principale !");
			Thread.yield();
		}
	}
}
```

**Exécution**

```
je suis la tache principale !
je suis le processus 1
je suis le processus 2
je suis la tache principale !
je suis le processus 1
je suis le processus 2
je suis la tache principale !
je suis le processus 1
je suis le processus 2
je suis la tache principale !
je suis le processus 1
je suis le processus 2
je suis la tache principale !
je suis le processus 1
je suis le processus 2
je suis la tache principale !
je suis le processus 1
je suis le processus 2
```

**Exemple 2**

```
public class SimpleThread extends Thread {
  long sleepTime;

  SimpleThread(long s) {
        sleepTime = s;
  }

  public void run() {
        try {
              for (int n = 5; n > 0; n--) {
                    System.out.println(getName() + ": " + n);
                    sleep(sleepTime);
              }
        }
        catch(InterruptedException e) {
              // Lancé si le thread est interrompu
        }
  }

  public static void main(String [] args) {
        new SimpleThread(1500).start();
        new SimpleThread(1100).start();
        new SimpleThread( 600).start();
  }
}
```

**Exécution**

```
Thread-0: 5
Thread-1: 5
Thread-2: 5
Thread-2: 4
Thread-1: 4
Thread-2: 3
Thread-0: 4
Thread-2: 2
Thread-1: 3
Thread-2: 1
Thread-0: 3
Thread-1: 2
Thread-1: 1
Thread-0: 2
Thread-0: 1
```

**Exemple 3**

```
public class MyThread extends Thread {


    public MyThread()
    {
        super();
        start();
    }


    @Override
    public void run() {
        while(true)
        {
            try {
                // mise en sommeil
                Thread.sleep((int)(Math.random() * 1000));
            }
            catch (InterruptedException e) { e.printStackTrace();  }
        }
    }

    // remarque: boucle infinie dans le run donc Arret via le stop dans la console
    public static void main(String[] args) {
        MyThread myThread = new MyThread();

        System.out.println("Mise en sommeil du thread");
        myThread.run();
    }
}
```

## Arrêter (stop) un thread

• La classe **Thread** déclare une méthode ~~**stop()**~~ permettant d'arrêter un thread.

• ~~**stop()**~~est maintenant déprécié:

***L'arrêt était brutal, sans prise en compte de l'action de l'objet en cours, pouvant mener a un état incohérent.***

***Les verrous acquis n'étaient pas libérés, menant à des situations d'interblocage.***

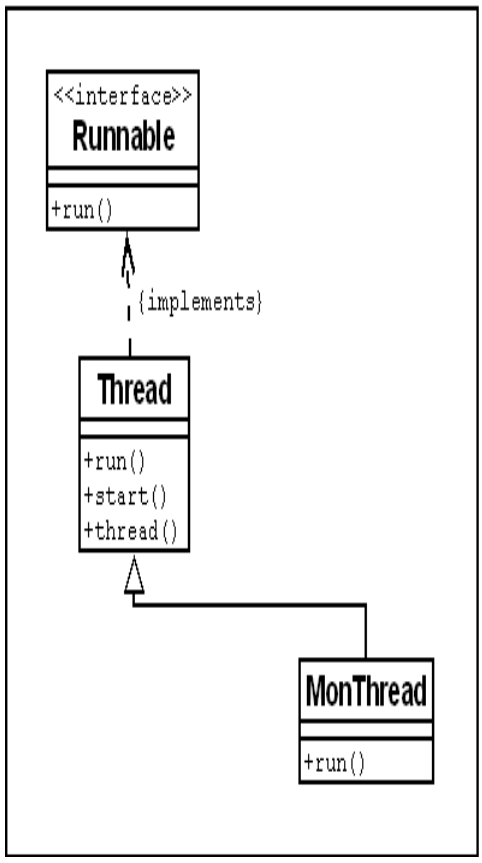
• Le modèle actuel est de demander au thread de s'arrêter, utilisant les méthodes qu'il a prévu pour le faire.

```
public class ThreadQuit implements Runnable {
        long sleepTime;
        Thread localThread;
        boolean keepGoing = true;

        ThreadQuit(long sleepTime) {
                this.sleepTime = sleepTime;
                localThread = new Thread(this);
                localThread.start();
        }

        public void run() {
                try {
                        for (int n = 5; n > 0; n--) {
                                System.out.println( localThread.getName() + ": " + n);
                                if ( !keepGoing )
                                        break;
                                Thread.sleep(sleepTime);
                        } }
                catch(InterruptedException e) { }
        }

        public void quit() {
                keepGoing = false;
        }

        public static void main(String [] args) throws InterruptedException {
                ThreadQuit q = new ThreadQuit(1000);

                System.out.println("tâche principale mise en sommeil...");
                Thread.sleep(5000); // essais: 100 1000 10000
                q.quit();
                System.out.println("Thread stoppé!");
        }}
```

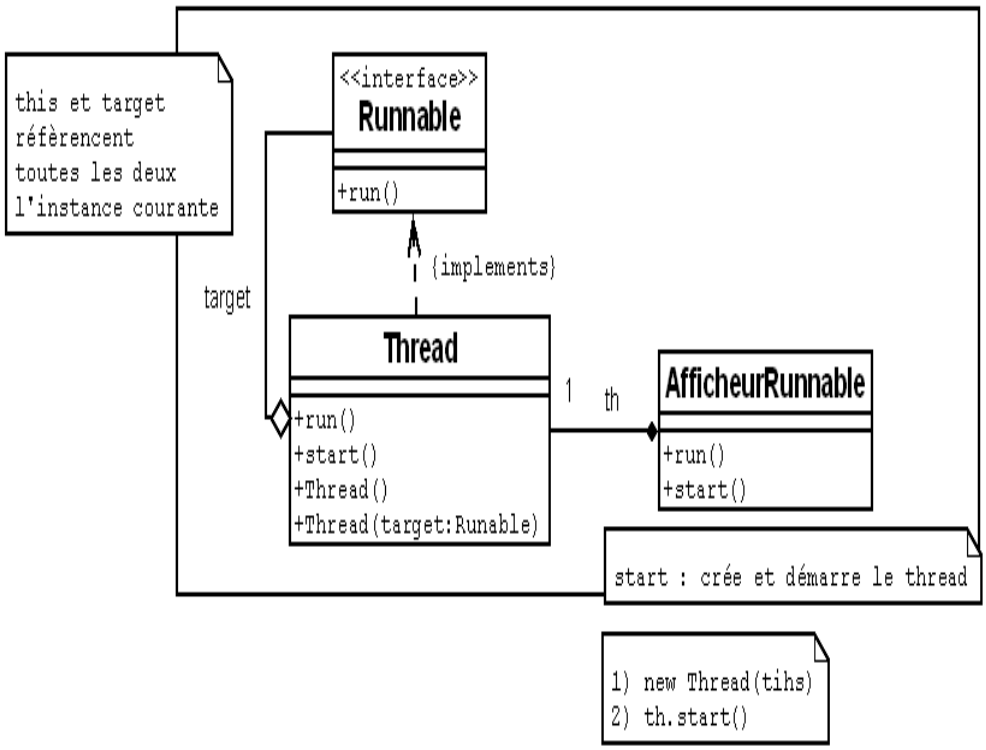
# MODELE 2: Implémenter l'interface Runnable

- En général on préfère implémenter Runnable.
- Conceptuellement, un objet n'est probablement pas un type de Thread
- Mais un objet qui se comporte comme un thread c-à-d Runnable.
- De plus, on a souvent besoin de dériver d'une autre classe que Thread, et Java ne permet pas l'héritage multiple.


modèle 1




modèle 2



- **L'interface Runnable** permet de **créer un objet, que l'on utilisera ensuite comme constructeur d'un Thread**.
- C'est la **méthode start() de l'objet** qui sera **responsable de la création et de l'activation du thread.**
- **Le thread est construit dans start ()** : **p = new Thread (this)**

**la méthode run () de ce thread lance la méthode run de la classe de l'objet p (implémentant Runnable).**

**<u>En fait, dans la classe Thread on a :</u>**

***L' attribut target est une référence sur this de la classe implémentant l'interface runnable***

```
private Runnable target ;
public void run()
{
        if (target != null)
                target.run();
}
```

**Exemple**

```
class AfficheurRunnable implements Runnable
{
        String name;
        long sleepTime;
        Thread localThread;;
        boolean keepGoing = true;


        public AfficheurRunnable ( String name, long sleepTime )
        {
                this.name = name;
                this.sleepTime = sleepTime;
        }

        public void start ()
        {
                if (localThread == null)
                {
                        localThread = new Thread  (this, name);
                        localThread.start();
                }
        }

        public void stop() {
                keepGoing = false;
        }


        public void run ()
        {
                try {
                        for (int n = 5; n > 0; n--) {
                                System.out.println( localThread.getName() + ": " + n);
                                if ( !keepGoing )
                                        break;
                                Thread.sleep(sleepTime);
                        } }
                catch(InterruptedException e) { }
        }
```

```
	public static void main ( String args[] )
	{
		AfficheurRunnable run1 = new AfficheurRunnable("run1", 1000);
		AfficheurRunnable run2 = new AfficheurRunnable("run2", 2000);

		run1.start();
		run2.start();

		System.out.println("tâche principale mise en sommeil...");
		try {
			Thread.sleep(1000); // essais 100 1000 3000 5000 10000
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		run1.stop();
		System.out.println("run1 stoppé!");
	}
}
```

# III Exclusion mutuelle

## Partager des ressources

- Dans un programme **uni-thread**, tous les éléments du programme appartiennent à ce thread, donc **pas de problème de partage de ressources**
    - Mémoire, interface graphique, fichiers, sockets, …
- Dans un programme **multi-thread,** plusieurs threads peuvent essayer d'**accéder à un ressource en même temps.**
    - Écrire vers un fichier, imprimer des documents, retirer et déposer dans un compte bancaire, ...
- **En Java, on encapsule un ressource à partager dans un objet**
- Le langage fournit un **mécanisme de verrou (monitor)** qui permet de **synchroniser les accès concurrents à l'objet.**

## Construction d'une classe moniteur

- On définit une **classe "moniteur"** contenant une **méthode synchronized** qui va permettre un accès non interruptible à une ressource partagée.
- Lorsque cette méthode est **appelée dans un thread**, elle **bloque le déroulement des autres threads** jusqu'à ce qu'elle soit arrivée à la fin de son traitement.
- Les méthodes déclarées avec le modificateur synchronized obligent les threads à prendre en compte l'**état du moniteur d'une instance**
- Le mécanisme du moniteur assure donc qu' un seul thread ne peut manipuler une instance via une méthode synchronisée. Si un **autre thread** tente d'appeler la méthode synchronisée sur la même instance, il est **désactivé et placé dans une file d'attente**.

**Attention : pour que ce mécanisme fonctionne, il faut que les threads partagent le même moniteur. Cela se traduit par la définition, dans la classe des threads, d'un attribut static pour contenir le moniteur sur lequel on invoquera les méthodes synchronisées.**

## Sécurisation d'une méthode

• Lorsque l'on crée une instance d'une certaine classe, on crée également un **moniteur** qui lui est associé. Lorsque l'on applique le **modificateur synchronized**, on place la méthode (le bloc de code) dans ce moniteur, ce qui assure l'**exclusion mutuelle (mutex)**

• Lorsqu'une méthode synchronized est invoquée, elle ne s'exécute que lorsqu'elle a obtenu un **verrou sur l'instance** à laquelle elle s'applique ; elle garde alors le verrou pendant toute son exécution.

• Les instructions qui suivent le mot-clé synchronized constituent une **section critique**.

**Exemple**

```
class MutexAccumulateur
{
        int accumulateur = 0;

        public synchronized void stocke ( int val )
        {
            accumulateur += val;
        }

        public int lit ()
        {
            return accumulateur;
        }
}
```

**L'opération d'incrémentation += prend plusieurs instructions.**

Un **changement de contexte** entre ces instructions pourrait créer une **incohérence des résultats**.

**Rappels:**

- Le modificateur synchronized indique que la méthode stocke fait partie du moniteur de l'instance, empêchant son exécution simultanée par plusieurs threads. Si le moniteur est déja occupé, les threads suivants seront mis en attente.
- L'ordre de réveil des processus n'est pas déterministe.

## Sécurisation de blocs de code

- **L'utilisation de méthodes synchronisées trop longues peut créer une baisse d'efficacité.**
- Il est possible de placer n'importe quel bloc dans un moniteur, ce qui permet ainsi de **réduire la longueur des sections critiques**.

**Remarque:** dans l'exemple suivant, le code de methode1 et de methode2 est équivalent.

```
synchronized void methode1()
{
    // section critique...
}
```

```
void methode2()
{
        synchronized ( this )
        {
                // section critique...
        }
}
```

**Exemple: La célébrissime Tortue du langage Logo (Seymour Papert)**

```
public class Tortue
{
        private Point pos;
        private int angle;

        public Tortue ( int x, int y, int angle ) { // ... }

        public synchronized void tourne ( int degres )
        {
                angle += degres;
        }

        public synchronized void avance(int distance)
        {
                pos.x += (int) ((double)distance*Math.cos((double)angle));
                pos.y += (int) ((double)distance*Math.sin((double)angle));
        }

        public int angle() {return angle; }

        public synchronized Point pos() {return new Point(pos.x,pos.y); }
    }

public class mutexLogo
{
        public int lectureEntier ()
        {
                // ...
                // lecture d'un nombre entier
                // pouvant prendre du temps
                // ...
        }

        public static void carre(Tortue tortue)
        {
                int largeur = lectureEntier();
                int longueur = lectureEntier();
                synchronized (tortue)
                {
                        tortue.tourne(90);tortue.avance(largeur);
                        tortue.tourne(90);tortue.avance(longueur);
                        tortue.tourne(90);tortue.avance(largeur);
                        tortue.tourne(90);tortue.avance(longueur);
                }
                // autres taches...
        }
    }
```

• **rappel:** Quand le modificateur synchronized qualifie une méthode, on place cette méthode dans le moniteur de l'instance. Cela permet de sécuriser l'accès à une instance particulière d'un objet.

• En créant un bloc **synchronized (tortue),** on entre dans le moniteur associé à l'instance tortue. Dans l'exemple ci-dessus, si un thread est en train d'afficher un carré en faisant appel à mutexLogo.carre(tortue), un autre thread ne pourra pas déplacer la tortue.

• Malgré tout, pendant que l'on se trouve dans le méthode carre et que l'on est en train de lire les dimensions du carré, un autre processus pourra utiliser la tortue. En effet, le moniteur n'est pas occupé, parce que la méthode n'est pas synchronized, seul un bloc l'est.

## Sécurisation des variable de classes

- Considérons maintenant le cas où il faut **sécuriser l'accès à une variable de classe.**
- **La solution est simple, il faut créer un moniteur qui est commun à toutes les instances de la classe.**
- La **méthode getClass()** retourne la classe de l'instance dans laquelle on l'appelle.
- Il suffit alors de créer un moniteur qui utilise le résultat de getClass() comme "verrou".

**Exemple:**

- Une classe d'accumulateur est définie. Elle incrémente une **variable de classe acces** chaque fois que l'on accède à stocke ou à lit.
- La variable acces est une variable de classe (déclarée public), elle est donc partagée par les différentes instances de cette classe.
- **La méthode getClass() retourne un objet de type Class avec lequel on crée un nouveau moniteur**

```
class mutexStatic
{
        private int accumulateur = 0;
        private static int acces = 0;

        public synchronized void stocke(int val)
        {
                accumulateur += val;
                synchronized ( getClass() )
                {
                        acces += 1;
                }
        }

        public int lit()
        {
                synchronized ( getClass() )
                {
                        acces += 1;
                }
                return accumulateur;
        }

        public int acces() {return acces;}
}
```

# IV Synchronisation : les méthodes wait() et notify()

**Il peut être nécessaire de synchroniser des processus qui accèdent aux mêmes ressources.**

L'utilisation des moniteurs permet de garantir l'exclusion mutuelle, et pour la synchronisation, on utilisera des **signaux** modélisés par les méthodes **wait()** et **notify()** **de la classe Object.**

**public final void wait() ;** le thread s'arrête jusqu'à ce qu'il soit notifié
**public final void wait(long millisec) ;** idem ou attente du délai passé en paramètre
**public final void notify() ;** un thread en attente a une chance de tourner
**public final void notifyAll() ;** tous les threads en attente ont une chance de tourner

• Il existe deux variantes de la méthode **wait()** qui permettent de spécifier un temps limite après lequel le processus sera réveillé, sans avoir à être notifié par un processus concurrent.

**wait(long milli)**, qui attend milli millisecondes

**wait(long milli,int nano)** qui attend nano nanosecondes en plus du temps milli.

• Il n'est pas possible de savoir si **wait()** s'est terminé à cause d'un appel à **notify()** par un autre processus, ou de l'épuisement du temps.

• La méthode **notifyAll()** **réveille tous les processus**, **synchronisés sur une instance donnée, et dès que le moniteur sera libre, ils se réveilleront tour à tour.**

• Le noyau Java ne fait **aucune garantie concernant l'élection des processus lors d'un appel** à **notify().** En particulier, il ne garantit pas que les processus seront débloqués dans l'ordre ou ils ont été bloqués. C'est à cause de cela que l'on a placé **wait()** dans une boucle dans l'exemple suivant car un consommateur pourrait réveiller un autre consommateur alors que le tampon est vide.

• Remarquons aussi que les méthodes **wait(), notify() et notifyAll()** ne peuvent être appelées que dans des méthodes synchronisées (synchronized).

## Scénario Producteur/Consommateur

- Une mémoire tampon est gérée par un producteur d'objets et un consommateur.
- Cette mémoire ne peut contenir qu'un objet.
- On synchronise l'accès à cette mémoire (production et consommation non simultanées).
- Si le producteur doit déposé un objet alors qu'il en existe déjà un dans la mémoire tampon, il attends ... blocage !
- Même type de blocage pour un consommateur qui attend le dépôt d'un objet.

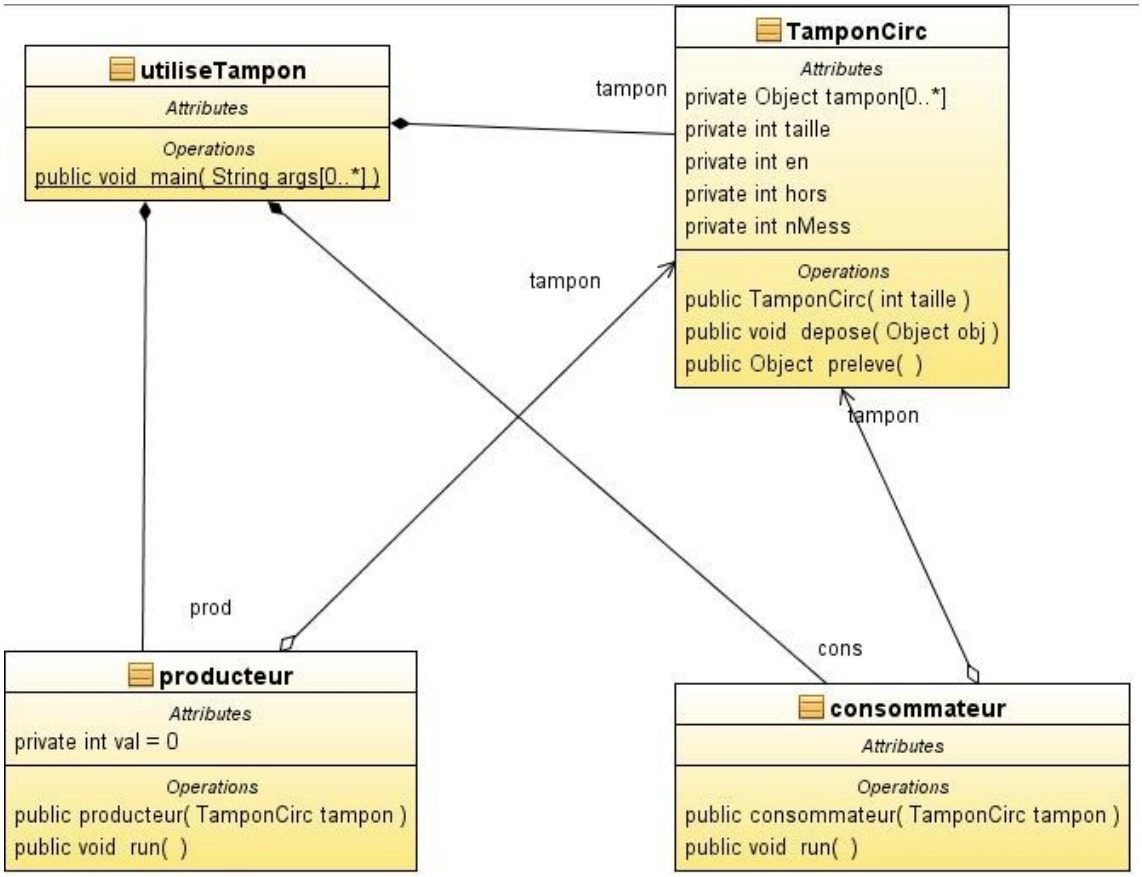
## Relâchement d'exclusion

**La solution du problème précédent passe par la mise en oeuvre d'une procédure de relâche d'exclusion.**

- L'attente du Producteur ou du Consommation se fait au moyen de la méthode wait() (susceptible de déclencher une exception InterruptedException) qui relâche l'exclusion sur l'objet.
- Un processus sort de l'attente lorsqu'un autre processus exécute la méthode notify() (qui relance *un* processus en attente) ou la méthode notifyAll() (qui relance *tous* les processus en attente)

**Prenons un tampon borné de n objets, un processus producteur et un processus consommateur.**

**Les données sont produites plus vite que le consommateur ne peut les prélever, à cause de la différence de durée des délais (aléatoires) introduits dans les 2 processus.**



**Exemple – Tampon circulaire**

```
class tamponCirc {
        // un conteneur d'objets
        // relation de composition
        private Object tampon[];
        private int taille;
        private int en, hors, nMess;

        /** constructeur. crée un tampon de taille éléments */
        public tamponCirc (int taille) {
                tampon = new Object[taille];
                this.taille = taille;
                en = 0;
                hors = 0;
                nMess = 0;
        }

        public synchronized void depose(Object obj) {
                while (nMess == taille) {    // si plein
                        try {
                                wait();              // attends non-plein
                        } catch (InterruptedException e) {}
                }
                tampon[en] = obj;
                nMess++;
                en = (en + 1) % taille;
                notify();              // envoie un signal non-vide
        }

        public synchronized Object preleve() {
                while (nMess == 0) {    // si vide
                        try {
                                wait();          // attends non-vide
                        } catch (InterruptedException e) {}
                }
                Object obj = tampon[hors];
                tampon[hors] = null;    // supprime la ref a l'objet
                nMess--;
                hors = (hors + 1) % taille;
                notify();              // envoie un signal non-plein
                return obj;
        }

}
```

```
class producteur extends Thread {

	// relation d'association
	private tamponCirc tampon;
	private int val = 0;

	public producteur(tamponCirc tampon) {
		this.tampon = tampon;
	}

	public void run() {
		while (true) {
			System.out.println("je depose "+val);
			tampon.depose(new Integer(val++));
			try {
				Thread.sleep((int)(Math.random()*100));    // attend jusqu'a 100
ms
			} catch (InterruptedException e) {}
		}
	}
}

class consommateur extends Thread {

	// relation d'association
	private tamponCirc tampon;

	public consommateur(tamponCirc tampon) {
		this.tampon = tampon;
	}

	public void run() {
		while (true) {
			System.out.println("je preleve "+((Integer)tampon.preleve()).toString());
			try {
				Thread.sleep((int)(Math.random()*200));    // attends jusqu'a
200 ms
			} catch (InterruptedException e) {}
		}
	}
}
```

```
class utiliseTampon {

        public static void main(String args[]) {

                tamponCirc tampon = new tamponCirc(5);
                producteur prod = new producteur(tampon);
                consommateur cons = new consommateur(tampon);

                prod.start();
                cons.start();
                try {
                        Thread.sleep(30000);    // s'execute pendant 30 secondes
                } catch (InterruptedException e) {}
        }

}
```

**Exécution :**

```
...
je depose 165
je depose 166
je preleve 161
je depose 167
je preleve 162
je depose 168
je preleve 163
je depose 169
je preleve 164
je depose 170
je preleve 165
je depose 171
je preleve 166
je preleve 167
...
```