Threads & cie; week 3

### 1/ Running & controlling tasks with threads

#### Basic Operations

* In order to execute the main method of the class, the JVM creates/calls : currentThread()
* creating a thread by providing a Runnable object is more flexible. It permits subclassing a thread, but also is less limited.
* **interleave =** depends on the speed & the number of processes on the computer & the OS
* **predictability =** precise scheduling of threads cannot be predicted
* The main terminates before the other threads. main terminating ⇒ executing threads terminates.
* 2 ways to instantiate a lambda expression :
  + Thread thread1 = new Thread(() -> {}, name);
  + Runnable runnable = () -> {}

Thread thread1 = new Thread(runnable, name);

#### b) Waiting for Tasks to Complete

While using Thread.join() you have to throw **InterruptedExcpetion** !

#### c) Executing Periodical Tasks

**periodical task =** task executed at a specified time intervals, repeated, minimal human intervention. (ex : monitoring tasks)

!! While using sleep method, you have to try catch **InterruptedException** !

* They both throw InterruptException because they block the thread. So when you try to interrupt those methods, it throws the exception in order to wake those methods.

Every blocking method of the thread class has to throw an exception.

#### d) Stopping tasks

**Thread.stop()** → inherently error-prone.

⇒ instead; notify to the thread it has to stop; it’ll stop at an appropriate time; avoiding

to stop when it is updating some shared var for example.

### 2/ Running and controlling tasks with executors

Using Executors > using threads

### Basic operations

**Executor interfaces :**

* **Executor :** simple, supports launching new tasks
* provides a single method : e.execute(r)
  + replacement for a thread-creation idiom.
  + less specific : use an existing worker thread to run the new one, or places the new one in a queue to wait for a worker thread to become available
  + java.util.concurrent
* **ExecutorService :** subinterface of Executor, helps managing the lifecycle of individual tasks/the executor
* new method : submit

→ accepts runnable & callable objects (allows the task to return a value)

→ returns a Future object

* returns Callable return value
* manages the status of Callable & Runnable
* methods managing the shutdown of executor.
* **ScheduledExecutorService :** subinterface of ExecutorService, supports future/periodic execution of tasks.
* schedule method : executes runnable or callable after a **specified delay**.
* methods scheduleAtFixedRate / scheduleWithFixedDelay ⇒ executes a specified task repeatedly with defined intervals.

**Thread pools :**

* worker threads. Used to execute multiple tasks
  + minimizes the overhead due to thread creation.

**Fixed thread pool :** java.util.concurrent.Executors && java.util.concurrent.ExecutorService

* specified number of threads running.
* if a thread is terminated → replaces it with a new thread
  + tasks submitted with an internal queue, holding extra tasks. (more active tasks > threads)
* see also : newCachedThreadPool (expandable thread pool), newSingleThreadExecutor (executor executing a single task a time).
* eg : ExecutorService exec = Executors.newFixedThreadPool(nbOfTasks);

1. Waiting for tasks to complete
2. Stopping tasks

### 3/ Synchronizing tasks at low level

1. Motivation : memory consistency errors
2. Motivation : thread interference - lost updates
3. Motivation : thread interference - inconsistent reads