*Notes from the book Java Concurrency…*

***Chap VI***

***Executor (Interface):***

    The basic interface for the used for any concurrency system

                      Has one major method (to be implemented by concrete subclass)

***void execute(Runnable command)***

 Too limited interface to be used (unlike its implementation below)

***Callable<V>:***

Variation of ***Runnable***, where a ***call()*** method is called instead by the ***ExecutorService*** (below).

        public MyCallable extends Callable<AType>{

            public AType call() throws Exception {

                //Work to do when ExecutorService

               //calls me

            }

       }

***Future<AType>:***

         Response object returned by some ExecutorService implementation,

                                on submit (). It main method is:

                                <AType> get   Return the result of the Callable object submited

(see ***ExecutorService***example)

                                                                                or a timeout  version

<AType>  get(long timeout, TimeUnit unit)   Wait to a maximum of Timeout time and leave

*Future<?> future = service.submit(new MyCallable());*

*try {*

*future.get(****100, TimeUnit.MILLISECONDS****);*

*} catch (TimeoutException e){*

*//The task took too long*

*}*

**Note**: The **get()** method blocks the calling thread till a result is returned by the task we are waiting…

         Has a ***isDone()*** to tell if the assigned task is completed

         Cancellation:  Has a ***isCancelled()*** to know if task is cancelled,

And a ***cancel()*** method, to stop the associated Runnable or Callable

***ExecutorService(Interface):***

The first interface having enough function to build operation concurrency.

         Has submit methods for regular **Runnable**

or ***Callable*** objects (See example below)

Future future = executorService.submit(new Callable(){

    public Object call() throws Exception {

        System.out.println("Asynchronous Callable");

        return "Callable Result";

    }

});

…

(and result retreiving done later by)

               future.get() = Callable Result

         Stopping the service:

The **shutdown**() method will try to kill the service

The **isShutdown**() method can tell is the service is already shutdown

***Exectuors:***

Factory and utility methods to instantiate the various type of ***ExecutorService***

First step of all concurrent problem is calling the Execturos factor to get

                     The Exector(Service) we want

  Methods availables:

         newSingleThreadExecutor()

Creates an Executor that uses a single worker thread

         newFixedThreadPool(int nThreads)

Creates a thread pool that reuses a fixed number of threads operating off a

shared unbounded queue

         newCachedThreadPool()

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

         newScheduledThreadPool(int corePoolSize)

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

***ScheduledExecutorService:***

Permit the scheduling of a task, instead of just submitting like the ***ExecutorService***

or straight execute of the ***Executor***.

         ***schedule***(Runnable command, long delay, TimeUnit unit)

Creates and executes a one-shot action that becomes enabled after the given delay.

         ***scheduleAtFixedRate***(Runnable command, long initialDelay, long period, TimeUnit unit)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then repeat at each period…

All methods return a **ScheduledFuture**, instead of a Future like the Executor.

Example:

ScheduledExecutorService scheduler =  Executors.newScheduledThreadPool(NBR\_THREADS\_ALLOWED);

final Runnable beeper = new Runnable() {

            public void run() { System.out.println("beep"); }

};

ScheduledFuture<?> singleBeeper =  scheduler.schedule(beeper, 20, TimeUnit.SECONDS);

ScheduledFuture<?> periodicBeeper =  scheduler.scheduleAtFixedRate(beeper, 2, 2, TimeUnit.SECONDS);

***CompletionService:***

                Wrap an Executor and uses a BlockQueue to manage the submitted task.

         Submit method is like regular ExecutorService

submit(Callable<V> task)

         ***take***() retrieve the next completed task

Blocks the calling thread is not task completed

         ***poll***(), like the take method, but returns null and doesn’t block

is no completed task…

Example:

ExecutorService executor = Executors.newFixedThreadPool(9);

CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);

 for (final String printRequest : printRequests) {

      completionService.submit(new Callable(){

                        public String call() throws Exception {

                            return "Dummy String!";

                        }

      );

}

 try {

       for (int t = 0, n = printRequests.size(); t < n; t++) {

             Future<String> f = completionService.take();

             System.out.print(f.get());

       }

} catch (InterruptedException e) {

     Thread.currentThread().interrupt();

} catch (ExecutionException e) {

     Thread.currentThread().interrupt();

}

 executor.shutdownNow();

Chap VII

***Thread Interruption:***

                Each thread has a interrupted flag”

         ***isInterrupt***() give the status of that flag

         ***interrupt***() set the flag to true

        Does **not stop the thread**, but **ask** it to stop

         ***interrupted***() clears the interrupt flag

***Usefull Tips:***

Gettting the current Thread

         ***Thread.currentThread()***

***InterruptedException:***

                Thrown when an already blocked thread execute another wait function,

                or use an object that will invoke a blocking code (like a BlockingQueue blocks

when we invoke the get() method and there is no item in the queue).

                Rule: A)Do something when it happens

or

          B) Propagate it to your calling throw an InterruptedException yourself

                                or

         C) Reset the interrupted flag by calling ***interrupt***()

                (This way, code in the execution stack may know what to do)

    Never “eat” this exception

***Future object and  Cancellation:***

                The Future object is returned by an ExecutorService when we submit a task.

                It can be used to get the result or cancel the task, if possible

         ***get***() will blocks, till it get the result of the task

         ***get***(long timeout, TimeUnit unit) , will block till the timeout expires

Throws a ***TimeoutException*** then

Waits if necessary for the computation to complete, and then retrieves its result.

         Can try to cancel the task, if not started already or completed

 ***cancel***(boolean mayInterruptIfRunning) `

         ***isCancelled***(), tells if the associated task is cancelled

         ***isDone***(), tells if the task is completed