Efficient Java Multithreading – Using Executors to do everything that you can do with Threads!

- By Arun K. Mehra

Disclaimer

- This presentation does **not** cover the *basics* of Java Multithreading
- It's focus is primarily on the *Executors Framework*
- It is used in the online video course on Udemy – "Efficient Java Multithreading with Executors"
- **It's probably a bit boring as it contains a lot of text but no funny images and no code listings** ☹
- The matter in the slides is actually explained in the video course (sample videos can be viewed on You-Tube or Udemy)
- Still strong? Let's roll! ☺

About me!

- My name is ***Arun Kumar Mehra***.
- Nearly 13 years in Software Industry architecting and developing Enterprise applications
- Worked in diverse domains – Life Insurance, Investment Banking and Telecommunications
- Mostly used Java/JEE technologies
- Diehard fan of Open Source especially, Java, Groovy, OSGi and Eclipse RCP/RAP

Pre-requisites

Good knowledge of Java and basic knowledge of multithreading required …

- Why multithreading
- Scenarios where multithreading is useful (and where it is not!)
- Difference between thread-of-execution and Thread class
- Various execution-thread states
- Thread class operations
- Etc.

Course Structure

01 Introduction
02 Creating and Running threads
03 Naming threads
04 Returning values from threads
05 Creating daemon threads
06 Checking threads for ‘life’
07 Terminating threads
08 Handling Uncaught Exceptions
09 Waiting for other threads to terminate
10 Scheduling tasks

02. CREATING & RUNNING THREADS

Starting Threads – Basic Mechanism

To run a task in a separate thread-of-execution, three things are required:

- ***Thread*** class object
- ***Runnable*** interface implementation or 'The Task'
- Invocation of ***start()*** method on the Thread object

Creating Threads Using Threads API

Quite a few ways. Difference lies in:

- The mechanism using which:
  - The Thread object is created
  - The Task is created
- The way these two objects are brought together and made to collaborate

Running Threads Using Executors API

Only one way!

- Create task definition class
- Provide task object to an Executor Service

Executors API Overview

Preferred way of running tasks...

- Uses thread-pools
- Allocates heavy-weight threads upfront
- Decouples task-submission from thread-creation-and-management
- Each thread in the pool executes multiple tasks one-by-one
- A tasks-queue holds the tasks
- Threads are stopped when the Executor Service is stopped

Important Classes/Interfaces

Some frequently referred classes/interfaces while using Executors framework:

- Executor (I)

```
void execute (Runnable task);
```

Important Classes/Interfaces

Some frequently referred classes/interfaces while using Executors framework:

- Executor (I)
- ExecutorService (I)

```
<T> Future<T> submit (Callable<T> task);
Future<?> submit (Runnable task);
void shutdown ( );
List<Runnable> shutdownNow ( );
boolean isShutdown ( );
...
```

Important Classes/Interfaces

Some frequently referred classes/interfaces while using Executors framework:

- Executor (I)
- ExecutorService (I)
- Executors (C)

```
public static ExecutorService newFixedThreadPool (int nThreads);
public static ExecutorService newCachedThreadPool ( );
public static ExecutorService newSingleThreadExecutor ( );
public static ExecutorService newSingleThreadScheduledExecutor ( );
...
```

Important Classes/Interfaces

Some frequently referred classes/interfaces while using Executors framework:

- Executor (I)
- ExecutorService (I)
- Executors (C)
- Future (I)

```
V get ( ) throws InterruptedException , ExecutionException;
boolean isDone ( );
...
```

Available Thread-Pools

Many types of thread-pools available out-of-the-box:

- Fixed Thread Pool
- Cached Thread Pool
- Single Thread Executor (technically not a ‘pool’)
- Scheduled Thread Pool
- Single Thread Scheduled Executor (technically not a ‘pool’)

03. NAMING THREADS

What's In A Name?

Why would you want to name threads …

- 'Thread-0', 'Thread-1', 'Thread-2' – not very helpful in identifying the threads
- More useful names would be:
  - Directory-Scanner-Thread
  - Timer-Thread
  - Cache-Invalidating-Thread
- Helps in identifying special threads while debugging multi-threaded applications

Naming Normal Threads

Java provides a couple of opportunities …

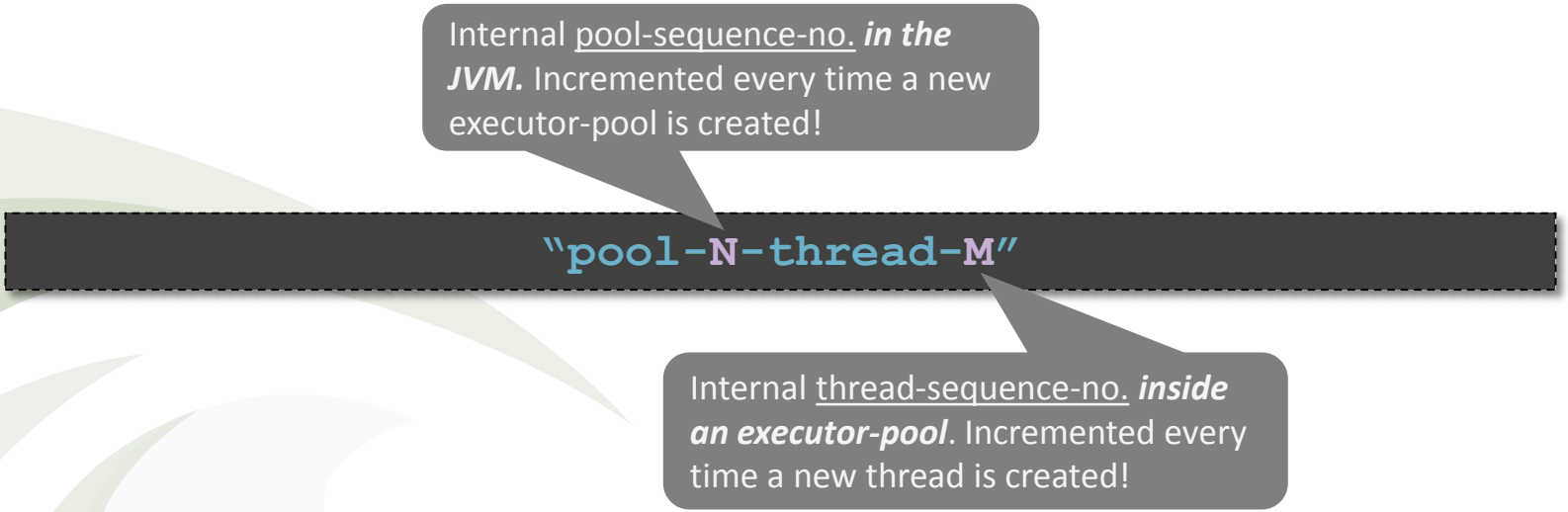
- From within the *task* – at ‘thread-running’ time
- At ‘thread-creation’ time

Naming Executor Threads

Two ways …

- From within the *task* - at ‘thread-running’ time. Similar to the technique that was used for renaming normal threads.
- Specify the thread-naming strategy at ‘executor-creation’ time

Default Names of Executor Threads



04. RETURNING VALUES FROM THREADS

Returning Values Using Normal Threads

CTM: Values are returned from *Tasks* and not *Threads*!

- No language level support in normal Threads but there is support in Executors
- A few ways for normal threads – see code

Returning Values Using Executors

Out-of-the-box support provided by Java …

- Use ***Callable<T>*** instead of *Runnable*
  - Override method: ***public T call() throws Exception***
- *ExecutorService.**submit**(Callable c)*
- Call to *submit()* returns a ***Future<T>***
- Task result can be retrieved using "***T Future.get()***" method

Processing Tasks' Results in Order of Completion

Use the following …

- CompletionService<V> (I)
- ExecutorCompletionService<V> (C)

```
Future<V> submit (Callable<V> task)
Future<V> submit (Runnable task, V result)
Future<V> take ( )
Future<V> poll ( )
```

05. DAEMON THREADS

Daemon Threads – What and Why?

Background and/or service providing threads …

- Killed by the JVM as soon as no user thread is running any longer
  - Can be stopped normally too – just like normal threads
- *'main'* thread is a 'user' thread – not 'daemon'
- Any kind of logic as desired can be put in daemon threads. E.g.:
  - Directory-watcher-thread
  - Socket-reader-thread
  - Long-calculation-thread

Creating ‘Daemon’ Threads

Thread class provides a method …

- *void Thread.**setDaemon**(boolean on)*
  - Also used in Executors API
- For Executors, implement a ThreadFactory and manipulate the thread there to make it ‘daemon’

06. CHECKING THREADS FOR ‘LIFE’

Checking Threads for ‘Life’ - Threads API

A thread is **live** if it has started but not terminated yet!!!

- Use the method: ***boolean Thread.isAlive()***
- Will return *true* if:
  - … the thread is still running
- Will return *false* in the following scenarios:
  - If the thread has not been started yet i.e *Thread.start()* has not been called yet or;
  - If the thread has terminated – irrespective of whether it has successfully finished executing its task or not

Checking Threads for 'Life' – Executors API

Are we really interested in a thread being **live** …???

- Actually we are interested in task completion!!!
- Use the method: ***boolean Future.isDone()***
- Will return *true* if:
  - … the task execution is completed - whether normally, or with exception or with cancellation
- Will return *false* if:
  - … the task execution has not been completed yet – in fact, it may not even have started yet!

This deck is part of the full-fledged video course on :

EFFICIENT JAVA MULTITHREADING WITH EXECUTORS

Visit: Course

07. TERMINATING THREADS

Terminating Normal Threads

There are a few ways …

- Terminating at non-blocked portions in the task:
  - Using a method coded for this purpose in the task
  - Using interrupts:
    - ✓ ***void Thread.interrupt()*** – usually called by another thread on the thread-to-be-interrupted
    - ✓ ***static boolean Thread.interrupted()*** – to be called from inside the interrupted thread
    - ✓ ***boolean Thread.isInterrupted()*** – to be called by another thread on the interrupted thread
- Terminating at points in the task where the thread is blocked:
  - Using interrupts – same methods as above

Terminating Individual Executor Tasks

There are a few ways …

- Terminating at non-blocked portions in the task:
  - Using a method coded for this purpose in the task
  - Using interrupts:
    - ✓ ***boolean Future.cancel(boolean mayInterruptIfRunning)*** – to be called by the class holding the Future ref.
    - ✓ ***static boolean Thread.interrupted()*** – to be called from inside the interrupted task
    - ✓ ***boolean Future.isCancelled()*** – to be called on the Future outside the interrupted task
- Terminating at points in the task where the thread is blocked:
  - Using interrupts – same methods as above

Terminating All Executor Tasks in One-Shot

Uses the same concepts as discussed previously …

- Use: ***List<Runnable> ExecutorService.shutdownNow()***
  - Uses interrupts under the hood
  - Returns a list of tasks that were awaiting execution yet
- Interrupting may terminate blocked as well as non-blocked tasks
- Tasks must be coded to properly handle the interrupts
- To await termination after shutting down, use:
  - ***boolean ExecService.awaitTermination(long timeout, TimeUnit unit)***

08. HANDLING UNCAUGHT EXCEPTIONS

Uncaught Exceptions in Threads

Leaking Exceptions from threads cannot be caught directly …

- Implement ***Thread.UncaughtExceptionHandler*** interface
  - Only one method: ***void uncaughtException(Thread t, Throwable e)***
- Three ways to use *UncaughtExceptionHandler*:
  - Set as default handler for all the threads in the system
    - ✓ ***static void Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler eh)***
  - Set different handlers for different threads
    - ✓ ***void Thread.setUncaughtExceptionHandler(UncaughtExceptionHandler eh)***
  - Use a combination of default-handler and thread-specific-handlers

Uncaught Exceptions in Executors

Similar ways of handling like the normal threads …

- Implement ***Thread.UncaughtExceptionHandler*** interface
  - Only one method: ***void uncaughtException(Thread t, Throwable e)***
- Three ways to use *UncaughtExceptionHandler*:
  - Set as default handler for all the threads in the system
    - ✓ ***static void Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler eh)***
  - Set different handlers for different threads
    - ✓ Implement a ThreadFactory and;
    - ✓ Use ***void Thread.setUncaughtExceptionHandler(UncaughtExceptionHandler eh)***
  - Use a combination of default-handler and thread-specific-handlers

09. WAITING FOR OTHER THREADS TO COMPLETE

Waiting for Normal-Threads to Finish

Suspend the current thread so that it continues only after some other thread(s) have finished …

- Previously discussed techniques:
  - ***boolean Thread.isAlive()***
    - ✓ To be called multiple times till you get a 'false'
    - ✓ Wastage of CPU cycles
  - ***wait() and notify() mechanism***
    - ✓ To be coded manually in the tasks by the programmer
    - ✓ A little difficult to get right
- Third technique:
  - ***void Thread.join()***
    - ✓ In-built mechanism – works just like wait() and notify()
    - ✓ Easy to use

Waiting for Executor Tasks to Finish

Suspend the current task so that it continues only after some other task(s) have finished …

- Use ***java.util.concurrent.CountDownLatch***
- An object of *CountDownLatch* is shared by the waiting tasks and the awaited tasks.
  - Waiting-tasks call ***void await()*** on the latch
  - Awaited-tasks call ***void countdown()*** on the latch after finishing their executions
  - When the count reaches zero, waiting tasks are released, thereby, bringing them out of suspended state and enabling their execution again.

10. SCHEDULING TASKS

Scheduling Tasks using Threads-API

For all scheduling needs, Threads-API provides two classes …

- ***java.util.Timer***
  - Spawns a thread for executing the tasks
  - Also contains scheduling logic

```
void schedule(TimerTask task, Date time)
void schedule(TimerTask task, long delay)
void schedule(TimerTask task, Date firstTime, long period)
void schedule(TimerTask task, long delay, long period)
void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)
void scheduleAtFixedRate(TimerTask task, long delay, long period)
void cancel()
```

- ***java.util.TimerTask***
  - Implements *Runnable* interface
  - Represents the task
  - Should be short-lived for repeated executions

```
long scheduledExecutionTime()
boolean cancel()
```

The *"java.util.Timer"* Class

Few things worth stressing upon about the *Timer* class …

- Single task-execution-thread per *Timer* object
- Timer tasks should complete quickly
- Always call *Timer.cancel()* when application is shutting down; otherwise it may cause memory leak
- Don't schedule any more tasks on the *Timer* after *cancelling* it
- *Timer* class is thread-safe
- *Timer* class does not offer any real-time guarantees

One-Time Execution in Future - Threads-API

Use ***Timer*** and ***TimerTask*** classes …

- Two ***Timer*** methods for scheduling one-time execution:
  - ***void schedule(TimerTask task, Date time)***
  - ***void schedule(TimerTask task, long delay)***

Repeated *Fixed-Delay* Executions – Threads API

Timer and ***TimerTask*** only, are used …

- Two methods in ***Timer*** class:
  - ***void schedule(TimerTask task, long delay, long period)***
  - ***void schedule(TimerTask task, Date firstTime, long period)***
- Next execution is scheduled when the current one begins
- Each execution is schedule relative to the actual start time of the previous one
- Start-times drift forward over long runs
  - Hence, frequency of executions becomes lower than that specified
- Appropriate for activities that require ‘smoothness’ over short periods of time e.g. blinking cursor, hourly chimes, etc.

Repeated *Fixed-Rate* Executions – Threads API

Timer and ***TimerTask*** only, are used …

- Two methods in ***Timer*** class:
  - ***void scheduleAtFixedRate(TimerTask task, long delay, long period)***
  - ***void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)***
- Next execution is scheduled when the current one begins
- Each execution is schedule relative to the scheduled start time of the ***first*** execution
- Start-times **DO NOT** drift forward over long runs
  - Hence, frequency of executions remains constant
- Appropriate for activities that are sensitive to 'absolute time' and accurate frequency of executions

Scheduling Tasks - Executors-API

Special thread-pool(s) provided by Executors for scheduling tasks …

- Available pools:
  - Single-thread-scheduled-executor
  - Scheduled-thread-pool
- Created using ***Executors (C)***
- ***ScheduledExecutorService (I)***
  - Extends ***ExecutorService (I)***
- ***ScheduledFuture (I)***. Extends:
  - ***Future (I)***
  - ***Delayed (I)***

```
static ScheduledExecutorService newSingleThreadScheduledExecutor()
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

```
ScheduledFuture<?> schedule(Runnable command, long delay,
                TimeUnit unit)
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay,
                TimeUnit unit)
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                long initialDelay, long delay, TimeUnit unit)
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                long initialDelay, long period, TimeUnit unit)
```

```
long getDelay(TimeUnit unit)
```

One Time Execution in Future – Executors API

ScheduledExecutorService is used …

- Two methods:
  - ***ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)***
  - ***ScheduledFuture<?> schedule(Callable<V> callable, long delay, TimeUnit unit)***
- Executions can be scheduled using a single-thread-executor or scheduled-thread-pool
- No need to extend *TimerTask*. Normal *Runnables* and *Callables* only are needed!

Repeated *Fixed-Delay* Executions – Executors API

ScheduledExecutorService is used …

- Only one method:
  - ***ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)***
- Executions can be scheduled using a single-thread-executor or scheduled-thread-pool
- Each execution is schedule relative to the termination-time of the ***previous*** execution
  - Start-times drift forward over long runs
  - Hence, frequency of executions becomes lower than that specified
- Only *Runnables* can be scheduled – no *Callables*!

Repeated *Fixed-Rate* Executions – Executors API

ScheduledExecutorService is used …

- Only one method:
  - ***ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)***
- Executions can be scheduled using a single-thread-executor or scheduled-thread-pool
- Each execution is schedule relative to the scheduled start time of the ***first*** execution
  - Start-times **DO NOT** drift forward over long runs
  - Hence, frequency of executions remains constant
- Only *Runnables* can be scheduled – no *Callables*!

THANK YOU !!!

This deck is part of the full-fledged video course on :

EFFICIENT JAVA MULTITHREADING WITH EXECUTORS

Visit: Course