**Java Executor Framework Tutorial and Best Practices**

March 12, 2013 by Lokesh Gupta

[Executors framework](http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html) (java.util.concurrent.Executor), released with the JDK 5 in package java.util.concurrent is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads.

We all know about that there are two ways to create a thread in java. If you want to read more about their comparison, [**read this post**](http://howtodoinjava.com/core-java/multi-threading/difference-between-implements-runnable-and-extends-thread-in-java/). Creating a thread in java is a very expensive process which includes memory overhead also. So, it’s a good idea if we can re-use these threads once created, to run our future runnables.

In this post, I will write some demo programs demonstrating the usage of Executor and then we will talk about some best practices which you need to keep in mind while designing your next multi-threaded application.

If you want read more about other**multi-threading** topics, [**follow this link**](http://howtodoinjava.com/category/core-java/multi-threading/).

**Basic usage demo application**

In our demo application, we have two tasks running. Neither is expected to terminate, and both should run for the duration of the application’s life. I will try to write a main wrapper class such that:

If any task throws an exception, the application will catch it and restart the task.  
If any task runs to completion, the application will notice and restart the task.

Below is the code sample of above required application:

|  |
| --- |
| package com.howtodoinjava.multiThreading.executors;    import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.Future;    public class DemoExecutorUsage {        private static ExecutorService executor = null;      private static volatile Future taskOneResults = null;      private static volatile Future taskTwoResults = null;        public static void main(String[] args) {          executor = Executors.newFixedThreadPool(2);          while (true)          {              try              {                  checkTasks();                  Thread.sleep(1000);              } catch (Exception e) {                  System.err.println("Caught exception: " + e.getMessage());              }          }      }        private static void checkTasks() throws Exception {          if (taskOneResults == null                  || taskOneResults.isDone()                  || taskOneResults.isCancelled())          {              taskOneResults = executor.submit(new TestOne());          }            if (taskTwoResults == null                  || taskTwoResults.isDone()                  || taskTwoResults.isCancelled())          {              taskTwoResults = executor.submit(new TestTwo());          }      }  }    class TestOne implements Runnable {      public void run() {          while (true)          {              System.out.println("Executing task one");              try              {                  Thread.sleep(1000);              } catch (Throwable e) {                  e.printStackTrace();              }          }        }  }    class TestTwo implements Runnable {      public void run() {          while (true)          {              System.out.println("Executing task two");              try              {                  Thread.sleep(1000);              } catch (Throwable e) {                  e.printStackTrace();              }          }      }  } |

Please do not forget to read best practices at the end of post.

**Executing multiple tasks in a single thread**

It’s not necessary that each Runnable should be executed in a separate thread. Sometimes, we need to do multiple jobs in a single thread and each job is instance of Runnable. To design this type of solution, a multi runnable should be used. This multi runnable is nothing but a collection of runnables which needs to be executed. Only addition is that this multi runnable is also a Runnable itself.

Below is the list of tasks which needs to be executed in a single thread.

|  |
| --- |
| package com.howtodoinjava.multiThreading.executors;    public class TaskOne implements Runnable {      @Override      public void run() {          System.out.println("Executing Task One");          try {              Thread.sleep(2000);          } catch (InterruptedException e) {              e.printStackTrace();          }      }  }    public class TaskTwo implements Runnable {      @Override      public void run() {          System.out.println("Executing Task Two");          try {              Thread.sleep(2000);          } catch (InterruptedException e) {              e.printStackTrace();          }      }  }    public class TaskThree implements Runnable {      @Override      public void run() {          System.out.println("Executing Task Three");          try {              Thread.sleep(2000);          } catch (InterruptedException e) {              e.printStackTrace();          }      }  } |

Lets create out multi runnable wrapper of above Tasks.

|  |
| --- |
| package com.howtodoinjava.demo.multiThread;    import java.util.List;    public class MultiRunnable implements Runnable {        private final List<Runnable> runnables;        public MultiRunnable(List<Runnable> runnables) {          this.runnables = runnables;      }        @Override      public void run() {          for (Runnable runnable : runnables) {               new Thread(runnable).start();          }      }  } |

Now above multi runnable can be executed this way as in below program:

|  |
| --- |
| package com.howtodoinjava.demo.multiThread;    import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.BlockingQueue;  import java.util.concurrent.RejectedExecutionHandler;  import java.util.concurrent.ThreadPoolExecutor;  import java.util.concurrent.TimeUnit;    public class MultiTaskExecutor {        public static void main(String[] args) {            BlockingQueue<Runnable> worksQueue = new ArrayBlockingQueue<Runnable>(10);          RejectedExecutionHandler rejectionHandler = new RejectedExecutionHandelerImpl();          ThreadPoolExecutor executor = new ThreadPoolExecutor(3, 3, 10, TimeUnit.SECONDS, worksQueue, rejectionHandler);            executor.prestartAllCoreThreads();            List<Runnable> taskGroup = new ArrayList<Runnable>();          taskGroup.add(new TestOne());          taskGroup.add(new TestTwo());          taskGroup.add(new TestThree());            worksQueue.add(new MultiRunnable(taskGroup));      }  }    class RejectedExecutionHandelerImpl implements RejectedExecutionHandler  {      @Override      public void rejectedExecution(Runnable runnable,              ThreadPoolExecutor executor)      {          System.out.println(runnable.toString() + " : I've been rejected ! ");      }  } |

**Best practices which must be followed**

1. Always run your java code against static analysis tools like [PMD](http://pmd.sourceforge.net/) and [FindBugs](http://findbugs.sourceforge.net/" \o "findbugs home page) to look for deeper issues. They are very helpful in determining ugly situations which may arise in future.
2. Always cross check and better plan a code review with senior guys to detect and possible [deadlock or livelock](https://en.wikipedia.org/wiki/Deadlock) in code during execution. Adding a health monitor in your application to check the status of running tasks is an excellent choice in most of the scenarios.
3. In multi-threaded programs, make a habit of catching errors too, not just exceptions. Sometimes unexpected things happen and Java throws an error at you, apart from an exception.
4. Use a back-off switch, so if something goes wrong and is non-recoverable, you don’t escalate the situation by eagerly starting another loop. Instead, you need to wait until the situation goes back to normal and then start again.
5. Please note that the whole point of executors is to abstract away the specifics of execution, so ordering is not guaranteed unless explicitly stated.