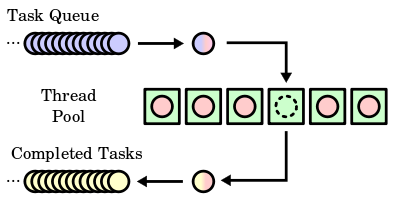
**Java Thread Pool – ThreadPoolExecutor Example**

Why you need **thread pool in Java**? Answer is usually when you develop a simple, concurrent application in Java, you create some Runnable objects and then create the corresponding Thread objects to execute them. [Creating a thread in Java](https://howtodoinjava.com/java/multi-threading/difference-between-implements-runnable-and-extends-thread-in-java/) is an expensive operation. And if you start creating new thread instance everytime to execute a task, application performance will degrade surely.

**1. How thread pool works in java**

A **thread pool is a collection of pre-initialized threads**. Generally the size of collection is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using same threads. If thread are more tasks than threads, then tasks need to wait in a queue like structure ([FIFO – First in first out](https://en.wikipedia.org/wiki/FIFO_and_LIFO_accounting#FIFO)).

When any thread completes it’s execution, it can pickup a new task from queue and execute it. When all tasks are completed the threads remain active and wait for more tasks in thread pool.

Thread Pool

A watcher keep watching queue (usually [BlockingQueue](https://howtodoinjava.com/java/multi-threading/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/" \t "_blank)) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them.

**2. ThreadPoolExecutor**

Since Java 5, the Java concurrency API provides a mechanism **Executor framework**. This is around the Executor interface, its sub-interface [ExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html" \t "_blank), and the ThreadPoolExecutor class that implements both interfaces.

ThreadPoolExecutor separates the task creation and its execution. With ThreadPoolExecutor, you only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads.

It goes beyond that and improves performance using a pool of threads. When you send a task to the executor, it tries to use a pooled thread for the execution of this task, to avoid continuous spawning of threads.

**3. How to create ThreadPoolExecutor**

We can create following 5 types of thread pool executors with pre-built methods in java.util.concurrent.Executors interface.

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is best fit for most off the real-life usecases.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10); |

1. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. DO NOT use this thread pool if tasks are long running. It can bring down the system if number of threads goes beyond what system can handle.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool(); |

1. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10); |

1. **Single thread pool executor** – Creates single thread to execute all tasks. Ute it when you have only one task to execute.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor(); |

1. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at single point of time, in multi-processor machines.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4); |

**4. ThreadPoolExecutor Example**

**4.1. Create Task**

Let’s create a task which will take random time to complete it, everytime.

|  |
| --- |
| Task.java |
| package com.howtodoinjava.threads;    import java.util.concurrent.TimeUnit;    public class Task implements Runnable {      private String name;        public Task(String name) {          this.name = name;      }        public String getName() {          return name;      }        public void run() {          try {              Long duration = (long) (Math.random() \* 10);              System.out.println("Executing : " + name);              TimeUnit.SECONDS.sleep(duration);          } catch (InterruptedException e) {              e.printStackTrace();          }      }  } |

**4.2. Execute tasks with thread pool executor**

Given program creates 5 tasks and submit to executor queue. Executor uses two threads to execute all tasks.

|  |
| --- |
| ThreadPoolExample.java |
| package com.howtodoinjava.threads;    import java.util.concurrent.Executors;  import java.util.concurrent.ThreadPoolExecutor;    public class ThreadPoolExample  {      public static void main(String[] args)      {          ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(2);            for (int i = 1; i <= 5; i++)          {              Task task = new Task("Task " + i);              System.out.println("Created : " + task.getName());                executor.execute(task);          }          executor.shutdown();      }  } |

Program output:

|  |
| --- |
| Console |
| Created : Task 1  Created : Task 2  Created : Task 3  Created : Task 4  Created : Task 5  Executing : Task 1  Executing : Task 2  Executing : Task 3  Executing : Task 4  Executing : Task 5 |

**5. ScheduledThreadPoolExecutor**

Fixed thread pools or cached thread pools are good when you have to execute one unique task only once. When you need to execute a task, repeatedly N times, either N fixed number of times or infinitively after fixed delay, you should be using **[ScheduledThreadPoolExecutor](https://howtodoinjava.com/java/multi-threading/task-scheduling-with-executors-scheduledthreadpoolexecutor-example/" \t "_blank)**.

ScheduledThreadPoolExecutor provides 4 methods which provide different capabilities to execute the tasks in repeated manner.

1. ScheduledFuture schedule(Runnable command, long delay, TimeUnit unit) – Creates and executes a task that becomes enabled after the given delay.
2. ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit) – Creates and executes a ScheduledFuture that becomes enabled after the given delay.
3. ScheduledFuture scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit) – Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay period. If any execution of this task takes longer than its period, then **subsequent executions may start late, but will not concurrently execute**.
4. ScheduledFuture scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) – Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay period. No matter how much time a long running task takes, there will be a fixed delay time gap between two executions.

**5.1. ScheduledThreadPoolExecutor Example**

|  |
| --- |
| ScheduledThreadPoolExecutorExample.java |
| package com.howtodoinjava.threads;    import java.util.concurrent.Executors;  import java.util.concurrent.ScheduledThreadPoolExecutor;  import java.util.concurrent.TimeUnit;    public class ScheduledThreadPoolExecutorExample  {      public static void main(String[] args)      {          ScheduledThreadPoolExecutor executor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(2);            Task task = new Task("Repeat Task");          System.out.println("Created : " + task.getName());            executor.scheduleWithFixedDelay(task, 2, 2, TimeUnit.SECONDS);      }  }    class Task implements Runnable {      private String name;        public Task(String name) {          this.name = name;      }        public String getName() {          return name;      }        public void run() {          System.out.println("Executing : " + name + ", Current Seconds : " + new Date().getSeconds());      }  } |

Program output:

|  |
| --- |
| Console |
| Created : Repeat Task  Executing : Repeat Task, Current Seconds : 36  Executing : Repeat Task, Current Seconds : 38  Executing : Repeat Task, Current Seconds : 41  Executing : Repeat Task, Current Seconds : 43  Executing : Repeat Task, Current Seconds : 45  Executing : Repeat Task, Current Seconds : 47 |

**6. Custom thread pool implementation in java**

Though Java has very robust thread pool functionality through [Executor framework](https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html). And you should not be creating your own **custom thread pool without executor**. I will strongly discourage any such attempt. Yet if you would like to create it for your own learning, the given below is such **thread pool implementation in Java**.

|  |
| --- |
| CustomThreadPool.java |
| package com.howtodoinjava.threads;    import java.util.concurrent.LinkedBlockingQueue;    @SuppressWarnings("unused")  public class CustomThreadPool  {      //Thread pool size      private final int poolSize;        //Internally pool is an array      private final WorkerThread[] workers;        // FIFO ordering      private final LinkedBlockingQueue<Runnable> queue;        public CustomThreadPool(int poolSize)      {          this.poolSize = poolSize;          queue = new LinkedBlockingQueue<Runnable>();          workers = new WorkerThread[poolSize];            for (int i = 0; i < poolSize; i++) {              workers[i] = new WorkerThread();              workers[i].start();          }      }        public void execute(Runnable task) {          synchronized (queue) {              queue.add(task);              queue.notify();          }      }        private class WorkerThread extends Thread {          public void run() {              Runnable task;                while (true) {                  synchronized (queue) {                      while (queue.isEmpty()) {                          try {                              queue.wait();                          } catch (InterruptedException e) {                              System.out.println("An error occurred while queue is waiting: " + e.getMessage());                          }                      }                      task = (Runnable) queue.poll();                  }                    try {                      task.run();                  } catch (RuntimeException e) {                      System.out.println("Thread pool is interrupted due to an issue: " + e.getMessage());                  }              }          }      }        public void shutdown() {          System.out.println("Shutting down thread pool");          for (int i = 0; i < poolSize; i++) {              workers[i] = null;          }      }  } |

Execute same task which we executed with ThreadPoolExecutor.

|  |
| --- |
| CustomThreadPoolExample.java |
| package com.howtodoinjava.threads;    public class CustomThreadPoolExample  {      public static void main(String[] args)      {          CustomThreadPool customThreadPool = new CustomThreadPool(2);            for (int i = 1; i <= 5; i++)          {              Task task = new Task("Task " + i);              System.out.println("Created : " + task.getName());                customThreadPool.execute(task);          }      }  } |

Program output:

|  |
| --- |
| Console |
| Created : Task 1  Created : Task 2  Created : Task 3  Created : Task 4  Created : Task 5  Executing : Task 1  Executing : Task 2  Executing : Task 3  Executing : Task 4  Executing : Task 5 |

Above is very raw thread pool implementation with scope of lots of improvements. But still, rather than perfecting above code, focus on learning **Java executor framework**.

Also note that incorrect pooling or queue handling can result in **deadlocks** or **resource thrashing** as well. You can certainly avoid these problems with Executor framework which is well tested by Java community.

**7. Summary**

1. The ThreadPoolExecutor class has four different constructors but, due to their complexity, the Java concurrency API provides the Executors class to construct executors and other related objects. Although we can create ThreadPoolExecutor directly using one of its constructors, it’s recommended to use the Executors class.
2. The cached thread pool, we have created above, creates new threads if needed to execute the new tasks, and reuses the existing ones if they have finished the execution of the task they were running, which are now available. The cached thread pool has, however, a disadvantage of constant lying threads for new tasks, so if you send too many tasks to this executor, you can overload the system. This can be overcome using fixed thread pool, which we will learn in next tutorial.
3. One critical aspect of the ThreadPoolExecutor class, and of the executors in general, is that you have to end it explicitly. If you don’t do this, the executor will continue its execution and the program won’t end. If the executor doesn’t have tasks to execute, it continues waiting for new tasks and it doesn’t end its execution. A Java application won’t end until all its non-daemon threads finish their execution, so, if you don’t terminate the executor, your application will never end.
4. To indicate to the executor that you want to finish it, you can use the shutdown() method of the ThreadPoolExecutor class. When the executor finishes the execution of all pending tasks, it finishes its execution. After you call the shutdown() method, if you try to send another task to the executor, it will be rejected and the executor will throw a RejectedExecutionException exception.
5. The ThreadPoolExecutor class provides a lot of methods to obtain information about its status. We used in the example the getPoolSize(), getActiveCount(), and getCompletedTaskCount() methods to obtain information about the size of the pool, the number of threads, and the number of completed tasks of the executor. You can also use the getLargestPoolSize() method that returns the maximum number of threads that has been in the pool at a time.
6. The ThreadPoolExecutor class also provides other methods related with the finalization of the executor. These methods are:
   * **shutdownNow()**: This method shut downs the executor immediately. It doesn’t execute the pending tasks. It returns a list with all these pending tasks. The tasks that are running when you call this method continue with their execution, but the method doesn’t wait for their finalization.
   * **isTerminated()**: This method returns true if you have called the shutdown() or shutdownNow() methods and the executor finishes the process of shutting it down.
   * **isShutdown()**: This method returns true if you have called the shutdown() method of the executor.
   * **awaitTermination(long timeout,TimeUnitunit)**: This method blocks the calling thread until the tasks of the executor have ended or the timeout occurs. The TimeUnit class is an enumeration with the following constants: DAYS, HOURS, MICROSECONDS etc.