# **Executor Framework- Understanding the basics (Part 1)**

Prior to Java 5, achieving concurrency was a difficult task for developers. But thanks to the introduction of the very powerful Executor framework, developers can do concurrent programming with much ease. Let's see the advantages of this framework.

* The most important feature of this framework is the separation of concerns. **It lets the developer to create tasks(Runnables, Callables), and let the framework decide when, how and where to execute that task** on a Thread which is totally configurable.
* It **relieves the developer from thread management**.
* It provides the developers **various types of queues for storing the tasks**. It also provides **various mechanisms for handling the scenario in which a task is rejected by the queue** when it is full.

**Let me explain this framework with a very relatable scenario.**

***Ever been to a railway station to get a ticket? There are various counters for issuing tickets. If the counter is free, you get the ticket. But if the counter is not free, you stand in a queue.***

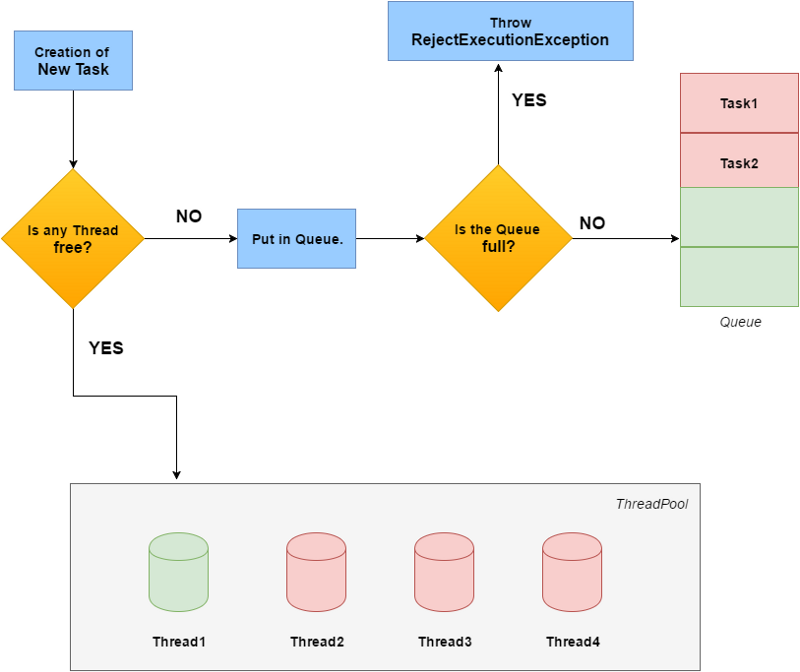
Now let me translate the above scenario in terms of this framework.

* For you and for all those people who want to get a ticket, getting a ticket is a **task** (*Runnable, Callable).*
* The counters who are issuing the tickets are working to issue the tickets. They are **Worker Threads and if we see them as one entity, they are a ThreadPool.**
* The people who are waiting to get a ticket are standing in a **queue.**

#### **How does Executor work?**

To simply put, the work of an executor is to execute tasks. The executor picks up a thread from the threadpool to execute a task. If a thread is not available and new threads cannot be created, then the executor stores these tasks in a queue. A task can also be removed from the queue. If the queue is full, then the queue will start rejecting the tasks. (Rejection of tasks can be handled).

The below diagram summarizes the work of the Executor framework.



Note: Red color means busy/full and Green color species empty/idle

After the task is completed, the framework will not terminate the executing thread immediately. The executor keeps a minimum number of threads in the thread pool even if all of them are not executing some task. But it will terminate the extra threads (number of threads which are greater than the minimum number of threads) after the specified duration.

#### **Using ThreadPoolExecutor**

Executor is the base interface of this framework which has only one method **execute (Runnable command).**

**ExecutorService** and **ScheduledExecutorService** are two other interfaces which extends Executor.

These two interfaces have a lot of important methods like

* **submit (Runnable task),**
* ***shutdown (),***
* ***schedule (Callable<V> callable, long delay, TimeUnit unit) etc.***

*which actually make this framework really useful.*

*The most commonly used implementations of these interfaces are*

* *ThreadPoolExecutor and*
* *ScheduledThreadPoolExecutor.*
* ForkJoinPool.

While creating an instance of ThreadPoolExecutor, we can give various configurations.

* **We specify the minimum and maximum number of threads that should be present in the ThreadPool**.
* We also specify the duration after which the extra threads (current threads — minimum threads) should be terminated.
* There are various types of queues which can be used for storing the tasks which cannot be executed immediately. **Based on the application requirements we specify the type of queue to be used**.

Below is the code snippet for creating an instance of ThreadPoolExecutor.

|  |
| --- |
| **public** ThreadPoolExecutor (intcorePoolSize, int maximumPoolSize, **long** keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,  RejectedExecutionHandler handler) |

Now let's understand the parameters of ThreadPoolExecutor. For optimal performance of the application, each parameter to the executor needs to be passed very carefully.

* **corePoolSize** : The minimum number of threads to keep in the pool.
* **maximumPoolSize :** The maximum number of threads to keep in the pool.
* **keepAliveTime *:*** If current number of threads are greater than the minimum threads, then wait for this time to terminate the extra threads.
* **Timeunit*:***The time unit for the previous argument.
* **workQueue *:*** The queue used for holding the tasks.
* **handler:** An instance of RejectionExecutionHandler, which handles the task which is rejected by the executor.

**Once the executor is created, it is ready to accept tasks for execution. We can execute tasks through these two methods:**

1. ***execute (Runnable command):*** *This method accepts a Runnable object but returns void.*
2. **submit (Runnable task)/*submit (Callable<T> task):*** *This method can accept a Runnable or Callable object and returns Future object.*

**For demonstrating the workings of ThreadPoolExecutor, I will be using a method will takes integer as an argument and returns a random alpha-numeric string of that length. The method internally invokes Thread.sleep(1000). So, there will be a delay of 1 second for every response.**

|  |
| --- |
| public class RandomClass{    private static final String characters = "abcdefghijklmnopqrstuvwxyz0123456789";    public static String getRandomString(int length) {  if (length < 0 || length > 10) {  return "Random";  }  StringBuilder stringBuilder = new StringBuilder ();  Random random = new Random ();  for (int i = 0; i < length; i++) {  int index = random.nextInt(36);  stringBuilder.append(characters.charAt(index));  }  Utils.putToSleep(1000);  return stringBuilder.toString();  }  } |

Let’s start with a very simple example.

* In the below code, an executor is created with 2 minimum threads and 2 maximum threads.
* The queue used is LinkedBlockingQueue.
* We call *printString()* function which creates 6 Runnables in a loop and passes them on for execution.
* Each Runnable calls *getRandomString(int length)* function of the above class with the current index of the loop as ‘*length*’ variable. So first Runnable will get String of length 1, second Runnable will get String of length 2 and so on.

|  |
| --- |
| public class StringPrinter {    private final ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 2, 0,  TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable> ());    public static void main (String [] args) {  new StringPrinter().printString();  }    public void printString() {  for (int i = 1; i <= 6; i++) {  threadPoolExecutor.execute(getRunnable(i));  }  }    private Runnable getRunnable(final int i) {  Runnable runnable = new Runnable () {  @Override  public void run () {  String randomString = RandomClass.getRandomString(i);  System.out.println("String returned is: + randomString);  }  };  return runnable;  }  } |

**Although there are 6 runnables, the executor has only 2 threads to execute these runnables. So only 2 runnables will be executed at a time.**

Once their execution is completed, the threads will pick up next 2 Runnables for execution and so on until all the tasks are completed.

The output of the above program is

— — — — — — — — — — — — **Delay of 1 second**— — — — — — — ——  
String returned is: *g*  
*String returned is:* *io*  
*— — — — — — — — — — — —* **Delay of 1 second**— — — — — — — — — —  
String returned is: *k9k*  
*String returned is:* *o6zh*  
*— — — — — — — — — — — —* **Delay of 1 second**— — — — — — — — — —   
String returned is: *mujt5*  
*String returned is:* *xn4gar*

***Output will vary because I am returning a Random string.***

By changing the thread configuration, we can change the output, If the executor is created with 3 minimum and 3 maximum threads, then 3 Runnables will be executed in parallel, And the output will be somewhat like this.

— — — — — — — — — — — — **Delay of 1 second**— — — — — — — — — —

*String returned is* : *g*  
*String returned is* : *io*  
 *String returned is* : *k9k*  
*— — — — — — — — — — — —* **Delay of 1 second**— — — — — — — ——  
String returned is: *o6zh*  
*String returned is:* *mujt5*  
*String returned is:* *xn4gar*

#### **Using ScheduledThreadPoolExecutor**

This executor can be used when we want to schedule the tasks. There are times when we want to the repeat the execution of a same task at certain time intervals- e.g. Executing a task after every 5 seconds. ScheduledThreadPoolExecutor is the perfect candidate for these types of tasks. There are various useful methods for scheduling tasks in this executor as explained below.

* ***schedule (Runnable command, long delay, TimeUnit unit):*** This will execute the task after the delay has expired. So, let’s say that you have to execute a task after 10 seconds use *schedule (runnable, 10, TimeUnit.SECONDS****).***
* ***scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)* :** For the first time,the runnable will be executed after the initialDelay has expired. After that, every time that Runnable will be run after the given *period* (3rd argument)has expired.

Let’s say that you want to run a task after every 10 seconds. Using ***schedule (runnable, 0, 10, TimeUnit.SECONDS)*** , the Runnable will execute immediately for the first time, and from there on it will be executed after every 10 seconds. The time interval between executing tasks does not depend upon the time taken by the task itself to complete.

* ***scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) :*** Although this function may seem similar to the above function, there is a difference between them. Unlike the previous function, this function waits for the previously executing task to complete and only after delay period (3rd argument) has expired, it will execute the task next time.

#### **Shutting down an Executor**

An executor can be shut down using ***shutDown*()** function. When the executor is shut down, **it will no longer accept any new task** and submitting any task to it will throw a RejectedExecutionException. **But the tasks already executing on the threads and stored in the queue for execution will be executed.**

But if we do not want to execute the tasks stored in queue, then we need to call ***shutDownNow()*.** Calling this method will make sure that any task stored in the queue should be not executed.

**What about the tasks which are already executing on the thread?**

Looks like calling any of the above two methods won’t cancel those tasks. But the executor sends an interrupt flag to all the currently executing threads. The runnable should check whether the thread is interrupted or not. And if the thread is interrupted, the runnable should decide what to do.

**public static** String getRandomString(**int** length) {

**if**(Thread.*currentThread*().isInterrupted()){  
 // Either ignore this block or do some action  
 }  
}

# **Executor Framework- Understanding the basics (Part 2)**

In my previous post, I had discussed about the basics of Executor framework. In this post, I am going to discuss about **BlockingQueues**.

**BlockingQueue:-**

BlockingQueue is an integral part of an Executor. BlockingQueue is used for storing the tasks when all the available threads are busy executing tasks and a new thread cannot be created.

A developer should consider the following things while choosing a particular BlockingQueue.

* **Size of the queue:** What should be size of the queue? Should the queue have a fixed size, or should the queue be unbounded?
* **Order of execution of the task:** What should be the order of execution of the tasks stored in the queue? Should a task which was submitted first be executed first or should the tasks be picked up based on their priority?

**The executor framework provides different kinds of BlockingQueues.**

Let's see the different types of queues available in the framework and how do they differ based on the above parameters.

#### **LinkedBlockingQueue**

**The size of the queue is dynamic**- i.e. it will be created with an initial size and then as the tasks keeps on adding to the queue, its size will be increased.

Theoretically it can add up-to Integer.MAX\_VALUE elements. Optionally the user can give a capacity to the queue using **LinkedBlockingQueue(int capacity)** constructor so that the queue doesn’t expand to more elements than are required.

**The queue works in FIFO approach**. So, the tasks, which are submitted first are executed first.

#### **ArrayBlockingQueue**

As the name suggests, the queue internally uses **an array of fixed size to hold the tasks**.

While creating an instance of ArrayBlockingQueue, we need to specify the size of the queue.

**The queue works in FIFO fashion-** i.e. a task which is submitted first will be executed first**.** But if the queue is full and a new task is submitted, then the queue will reject that task.

|  |
| --- |
| Now in this code, let’s replace LinkedBlockingQueue with ArrayBlockingQueue which has a size of 2 elements.  ***private final ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 2, 0,***  ***TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable> (2));***  **What happens when we run the above code?**  When the 1st task is picked up, a new thread will be created for its execution.  Similarly, when the 2nd task is picked up, a new thread will be created.  Now since both the threads are busy executing tasks, any new incoming tasks will be stored in the queue. Therefore 3rd and 4th tasks will be stored in the queue.  But after storing 2 tasks, the queue becomes full. When the 5th task is submitted, the queue is already full and cannot accept any incoming task.  The executor will throw **RejectedExecutionException**.  Similarly, 6th task will be rejected. So only the first 4 tasks will be executed and the remaining 2 will be rejected. |

#### **PriorityBlockingQueue**

This is a special type of queue in which the executor picks up a task from the queue based on its priority and not its order.

While creating a task, we need to assign a priority to every task.

|  |
| --- |
| public enum Priority {    HIGHEST (3),  HIGH (2),  MEDIUM (1),  LOW (0);    int priority;    Priority (int priority) {  this.priority = priority;  }    public int getPriority() {  return priority;  }  } |

The executor uses Comparable interface to find the task with higher priority.

In the code below, we create a PriorityRunnable which extends Runnable and Comparable interfaces. Through *setPriority()* method we set the priority for the runnable.

|  |
| --- |
| public class PriorityRunnable implements Runnable, Comparable<PriorityRunnable> {    private Priority priority;  private int length;    public PriorityRunnable(int length) {  super ();  this.length = length;  }    public void setPriority(Priority priority) {  this.priority = priority;  }    public Priority getPriority() {  return priority;  }    @Override  public void run () {  String randomString = RandomClass.getRandomString(length);  System.out.println("String returned is " + randomString);  }    @Override  public int compareTo(PriorityRunnable other) {  return other.priority.getPriority() - priority.getPriority();  } |

***If the Runnable submitted to the executor does not implement Comparable, then it will throw the following exception.***  
***PriorityRunnable cannot be cast to java.lang.Comparable***

**Now let’s modify our code of StringPrinter.java.**

* In this code, we are passing a **‘*length*’** variable to the ***getRunnable(int length)***method.
* Now we create a new method ***getPriorityRunnable(int length).*** This method will create a PriorityRunnable and assign a HIGH priority to it if the length passed is even and LOW priority if the length passed is odd.

|  |
| --- |
| private Runnable getPriorityRunnable(final int length) {  //Default priority is HIGH  Priority priority = Priority.HIGH;  if (length % 2 == 1) {  //If odd length, set priority to LOW.  priority = Priority.LOW;  }  PriorityRunnable priorityRunnable = new PriorityRunnable(length);  priorityRunnable.setPriority(priority);  return priorityRunnable;  } |

**What happens when we run the code?**  
The Runnables which were created with even length will be executed first and then the odd length Runnables will be executed. But if two Runnables have the same priority, then the executor can pick any one of them. The order of execution of the Runnables with same priority is not certain. The executor can pick up the tasks with same priority in any order.

***But be cautious in assigning the priority to the tasks. The queue can result in starvation of certain tasks as there is a chance that the tasks with low priority will never be picked*.**

#### **LinkedBlockingDeque**

Similar to LinkedBlockingQueue, this queue is also dynamic in size and optionally unbounded. But unlike LinkedBlockingQueue, where the tasks are always appended to the end of the queue, here we can append a task at the beginning of the queue also. **Basically, we can access the queue from both the ends. We can insert a task either to the starting or the end of the queue. The executor can pick up a task for execution either from the starting or the end of the queue.**

//Get the underlying queue from the executor  
LinkedBlockingDeque queue = (LinkedBlockingDeque) executor.getQueue();//Insert the task to the beginning of the queue queue.addFirst(runnable);

#### **DelayQueue**

This is also a special type of queue, in which every task is created with a delay period. The executor will not execute a task before its delay has been expired. The executor uses **Delayed interface** to compare the two tasks for their delay period.

In the below code, we create a DelayedRunnable which extends Runnable and Delayed interfaces. Through *setDelay(int delay)* method we set the delay for the runnable.

|  |
| --- |
| public class DelayedRunnable implements Runnable, Delayed {    private int length;  private int delay;    public DelayedRunnable(int length) {  super ();  this.length = length;  }    public void setDelay(int delay) {  this.delay = delay;  }    public int getDelay() {  return delay;  }    @Override  public void run () {  String randomString = RandomClass.getRandomString(length);  System.out.println("String returned is " + randomString);  }    @Override  public long getDelay(TimeUnit unit) {  return unit.convert(getDelay(), TimeUnit.MILLISECONDS);  }    @Override  public int compareTo(Delayed another) {  int delay1 = getDelay();  int delay2 = ((DelayedRunnable) another).getDelay();  return delay1 - delay2;  }  } |

***If the Runnable submitted to the executor does not implement Delayed, then it will throw the following exception.***  
***DelayedRunnable cannot be cast to java.util.concurrent.Delayed***

**Now let’s modify our code of StringPrinter.java.**

* Till now we were passing a **‘*length*’** variable to the ***getRunnable(int length)* method.**
* Now we create a new method ***getDelayedRunnable(int length).*** This method will create a DelayedRunnable and assign a delay of 3000 milliseconds if the length is odd and 6000 milliseconds if the length is even.

|  |
| --- |
| private Runnable getDelayedRunnable(final int length) {  //Default delay is 3000 milliseconds  int delay = 3000;  if (length % 2 == 0) {  //If even length, set delay to 6000 milliseconds  delay = 6000;  }  DelayedRunnable delayedRunnable = new DelayedRunnable(length);  delayedRunnable.setDelay(delay);  return delayedRunnable;  } |

**What happens when we run the code?**  
The Runnables which were created with odd length will be executed first because they have shorter delay period i.e. 3000 milliseconds. After that the Runnables which were created by passing by even length will be executed. But if two Runnables have the same delay period, then the executor can pick any one of them. The order of execution of the Runnables with same delay period is not certain.

***Similar to PriorityBlockingQueue, this queue can result in starvation of certain tasks as there is a chance that the tasks with higher delay period will never be picked*.**

#### **SynchronousQueue**

This is a special kind of queue and even though it is a queue, it does not hold any elements (tasks). **The function of this queue is to receive the tasks and pass them for execution but not hold them**. If the queue receives a task, and if a worker thread is present, it will pass on that task for execution to that thread, otherwise it will reject the task. Optionally it can wait for some time till the thread becomes available for executing the task before rejecting the task.

#### **What happens when the queue is full?**

Till now, the task of the executor seemed simple enough — **Execute a task on the thread if the thread is available or put the task on the queue for later execution**. But what happens when a task is submitted to the queue and the queue is full? Simple. **The queue will reject the task**. But this can be handled. While adding a task to the queue, you can specify to wait for a certain time if the queue is full. Only after the specified duration has been passed and the queue is still not empty, the task will be rejected.

You can also pass a **RejectedExecutionHandler** while creating an executor which gets back the rejected tasks in its **rejectedExecution(Runnable r, ThreadPoolExecutor executor)** method.

From here, you can decide what to do with the rejected tasks.

The below code snippet tries to execute the rejected task after 3 seconds.

|  |
| --- |
| public class CustomRejectionExecutionHandler implements RejectedExecutionHandler {  @Override  public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  try {  //Get the underlying queue used in the executor  BlockingQueue queue = executor.getQueue();  //Add the rejected runnable to the queue after 3000 milliseconds.  queue.offer(r, 3000, TimeUnit.MILLISECONDS);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

# **A Deep Dive into the Java ExecutorService**

The Java ExecutorService is a construct that allows you to pass a task to be executed by a thread asynchronously. The executor service creates and maintains a reusable pool of threads for executing submitted tasks. The service also manages a queue, which is used when there are more tasks than the number of threads in the pool and there is a need to queue up tasks until there is a free thread available to execute the task.

**In this article, we'll focus on the ThreadPoolExecutor implementation of the ExecutorService interface.**

There are two ways to instantiate a Thread Pool Executor.

* You can either directly instantiate it using one of its constructor overloads or
* you can use one of the factory methods in the Executors class.

**Let's look at a few examples.**

* **Directly instantiating a ThreadPoolExecutor with 10 threads, a keepAliveTime of 0 milliseconds, and a LinkedBlockingQueue:-**

|  |
| --- |
| ExecutorService executorService =  new ThreadPoolExecutor(10, 10, 0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable> ()); |

* **Instantiating a ThreadPoolExecutor with 10 threads using an Executors factory method: -**

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

* **Instantiating a ThreadPoolExecutor with a single thread using an Executors factory method: -**

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

* **Instantiating a ThreadPoolExecutor that adds threads to the pool as needed using an Executors factory method: -**

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

When you instantiate your Executor Service, a few parameters are initialized. Depending on how you instantiated your Executor Service, you may manually specify these parameters, or they may be provided for you by default.

**These parameters are:-**

* **corePool size**
* **maxPool size**
* **workQueue**
* **keepAliveTime**
* **threadFactory**
* **rejectedExecutionHandler**

Using one of the factory methods available in the Executors class simply selects some default values for the above for you based on your inputs.

**Executors.newSingleThreadExecutor()** creates a pool with a core size of 1, max size of 1, a keepAliveTime of 0ms (which means that the thread in the pool would stay alive unless explicitly closed), an unbounded LinkedBlockingQueue, the default threadFactory, and the default rejectedExecutionHandler.

Meanwhile, **Executors.newFixedThreadPool(10)** creates a pool with a core size of 10, max size of 10, a keepAliveTime of 0ms, an unbounded LinkedBlockingQueue, the default threadFactory, and the default rejectedExecutionHandler.

**So, what do all these parameters mean?**

**The core pool size** is the minimum number of threads that should be kept in the pool. The number of threads may grow to reach the max pool size (if it is higher than the core pool size), but in general, it represents the number of threads you expect to have alive in the pool. When a task is submitted to the executor, it checks if the actual running number of threads is less than the core pool size. If it is, then it creates a new worker using the specified threadFactory.

**The max pool size** is the maximum number of workers that can be in the pool. If the max pool size is greater than the core pool size, it means that the pool can grow in size, i.e. more workers can be added to the pool. Workers are added to the pool when a task is submitted but the work queue is full. Every time this happens, a new worker is added until the max pool size is reached. If the max pool size has already been reached and the work queue is full, then the next task will be rejected.

**The work queue** is used to queue up tasks for the available worker threads. The queue can be bounded or unbounded. For bounded queues, setting the queue size is an important exercise, as it affects how the worker pool grows and when you start running into **RejectedExecutionExceptions**. If you have a work pool that you expect to grow; say from a core pool size of 20 workers to a max of 100 workers, then you may not want to set the queue size to a number that is too high, like 10,000, because it means that 10,000 tasks must be enqueued before each additional worker gets added to the pool. Unbounded queues and bounded queues with very high capacities are more suited to be used with fixed size pools (i.e. pools where the core and max pools sizes are the same).

**If a thread pool grows to the max size, how does it shrink back to the core size?**

**That's where the keepAliveTime comes in.** If the current number of worker threads exceeds the core pool size and a keepAliveTime is set, then worker threads are shut down when there is no more work to do until the number of worker threads is back to the core pool size; a thread will wait for work for the **keepAlive time**, and when that is exceeded and no work arrives, it will shut down.

* **Side note 1:** You can set **allowCoreThreadTimeOut** to true on your ThreadPoolExecutor instance, and if you do so, then not only do workers threads that exceed the core pool size get shut down on idle, but core worker threads also get shut down on idle. By default, this is set to false.
* **Side note 2**: If your worker threads acquire and maintain expensive resources and only release those resources on shutdown, then it becomes important to optimally configure your keepAlive time. A keepAlive time of 0 ms means that your workers never shut down after they are created, unless the executor service itself is shut down.

Most times, using the Default ThreadFactory is sufficient. The default thread factory creates worker threads that have a normal priority and are not daemon threads. It also gives the threads a name with the format: **"pool-{poolNumber}-thread-{threadNumber}"**. If you want to customize any of these attributes, such as the thread name or priority, then you should provide your own threadFactory implementation. Another benefit of providing your own threadFactory implementation is that you can set the thread's uncaught exception handler, which can be very useful in combating silent failures.

**Speaking of exceptions,** I mentioned that for pools with a bounded work queue, task rejections occur when the queue is full, and no more workers can be added. **You can configure a handler to run when such a rejection occurs.** These handlers are called "**Policies**".

By default, the **AbortPolicy** is used, which throws a **RejectedExecutionException**. You can choose to use another Policy such as the **DiscardPolicy**, which simply discards the task silently; the **CallerRunsPolicy**, which executes the task on the calling thread instead of one of the worker threads; or any another policy implementation you create.

To wrap up, the Java Executor Service hides a lot of complexity but also makes it easy for you to dive in and tweak the inner workings if you so choose. The Executors class provides a lot of factory methods that address different use cases; `**newFixedThreadPool**()` for when you just need a fixed number of threads that execute tasks, `**newCachedThreadPool**()` for when you want to create new threads as needed and shrink the pool when not needed, etc. In many cases, these pre-defined pools will meet your needs. However, if you have more defined parameters, then it helps to know some of the knobs you can tweak and how that affects the thread pool's behavior.