Java 5 added a new Java package to the Java platform, the java.util.concurrent package. This package contains a set of classes that makes it easier to develop concurrent (multithreaded) applications in Java. Before this package was added, you would have to program your utility classes yourself.

In this tutorial I will take you through the new java.util.concurrent classes, one by one, so you can learn how to use them. I will use the versions in Java 6. I am not sure if there are any differences to the Java 5 versions.

I will not explain the core issues of concurrency in Java - the theory behind it, that is. If you are interested in that, check out my [**Java Concurrency tutorial**](https://jenkov.com/java-concurrency/index.html).

# Guide to ExecutorService in Java

Learn to use Java *ExecutorService* to execute a [Runnable](https://howtodoinjava.com/java/multi-threading/java-runnable-vs-thread/) or [Callable](https://howtodoinjava.com/java/multi-threading/java-callable-future-example/) class in an asynchronous way. Also learn the various best practices to utilize it in the most efficient manner in any Java application.

## **1. What is Executor Framework?**

In simple Java applications, we do not face many challenges while working with a small number of threads. If we have to develop a program that runs a lot of concurrent tasks, this approach will present many disadvantages such as lots of boilerplate code (create and manage threads), executing threads manually and keeping track of thread execution results.

[Executor framework](https://howtodoinjava.com/java/multi-threading/executor-service-example/) (since Java 1.5) solved this problem. The framework consists of three main interfaces (and lots of child interfaces):

* *Executor*,
* *ExecutorService*
* [*ThreadPoolExecutor*](https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/)

### **1.1. Benefits of Executor Framework**

* The framework mainly separates task creation and execution. Task creation is mainly boilerplate code and is easily replaceable.
* With an executor, we have to create tasks that implement either Runnable or Callable interface and send them to the executor.
* Executor internally maintains a (configurable) thread pool to improve application performance by avoiding the continuous spawning of threads.
* Executor is responsible for executing the tasks, and running them with the necessary threads from the pool.

### **1.2. Callable and Future**

Another important advantage of the Executor framework is the use of the Callable interface. It’s similar to the Runnable interface with two benefits:

1. It’s call() method returns a result after the thread execution is complete.
2. When we send a Callable object to an executor, we get a Future object’s reference. We can use this object to query the status of the thread and the result of the Callable object.

## **2. Creating ExecutorService Instance**

ExecutorService is an interface and its implementations can execute a Runnable or Callable class in an asynchronous way. Note that invoking the run() method of a Runnable interface in a synchronous way is simply calling a method.

We can create an instance of ExecutorService in following ways:

### **2.1. Using *Executors***

Executors is a utility class that provides factory methods for creating the implementations of the interface.

*//Executes only one thread*

ExecutorService es = Executors.newSingleThreadExecutor();

*//Internally manages thread pool of 2 threads*

ExecutorService es = Executors.newFixedThreadPool(2);

*//Internally manages thread pool of 10 threads to run scheduled tasks*

ExecutorService es = Executors.newScheduledThreadPool(10);

### **2.2. Using Constructors**

We can choose an implementation class of ExecutorService interface and create its instance directly. The below statement creates a thread pool executor with a minimum thread count 10, maximum threads count 100 and 5 milliseconds keep alive time and a blocking queue to watch for tasks in future.

ExecutorService executorService = new ThreadPoolExecutor(10, 100, 5L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

## **3. Submitting Tasks to ExecutorService**

Generally, tasks are created by implementing either Runnable or Callable interface. Let’s see the example of both cases.

### **3.1. Executing *Runnable* Tasks**

We can execute runnables using the following methods :

* void execute(Runnable task) – executes the given command at some time in the future.
* Future submit(Runnable task) – submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return null upon successful completion.
* Future submit(Runnable task, T result) – Submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return the given result upon successful completion.

# Java Thread Pools and ThreadPoolExecutor

[Lokesh Gupta](https://howtodoinjava.com/author/lokeshgupta1981/)August 10, 2022

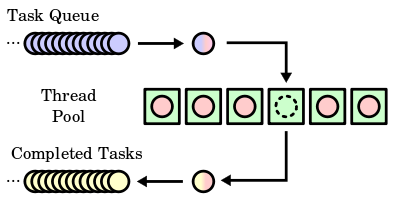
[Java Concurrency](https://howtodoinjava.com/java/multi-threading/)[Concurrency](https://howtodoinjava.com/tag/concurrency/), [Java Executor](https://howtodoinjava.com/tag/java-executor/), [Thread Pool](https://howtodoinjava.com/tag/thread-pool/)

Why do we need a thread pool in Java? The answer is when we develop a simple, concurrent application in Java, we create some Runnable objects and then create the corresponding Thread objects to execute them. Creating a thread in Java is an expensive operation. And if you start creating a new thread instance everytime to accomplish a task, application performance will degrade.

## **1. How does a Thread Pool Works?**

A thread pool is a collection of pre-initialized threads. Generally, the collection size is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using the same threads. If there 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 its execution, it can pickup a new task from the queue and execute it. When all tasks are completed, the threads remain active and wait for more tasks in the thread pool.

Thread Pool

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

## **2. *ThreadPoolExecutor* class**

Since Java 5, the Java concurrency API provides a mechanism [Executor framework](https://howtodoinjava.com/java/multi-threading/executor-service-example/). The main pieces are Executor interface, its sub-interface ExecutorService and the ThreadPoolExecutor class that implements both interfaces.

ThreadPoolExecutor separates the task creation and its execution. With ThreadPoolExecutor, we only have to implement the Runnable objects and send them to the executor. It is responsible for executing, instantiating, and running the tasks 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 to execute this task, to avoid the continuous spawning of threads.

## **3. Creating *ThreadPoolExecutor***

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

### **3.1. Fixed Sized 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 the best fit for most of real-life use-cases.

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

### **3.2. 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 the number of threads exceeds what the system can handle.

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

### **3.3. 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);

### **3.4. Single Thread Pool Executor**

Creates a single thread to execute all tasks. Use it when you have only one task to execute.

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

### **3.5. Work Stealing Thread Pool Executor**

Creates a thread pool that maintains enough threads to support the given parallelism level. Here, [parallelism](https://howtodoinjava.com/java/multi-threading/concurrency-vs-parallelism/) level means the maximum number of threads that will be used to execute a given task at a single point in multi-processor machines.

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