### ****Executor Framework****

The **Executor Framework** was introduced in Java 5 as part of the java.util.concurrent package to provide a higher-level replacement for managing and controlling thread execution in Java applications. The Executor Framework decouples the task submission from the mechanics of how each task will be executed, such as creating new threads, scheduling tasks, etc. This framework provides a simpler, more flexible mechanism for managing thread pools, ensuring efficient and reusable execution.

### ****Key Components of the Executor Framework****

1. **Executor**: The basic interface for task execution.
2. **ExecutorService**: Extends Executor and provides more advanced methods to manage the lifecycle of tasks and handle tasks with result values (through Callable).
3. **Executors**: A utility class that provides factory methods for creating different types of Executor and ExecutorService instances.

### ****1. Executor Interface****

The Executor interface is the simplest interface in the Executor framework. It provides a single method:

* **void execute(Runnable command)**: Executes the given Runnable task. The Runnable interface represents a task that can be executed concurrently by a thread.

#### ****Example of Executor Interface:****

import java.util.concurrent.Executor;

public class ExecutorExample {

public static void main(String[] args) {

Executor executor = new Executor() {

@Override

public void execute(Runnable command) {

new Thread(command).start(); // Create a new thread to execute the command

}

};

Runnable task = () -> System.out.println("Executing task in thread: " + Thread.currentThread().getName());

executor.execute(task); // Submitting the task to the executor

}

}

In this example:

* We create an anonymous class that implements the Executor interface.
* The execute() method is called to submit a Runnable task to be executed by a new thread.

#### Output:

Executing task in thread: Thread-0

### ****2. ExecutorService Interface****

The ExecutorService interface extends Executor and adds more powerful features for managing and controlling task execution:

* **submit(Callable<T> task)**: Submits a task for execution that returns a result.
* **submit(Runnable task)**: Submits a Runnable task for execution that does not return any result.
* **shutdown()**: Initiates an orderly shutdown of the executor, where previously submitted tasks are executed, but no new tasks will be accepted.
* **invokeAll(Collection<? extends Callable<T>> tasks)**: Executes a collection of Callable tasks and returns a list of Future objects.
* **invokeAny(Collection<? extends Callable<T>> tasks)**: Executes a collection of Callable tasks and returns the result of one that completes first.

#### ****Example of ExecutorService:****

import java.util.concurrent.\*;

public class ExecutorServiceExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

// Create an ExecutorService

ExecutorService executorService = Executors.newFixedThreadPool(2);

// Submit tasks

executorService.submit(() -> System.out.println("Task 1 executed by " + Thread.currentThread().getName()));

executorService.submit(() -> System.out.println("Task 2 executed by " + Thread.currentThread().getName()));

// Submit a Callable task that returns a result

Future<Integer> result = executorService.submit(() -> {

Thread.sleep(1000);

return 42;

});

// Get the result of the Callable task

System.out.println("Callable result: " + result.get());

// Shutdown the executor

executorService.shutdown();

}

}

#### Output:

Task 1 executed by pool-1-thread-1

Task 2 executed by pool-1-thread-2

Callable result: 42

In this example:

* We create an ExecutorService using Executors.newFixedThreadPool(2), which creates a thread pool with 2 threads.
* We submit Runnable tasks for execution.
* We submit a Callable task and retrieve its result using Future.

#### Key Methods of ExecutorService:

* **submit()**: Submits a task for execution and returns a Future that can be used to retrieve the result of the task.
* **shutdown()**: Initiates the shutdown of the executor and prevents new tasks from being accepted.
* **invokeAll()**: Executes a list of tasks and waits for all of them to finish.
* **invokeAny()**: Executes a list of tasks and returns the result of the first task that completes.

### ****3. Executors Utility Class****

The Executors class is a utility class that provides factory methods for creating different types of ExecutorService instances, such as:

* **newFixedThreadPool(int nThreads)**: Creates a thread pool with a fixed number of threads.
* **newCachedThreadPool()**: Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when available.
* **newSingleThreadExecutor()**: Creates an executor that uses a single worker thread.
* **newScheduledThreadPool(int corePoolSize)**: Creates a thread pool that can schedule tasks with fixed-rate or fixed-delay execution policies.

#### ****Example Using Executors Class:****

import java.util.concurrent.\*;

public class ExecutorsExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

// Create a fixed thread pool using Executors utility class

ExecutorService executorService = Executors.newFixedThreadPool(3);

// Submit tasks for execution

executorService.submit(() -> System.out.println("Task 1 executed by " + Thread.currentThread().getName()));

executorService.submit(() -> System.out.println("Task 2 executed by " + Thread.currentThread().getName()));

executorService.submit(() -> System.out.println("Task 3 executed by " + Thread.currentThread().getName()));

// Shutdown the executor

executorService.shutdown();

}

}

#### Output:

Task 1 executed by pool-1-thread-1

Task 2 executed by pool-1-thread-2

Task 3 executed by pool-1-thread-3

In this example, we use Executors.newFixedThreadPool(3) to create an executor with a fixed pool of 3 threads, and submit 3 tasks for execution.

### ****Key Factory Methods in**** Executors:

* **newFixedThreadPool(int nThreads)**: A thread pool with a fixed number of threads.
* **newCachedThreadPool()**: A thread pool that creates new threads as needed but will reuse previously constructed threads when available.
* **newSingleThreadExecutor()**: A single-threaded executor.
* **newScheduledThreadPool(int corePoolSize)**: A thread pool for scheduled tasks.
* **newWorkStealingPool()**: A pool where worker threads can steal tasks from other worker threads.

### ****Key Differences Between**** Executor****,**** ExecutorService****, and**** Executors

| **Feature** | **Executor** | **ExecutorService** | **Executors** |
| --- | --- | --- | --- |
| **Interface Type** | Basic interface for task execution. | Extends Executor and provides more advanced features. | Utility class providing factory methods. |
| **Methods** | Only execute(Runnable task) | Methods for lifecycle management (shutdown(), submit(), invokeAll(), etc.) | Factory methods like newFixedThreadPool(), newCachedThreadPool(), etc. |
| **Usage** | Simple task execution. | Used for more complex task submission and result management. | Used to create ExecutorService instances. |
| **Shutdown Management** | No direct shutdown management. | Provides shutdown() and shutdownNow() for managing lifecycle. | N/A (Factory class, not for direct task submission) |

### ****When to Use Each?****

* **Use Executor**: When you only need basic task submission without lifecycle management, and you want simple execution control.
* **Use ExecutorService**: When you need to submit tasks that may return results (Callable tasks), manage the shutdown of the executor, or manage the lifecycle of tasks.
* **Use Executors**: When you need to easily create different types of thread pools or executors for different tasks.