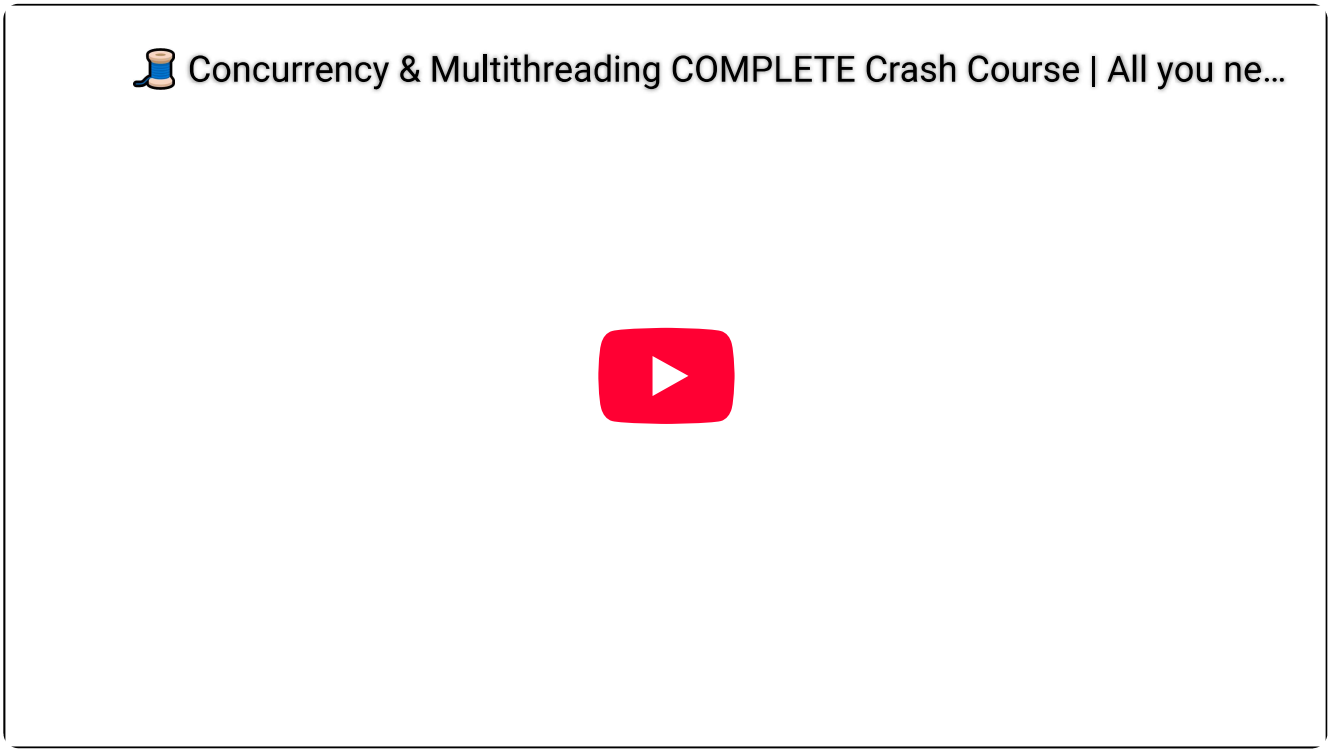
Thread Executors



Topic Tags:

System Design LLD

🐆⬛ **Github Codes Link**: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Thread Executors in Java: The Power of Structured Concurrency 🚀⚙️

Thread Executors are a high-level concurrency framework in Java that provide a powerful abstraction over thread management. They simplify the complex task of creating, scheduling, and controlling threads, allowing developers to focus on business logic rather than thread lifecycle management. This article explores Thread Executors and their most critical methods for effective concurrent programming.

Why Thread Executors? 🤔

While raw threads and thread pools offer control over concurrent operations, Thread Executors provide structured, task-based concurrency with several key advantages:

• Separation of task submission from execution details 📋

Java

```java
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> {
    System.out.println("Task executed by: " + Thread.currentThread().getName
});
```

Explanation:
You just submit a task without worrying about creating or starting a thread—Executor handles it behind the scenes.

Built-in thread pooling and resource management 👩‍🏭

Java

```java
ExecutorService pool = Executors.newFixedThreadPool(5);
for (int i = 0; i < 10; i++) {
    pool.execute(() -> {
        System.out.println("Running: " + Thread.currentThread().getName());
    });
}
```

Explanation:
The executor reuses the same 5 threads to run 10 tasks, preventing overhead from creating a new thread for each task.

Task queuing, scheduling, and execution policies ⏱️

Java

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.schedule(() -> {
```

```
    System.out.println("Executed after 3 seconds!");
}, 3, TimeUnit.SECONDS);
```

Explanation:
You can delay task execution or schedule it periodically—useful for cron-like jobs or time-based tasks.

Lifecycle control and graceful shutdown capabilities 🛑

Java

```java
ExecutorService executor = Executors.newCachedThreadPool();
executor.submit(() -> System.out.println("Working..."));
executor.shutdown(); // Initiates an orderly shutdown
```

Explanation:
You can shut down the executor gracefully, allowing current tasks to complete without abruptly killing threads.

Monitoring and management facilities 📊

Java

```java
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadP

executor.submit(() -> {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ignored) {
  }
});

System.out.println("Active Threads: " + executor.getActiveCount());
System.out.println("Queued Tasks: " + executor.getQueue().size());
```

Explanation:

You get insights like active thread count and queue size to help with performance tuning and debugging.

Core Executor Interfaces and Classes 🧩

The Java concurrency framework provides several key interfaces and classes for working with Thread Executors:

1. Executor: The base interface defining task execution

Java

```java
Executor executor = command -> new Thread(command).start();
executor.execute(() -> System.out.println("Simple task executed"));
```

Explanation:
Executor is the simplest form—just defines a execute(Runnable) method. You provide a task, and it decides how to run it (in this case, a new thread).

2. ExecutorService: Extends Executor with lifecycle management :

Syntax :

Java

```java
ExecutorService executorService = Executors.newFixedThreadPool(int nThreads)
```

Example

Java

```java
ExecutorService executorService = Executors.newFixedThreadPool(2);
Future<String> future = executorService.submit(() -> "Hello ExecutorService"
System.out.println(future.get()); // Output: Hello ExecutorService
executorService.shutdown();
```

Explanation:
ExecutorService allows advanced task handling with submit(), invokeAll(), shutdown(), and Future results for return values and tracking task completion.

3. **ScheduledExecutorService: Adds task scheduling capabilities**

Syntax :

Java

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(int co
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
```

Example

Java

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.scheduleAtFixedRate(() -> {
    System.out.println("Running every 2 seconds");
}, 0, 2, TimeUnit.SECONDS);
```

Explanation:
Allows you to schedule tasks to run after a delay or at a fixed rate—ideal for repeated jobs like health checks or polling.

4. **ThreadPoolExecutor: Primary implementation of ExecutorService**

Syntax :

Java

```java
ThreadPoolExecutor customPool = new ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue
);
```

Example

Java

```
ThreadPoolExecutor customPool = new ThreadPoolExecutor(
    2, 4, 60, TimeUnit.SECONDS,
    new LinkedBlockingQueue<>()
);
customPool.execute(() -> System.out.println("Task in custom pool"));
```

Explanation:
Gives full control—core/max threads, queue type, keep-alive time, rejection policy. Great for performance-tuned, production-grade thread pool management.

5. **ScheduledThreadPoolExecutor: Implementation of ScheduledExecutorService**

Syntax :

Java

```
ScheduledThreadPoolExecutor scheduledPool = new ScheduledThreadPoolExecutor(
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
ScheduledThreadPoolExecutor scheduledPool = new ScheduledThreadPoolExecutor(
scheduledPool.schedule(() -> System.out.println("Scheduled once"), 5, TimeUn
```

Explanation:
A concrete subclass of ScheduledExecutorService, enabling delayed and repeated task scheduling with all thread pool tuning options.

6. **Executors: Factory class for creating executor instances**

Syntax :

Java

```
ExecutorService newFixedThreadPool(int nThreads);
ScheduledExecutorService newScheduledThreadPool(int corePoolSize);
```

Example

Java

```
ExecutorService fixedPool = Executors.newFixedThreadPool(3);
```

```
ScheduledExecutorService scheduled = Executors.newScheduledThreadPool(1);
```

Explanation:
Utility class with factory methods (newFixedThreadPool, newCachedThreadPool, newSingleThreadExecutor, etc.) to easily spin up ready-to-use executors without complex configuration.

Essential Methods of ExecutorService 🛠️

• **Task Submission Methods** 📥

○ void execute(Runnable command) :
Syntax :

Java

```
ExecutorService newFixedThreadPool(int nThreads);
```

Example

Java

```
// Submit a Runnable task with no result
      ExecutorService executor = Executors.newFixedThreadPool(1);
      executor.execute(() -> System.out.println("Task executed"));
```

✅ Success:
Task runs asynchronously. No result is expected or tracked.

⚠️ Failure Scenarios:
▪ If the task throws an exception, it's lost unless the thread is wrapped with error logging, where the log can be made to track the same.
▪ You won't know if it failed, retried, or completed — no result tracking.
▪ If the machine crashes or process exits, task is lost.

Future<?> submit :

Java

```
// Submit a Runnable or a callable task with a Future result
    Future<?> submit(Runnable task)
```

```
    Future<?> submit(Callable task)
    // Submit a Callable task with a Future result
    ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(() -> "Hello from Callable");
```

✅ Success:
▪ Returns a Future. You can block and get the result using future.get().

⚠️ Failure Scenarios:
▪ If the task fails (e.g., throws an exception), future.get() will throw ExecutionException.
▪ You can still check future.isCancelled() or future.isDone().
▪ If machine crashes or JVM exits — in-flight tasks are lost.

invokeAll(Collection<? extends Callable<T>> tasks):

Java

```
List<Future<String>> results = executor.invokeAll(tasks);
```

✅ Success:
▪ Runs all tasks in parallel. Waits until all finish. You get a list of Futures.

⚠️ Failure Scenarios:
▪ If one task fails, its Future will throw an exception on get(), but others keep running.
▪ If the executor shuts down in the middle, only the remaining tasks are interrupted.
▪ Machine crash = all tasks in memory are lost.

> *Note: We also have an overloaded method invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) that allows you to specify a timeout for the completion of all tasks.*

Syntax :

Java

```
ExecutorService newFixedThreadPool(int nThreads);
  public static void main(String[] args) {
          ExecutorService executor = Executors.newFixedThreadPool(2); // t
          // invokeAll Example
          Collection<Callable<String>> allTasks = Arrays.asList(() -> "Tas
          try {
```

```
7                // Process results
8                List<Future<String>> results = executor.invokeAll(allTasks);
9                // Process timeout results
10               List<Future<String>> timeoutResults = executor.invokeAll(all
11
12           } catch (InterruptedException e) {
13               Thread.currentThread().interrupt();
14           }
15   }
```

• Lifecycle Management Methods ⚙️

Java

```
1  // Initiate an orderly shutdown
2  void shutdown()
3  // Attempt to stop all actively executing tasks
4  List<Runnable> shutdownNow()
```

> *NOTE: For the above methods implementation refer to the Previous Article of Thread Pool and Thread Lifecycle*

• Scheduled Executor Service Methods ⏰

schedule(Runnable command, long delay, TimeUnit unit):

Java

```
1  ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
2  scheduler.schedule(() -> System.out.println("Run once later"), 2, TimeUnit.S
```

✅ Success:
▪ Schedules a task to run once after a delay.

⚠️ Failure Scenarios:
▪ If the task throws an exception, it won't run again.
▪ Lost completely if the system crashes before the delay elapses.

▪ No built-in retry logic.

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

Java

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
    // Schedule a task to run periodically at a fixed rate
    scheduler.scheduleAtFixedRate(() -> System.out.println("Repeats"), 1, 3,
```

✅ Success:
▪ Runs repeatedly with a fixed rate between task starts (not finishes).

⚠️ Failure Scenarios:
▪ If a task throws an exception, it stops future executions.
▪ If a task takes longer than the rate interval, it may overlap (depending on thread pool size).
▪ Crashes = all schedules lost unless you persist them externally.

Creating Executors with the Executors Factory 🏭

Java

```java
// Fixed thread pool with a specified number of threads
ExecutorService fixedPool = Executors.newFixedThreadPool(nThreads);

// Single-threaded executor
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

// Cached thread pool that creates new threads as needed
ExecutorService cachedPool = Executors.newCachedThreadPool();

// Scheduled executor with a specified pool size
ScheduledExecutorService scheduledPool = Executors.newScheduledThreadPool(co
```

• ThreadPoolExecutor Configuration Parameters ⚙️

When more control is needed, the Thread Pool Executor class can be directly instantiated with specific parameters:

Java

```java
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    corePoolSize,      // Minimum number of threads to keep alive
    maximumPoolSize,   // Maximum number of threads allowed
    keepAliveTime,     // Time to keep non-core threads alive when idle
    timeUnit,          // Unit for keepAliveTime
    workQueue,         // Queue for holding tasks before execution
    threadFactory,     // Factory for creating new threads
    rejectionHandler   // Handler for rejected tasks
);
```

Interview Questions 🎯

1. What's the difference between execute() and submit() methods?
Answer: execute() accepts only Runnable tasks and doesn't return any result. submit() accepts both Runnable and Callable tasks and returns a Future object that can be used to retrieve results or check completion status. 🔄

Java

```java
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(() -> System.out.println("Runnable executed"));
Future<Integer> future = executor.submit(() -> 42);
System.out.println("Callable result: " + future.get());
executor.shutdown();
```

Output :

```
Runnable executed
Callable result: 42
```

2. How does ThreadPoolExecutor decide whether to create a new thread or queue a task?

Answer: It follows this sequence:
• If fewer than corePoolSize threads are running, create a new thread.
• If corePoolSize or more threads are running, add the task to the queue.

• If the queue is full and fewer than maximumPoolSize threads are running, create a new thread.
• If the queue is full and maximumPoolSize threads are running, reject the task. 🔢

Syntax :

Java

```java
ThreadPoolExecutor customPool = new ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue
);
```

Example:

Java

```java
ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 4, 1, TimeUnit.SECON
```

3. What happens if you don't explicitly shut down an ExecutorService?
Answer: The executor's threads will continue running, preventing the JVM from shutting down normally (unless they're daemon threads). This can cause memory leaks and resource exhaustion. Always call shutdown() or shutdownNow() when done with an executor. 🛑

4. What is the difference between scheduleAtFixedRate and scheduleWithFixedDelay methods ?
Answer:
• scheduleAtFixedRate attempts to execute tasks at a consistent rate regardless of how long each task takes (tasks might overlap if execution takes longer than the period).
• scheduleWithFixedDelay waits for the specified delay time after each task completes before starting the next execution. ⏱️

Syntax :

Java

```java
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDela
```

Example:

Java

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.scheduleAtFixedRate(() -> System.out.println("Fixed Rate Task"), 0
scheduler.scheduleWithFixedDelay(() -> System.out.println("Fixed Delay Task"
```

5. How can you handle exceptions thrown by tasks submitted to an ExecutorService?
Answer: For submit() methods, exceptions are stored in the returned Future and thrown when calling get().

Java

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
  Future<Integer> future = executor.submit(() -> {
      throw new RuntimeException("Boom!");
  });
  try {
      future.get(); // Will throw ExecutionException wrapping the original
  } catch (ExecutionException e) {
      System.out.println("Caught: " + e.getCause()); // prints: Boom!
  } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
  }
```

🔍 **Why:**
submit() captures exceptions in the returned Future. Calling get() will throw an ExecutionException. use try-catch inside the task

Java

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.submit(() -> {
        try {
            throw new RuntimeException("Handled inside task");
        } catch (Exception e) {
            System.out.println("Caught in task: " + e.getMessage());
```

```
        }
    });
```

🔍 **Why:**
Always the safest fallback. Catch exceptions inside the task to log or recover locally.

Conclusion 🎯

Thread Executors represent a significant advancement in Java concurrency programming, abstracting away the complexities of direct thread manipulation while providing powerful tools for task management and execution control. By understanding the core interfaces, essential methods, and best practices outlined in this article, developers can create robust, efficient concurrent applications that fully leverage modern hardware capabilities while maintaining predictable behavior and resource usage.

As applications grow in complexity and scale, mastering Thread Executors becomes increasingly important for delivering responsive, resilient software systems. Whether handling asynchronous operations, scheduling recurring tasks, or processing parallel workloads, Thread Executors provide the structured approach needed for successful concurrent programming in Java. 🚀🧵