The **Executor Framework** in Java, introduced in **Java 5**, provides a high-level API for managing and controlling threads, handling their lifecycle, and efficiently running tasks asynchronously. At its core, it abstracts the complexities of thread management, such as creating, managing, and terminating threads, and offers various **executor services** to handle tasks concurrently.

**Key Components of the Executor Framework:**

1. **Executor Interface**: The base interface that defines a method execute(Runnable command) for executing tasks asynchronously.
2. **ExecutorService Interface**: Extends Executor and provides more advanced methods, such as submit(), shutdown(), invokeAll(), invokeAny(), and awaitTermination(), for task management and lifecycle control.
3. **ThreadPoolExecutor**: The core implementation of ExecutorService, which manages a pool of worker threads for running tasks. It handles task queuing, thread creation, and lifecycle events.

**Thread Lifecycle in the Executor Framework**

In the context of the **Executor Framework**, thread lifecycle management is abstracted, and the framework manages the creation, reuse, and termination of threads internally. Here’s how the thread lifecycle works:

**1. Thread Creation:**

* When a task is submitted to an executor (e.g., ExecutorService.submit() or execute()), the framework first checks if there are any idle threads in the pool.
* If there are no idle threads and the pool is not yet at its maximum size, the executor creates a new thread to handle the task.
* Threads in the pool are typically created by the ThreadFactory, a pluggable component that provides flexibility for customizing how threads are created (e.g., naming, priority).
* The thread is started and immediately begins executing the task.

**2. Thread Execution:**

* Once a thread is created or reused, it picks a task from the task queue and executes it.
* Tasks are typically submitted as Runnable or Callable instances.
* If the task is a Runnable, the run() method is executed.
* If the task is a Callable, the call() method is executed, and the result is returned (or a Future is provided).
* The thread keeps running, checking the task queue for more work after completing a task, which prevents constant thread creation and destruction (reducing overhead).

**3. Thread Reuse:**

* Once a thread finishes executing a task, it does not terminate immediately. Instead, it returns to the pool, where it waits for new tasks to be assigned.
* This reuse of threads improves performance because creating and destroying threads is costly in terms of resources.
* The idle threads stay alive based on the pool's configuration (e.g., core pool size, max pool size, and keep-alive time).

**4. Task Queuing:**

* The ExecutorService uses a queue to hold tasks that are waiting to be executed.
* The type of queue used (e.g., **BlockingQueue**, **LinkedBlockingQueue**, **SynchronousQueue**) can affect task scheduling:
  + **LinkedBlockingQueue**: Tasks are queued when all threads are busy, and the pool can handle large volumes of tasks.
  + **SynchronousQueue**: No queuing, tasks are directly handed off to threads.
* If all threads are busy and the task queue is full (depending on the queue capacity), the executor may reject new tasks (using a rejection policy, e.g., AbortPolicy, CallerRunsPolicy, etc.).

**5. Thread Termination:**

* **Idle Thread Termination**: If a thread has been idle (i.e., not executing any tasks) for longer than the configured keepAliveTime, and the number of threads exceeds the **core pool size**, the executor may terminate idle threads to free resources.
* **Shutdown**: When ExecutorService.shutdown() is called:
  + No new tasks are accepted.
  + Previously submitted tasks continue to run until completion.
  + Once all tasks are finished, the pool is shut down, and all threads are terminated.
* **Immediate Shutdown**: When ExecutorService.shutdownNow() is called:
  + The executor attempts to stop all currently executing tasks and halts any queued tasks.
  + It returns a list of tasks that were waiting to be executed.
  + Threads are interrupted if they are blocked, attempting to finish any current work and exit.
* **Awaiting Termination**: You can wait for the threads to complete gracefully by calling awaitTermination() after invoking shutdown(). This method blocks until all tasks are finished or the timeout expires.

**Thread Lifecycle Summary**

In a **ThreadPoolExecutor**, the thread lifecycle typically follows these stages:

1. **Created**: A new thread is created if there are no idle threads available and the pool is not full.
2. **Running**: The thread executes the task it has been assigned.
3. **Waiting/Idle**: Once the task is completed, the thread goes back to the pool, waiting for new tasks. Idle threads remain alive unless keepAliveTime expires or the pool is shut down.
4. **Terminated**: When the pool shuts down or idle threads exceed their keepAliveTime and the thread count exceeds the core pool size, the thread is terminated.

1 Request Type M Fixed Value: ‘G’

22 TRN M Unique Transaction id

20 Name M

1 Format Type M Fixed Value: &#39;U&#39;

19 Amount M 3 decimal places and decimal sign must be

supplied. Leading plus (+) sign is optional.

Leading minus (-) sign is mandatory for

negative numbers.

Sample 17.45

000000000000017.450

3 Currency M ISO Standards currency codes

For examples, EUR, GBP.

3 Service M Values:

ATZ

AUZ

ATC

2 Source Country

Code

M Values: For example e.g. DE, GB, AT

Main

Message

Text M See below for structure of Main message