ForkJoinPool in Java - Complete Explanation

What is ForkJoinPool?

ForkJoinPool is a special thread pool introduced in Java 7, designed to efficiently execute tasks that can be broken down into smaller subtasks and processed in parallel. It is based on the "divide and conquer" principle.

Key Responsibilities

- Executes **fork-join tasks** concurrently.
- Supports **work stealing** to keep all threads busy.
- Manages a pool of **worker threads**, each with its own task queue.

Important Note:

> ForkJoinPool **does not automatically divide** the given task into multiple subtasks. It is the **developer's responsibility** to break down the task using `.fork()` and combine results using `.join()`. The `.fork()` method does **not** split a task — it simply **submits a subtask** (already created) to the pool for execution.

Where is ForkJoinPool Used?

1. **Parallel Streams**
2. `list.parallelStream().forEach(...)`
3. Internally uses the common ForkJoinPool.
4. **CompletableFuture**
5. `CompletableFuture.supplyAsync(...)`
6. Uses the common ForkJoinPool by default.
7. **Direct Task Execution**
8. Developers can extend `RecursiveTask` or `RecursiveAction` and manually submit tasks to ForkJoinPool.

Relationship to Processor Cores

By default, the common ForkJoinPool creates:

```
Runtime.getRuntime().availableProcessors() - 1
```

This means it uses almost all available logical processors, leaving one for the main thread.

How ForkJoinPool Works Internally

1. A task is submitted to the pool.
2. The pool does **not automatically split the task**.
3. If you're using it directly, **you** must manually divide the task (fork).
4. Each thread has its own **deque (double-ended queue)**.
5. Threads execute tasks from the **head** of their own deque.
6. If a thread becomes idle, it **steals tasks** from the **tail** of another thread's deque.

Work-Stealing Algorithm

- Designed to keep all threads busy.
- When a thread finishes its own tasks, it looks for work in other threads' queues.
- It steals tasks from the **tail** of another thread's deque.
- This reduces contention and improves performance.

LIFO (Last-In-First-Out) for own deque: better cache locality.\ **FIFO (First-In-First-Out)** for stealing: prevents contention.

Manual Usage Example

```
class SumTask extends RecursiveTask<Integer> {
    int[] arr;
    int start, end;

    public SumTask(int[] arr, int start, int end) {
        this.arr = arr;
        this.start = start;
        this.end = end;
    }

    protected Integer compute() {
        if (end - start <= 10) {
            int sum = 0;
```

```
            for (int i = start; i < end; i++) sum += arr[i];
            return sum;
        } else {
            int mid = (start + end) / 2;
            SumTask left = new SumTask(arr, start, mid);
            SumTask right = new SumTask(arr, mid, end);

            left.fork();
            int rightResult = right.compute();
            int leftResult = left.join();
            return leftResult + rightResult;
        }
    }
}
```

In this example, the developer is responsible for splitting and managing tasks.

- `.fork()` **does not divide the task**; it simply submits the already-created subtask to the ForkJoinPool for asynchronous execution by one of the available worker threads.
- `.join()` waits for the result of the forked task and combines it with the current task's result.

Summary Table

Usage Type	Who splits the task?	Who manages execution?
Direct ForkJoinPool	You (via fork/join)	ForkJoinPool
Parallel Streams	Java (via Spliterator)	ForkJoinPool
CompletableFuture	You control logic	ForkJoinPool

Conclusion

ForkJoinPool is a powerful tool for parallel execution in Java. While it manages threads and task scheduling, **the responsibility of dividing tasks lies with the developer**, unless you use abstractions like parallel streams or CompletableFuture, which handle that internally.

The `.fork()` method does not split work but assigns the subtask (already created by the developer) to a worker thread in the pool. The `.join()` method is then used to collect the result of the subtask, effectively synchronizing them into the final outcome.

Work-stealing ensures maximum CPU utilization, making ForkJoinPool suitable for both compute-intensive tasks and high-performance applications.