# Fork Join Framework

Principle of Programming Languages

# Introduction

In Java, the fork/join framework provides support for parallel programming by splitting up a task into smaller tasks to process them using the available CPU cores.

# Working of Fork/Join

- Parallelism: simultaneous execution of two or more tasks.
- Concurrency: execution of two or more tasks in overlapping time periods that are not necessarily simultaneous.

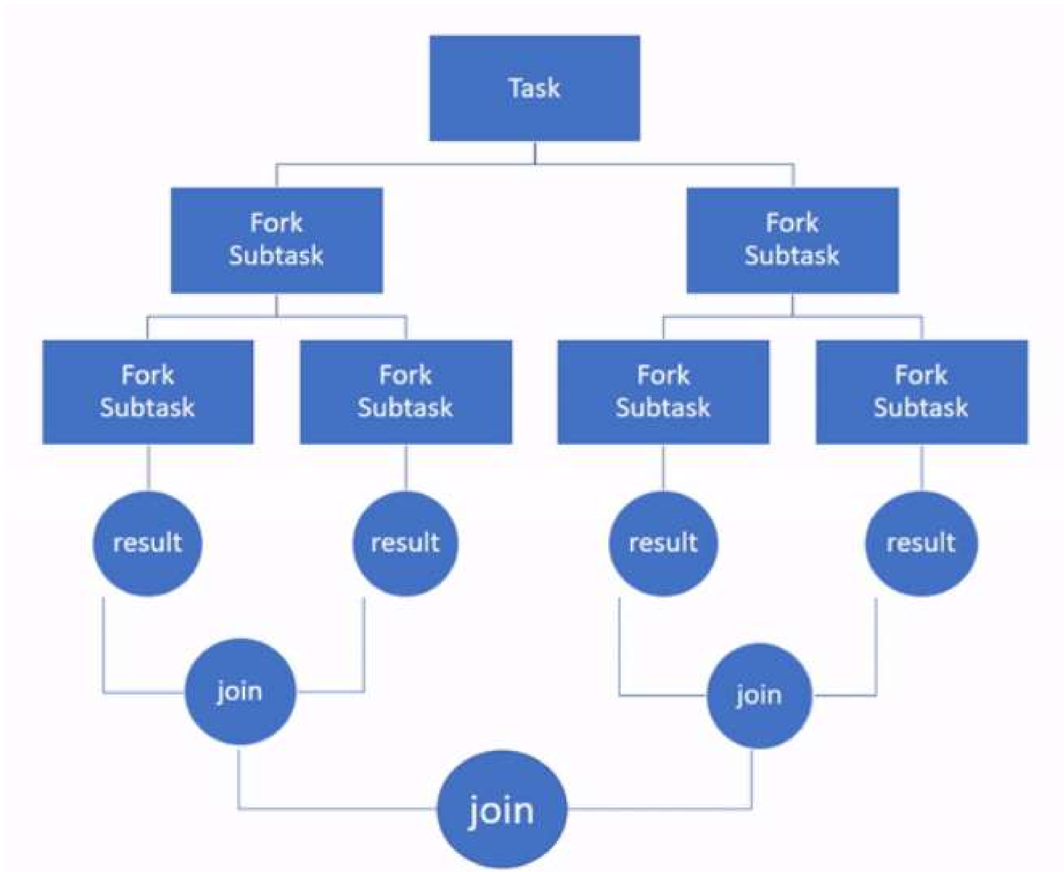
- The fork/join framework was designed to speed up the execution of tasks that can be divided into other smaller subtasks, executing them in parallel and then combining their results to get a single one.

# Working of fork/join

- Applying a divide and conquer principle, the framework recursively divides the task into smaller subtasks until a given threshold is reached. This is the fork part.

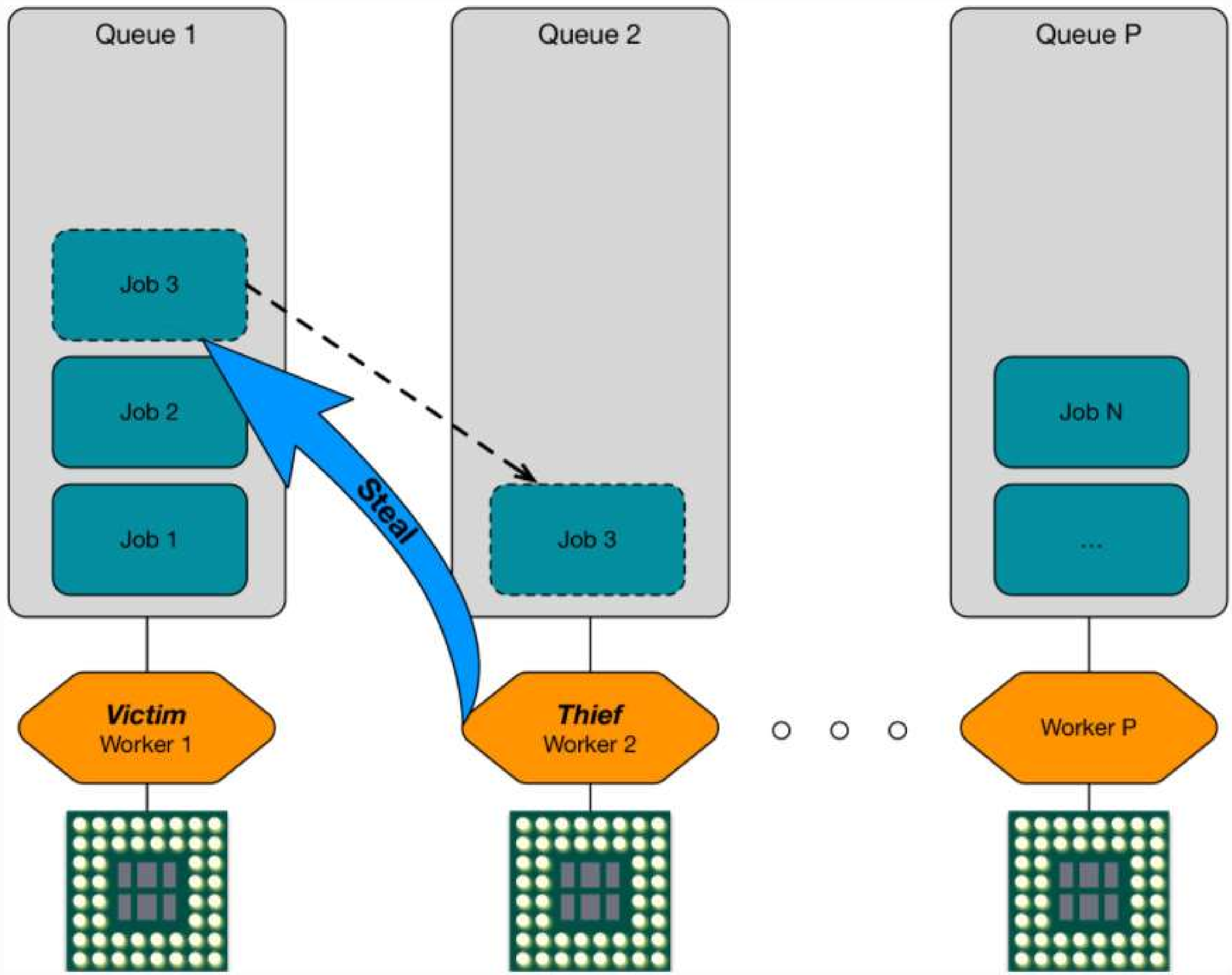
- Then, the subtasks are processed independently and if they return a result, all the results are recursively combined into a single result. This is the join part.

# Working of fork/join



# Working of fork/join

- To execute the tasks in parallel, the framework uses a pool of threads, with several threads equal to the number of processors available to the Java Virtual Machine (JVM) by default.
- Each thread has its double-ended queue (deque) to store the tasks that will execute.
- A deque is a queue that supports adding or removing elements from either the front (head) or the back (tail). This allows two things:
  - A thread can execute only one task at a time (the task at the head of its deque).
  - A work-stealing algorithm is implemented to balance the thread's workload.




*Stealing of work items*

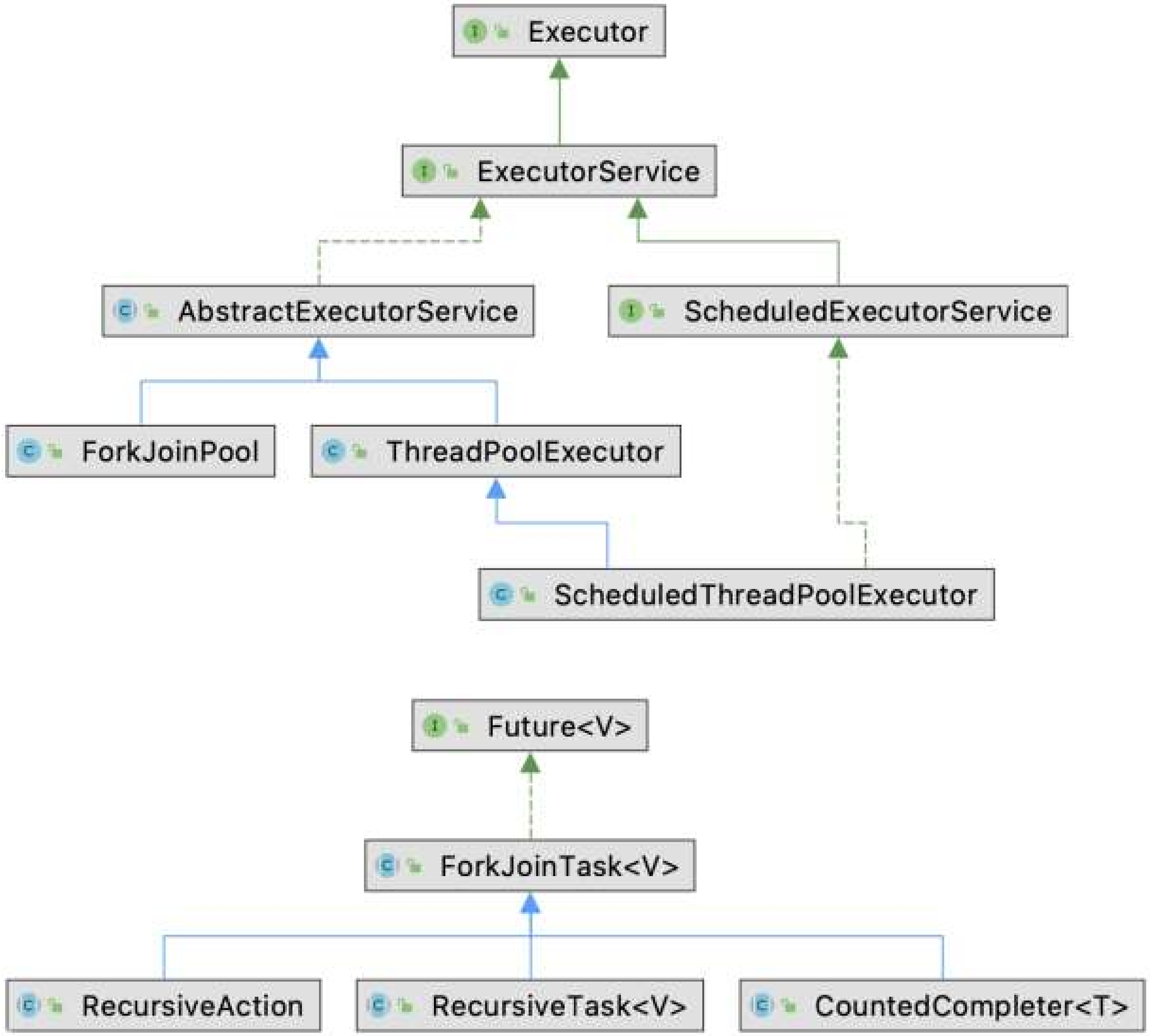
# Working of fork/join

- With the work-stealing algorithm, threads that run out of tasks to process can steal tasks from other threads that are still busy (by removing tasks from the tail of their deque).

- This approach makes processing more efficient by increasing throughput when there are many tasks to process or when one task diverges into many subtasks.

# Classes in the framework

- ForkJoinPool: An executor dedicated to running instances implementing ForkJoinTask<V>. Implements the Work Stealing Algorithm to balance the load among threads: if a worker thread runs out of things to do, it can steal tasks from other threads that are still busy.
- ForkJoinTask<V>: An abstract class that defines a task that runs within a ForkJoinPool.
- RecursiveAction: A ForkJoinTask subclass for tasks that don’t return values.
- RecursiveTask<V>: A ForkJoinTask subclass for tasks that return values.


Executor
ExecutorService
AbstractExecutorService
ScheduledExecutorService
ForkJoinPool
ThreadPoolExecutor
ScheduledThreadPoolExecutor
Future<V>
ForkJoinTask<V>
RecursiveAction
RecursiveTask<V>
CountedCompleter<T>

# Understanding the framework classes

- The fork/join framework has two main classes,
  - ForkJoinPool and ForkJoinTask.
- ForkJoinPool implements the interface ExecutorService, which uses the Work Stealing Algorithm.
- There's a common ForkJoinPool instance available to all applications that you can get with the static method commonPool()

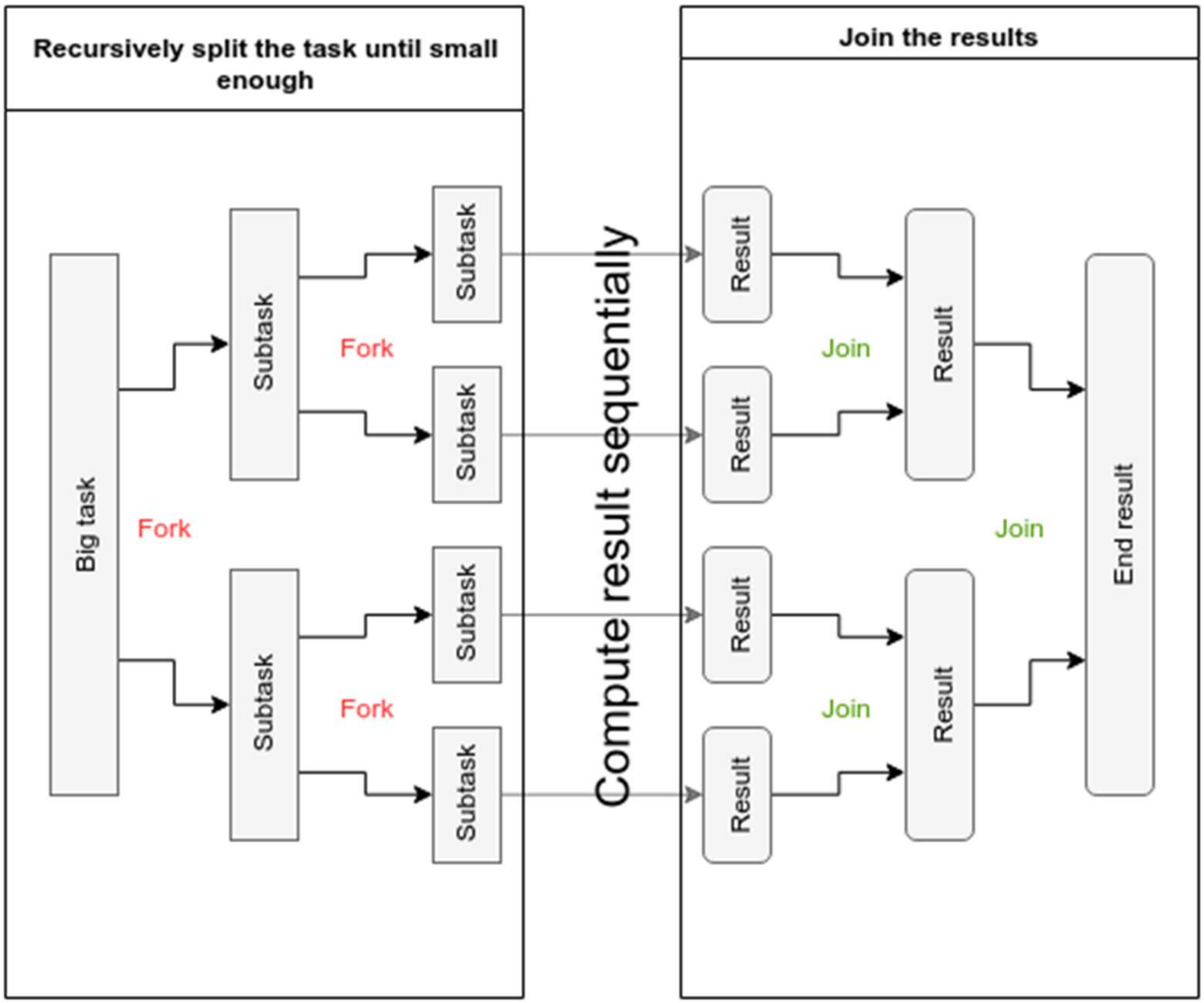
```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

# Understanding the framework classes

- Just like an ExecutorService executes an implementation of either the Runnable or the Callable, the ForkJoinPool class invokes a task of the type ForkJoinTask, which you have to implement by extending one of its two subclasses:
  - RecursiveAction, which represents tasks that do not yield a return value, like a Runnable.
  - RecursiveTask, which represents tasks that yield return values, like a Callable.
- These classes contain the compute() method, which will be responsible for solving the problem directly or by executing the task in parallel.
- ForkJoinTask subclasses also contain the following methods:
  - fork(), which allows a ForkJoinTask to be scheduled for asynchronous execution (launching a new subtask from an existing one).
  - join(), which returns the result of the computation when it is done, allowing a task to wait for the completion of another one.

# Understanding the framework classes

- First, decide when the problem is small enough to solve directly. This acts as the base case.
- A big task is divided into smaller tasks recursively until the base case is reached.
- Each time a task is divided, call the fork() method to place the first subtask in the current thread's deque.
- Then call the compute() method on the second subtask to process it recursively.
- Finally, to get the result of the first subtask, you call the join() method on this first subtask.
- This should be the last step because join() will block the next program from being processed until the result is returned.


Recursively split the task until small enough
Big task
Fork
Subtask
Subtask
Fork
Fork
Subtask
Subtask
Subtask
Subtask
Compute result sequentially
Join the results
Result
Result
Result
Result
Join
Join
Result
Result
Join
End result

# Understanding the framework classes

- To submit a task to the thread pool, use the execute(ForkJoinTask<?> task) as follows:

```
forkJoinPool.execute(recursiveAction);
recursiveAction.join();

// Or

forkJoinPool.execute(recursiveTask);
Object result = recursiveTask.join();
```

```
forkJoinPool.execute(recursiveAction).join();
// Or if a value is returned
Object result = forkJoinPool.execute(recursiveTask).join();
```

# Understanding the framework classes

- Typically, one can use invoke(ForkJoinTask), which performs the given task, returning its result upon completion:

```
forkJoinPool.invoke(recursiveAction);
// Or if a value is returned
Object result = forkJoinPool.invoke(recursiveTask);
```

# Implementation

To find the sum of all the elements in a list

# Implementation

**Step1**:Create a class that extends from RecursiveTask.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer>{

}
```

# Implementation

**Step2**: Choose a value that represents the number of elements that can be processed sequentially without any problem. A value neither too small nor too big.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer> {
 public static final int SEQUENTIAL_THRESHOLD = 10;

        private int lo, hi;
        private int[] arr;
}
```

# Implementation

**Step3**: Pass to the class constructor the data to work and save it as an instance variable.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer> {
//..
 public ForkJoinRecursiveSum(int[] arr, int lo, int hi) {
                this.lo = lo;
                this.hi = hi;
                this.arr = arr;
        }
}
```

# Implementation

**Step4**: Implement compute( ) to create tasks by passing sublists without having to create a new list every time.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer> {
 // ...
 @Override
public Integer compute() {
        if (hi - lo <= SEQUENTIAL_THRESHOLD) {
            int ans = 0;
            for (int i = lo; i < hi; i++) {
                ans += arr[i];
            }
            return ans;
        } else {
            int mid = (lo + hi) / 2;
            ForkJoinRecursiveSum left = new ForkJoinRecursiveSum(arr, lo, mid);
            ForkJoinRecursiveSum right = new ForkJoinRecursiveSum(arr, mid, hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}
```

# Implementation

**Step5**: Implement a method to compute the sum in the class.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer> {
  // ...

  public static int sum(int[] arr) throws InterruptedException {
          return fjPool.invoke(new ForkJoinRecursiveSum(arr, 0, arr.length));
      }
}
```

# Implementation

**Step6**: Add a main method to execute the class.

```
public class ForkJoinRecursiveSum extends RecursiveTask<Integer> {
 // ...
private static final ForkJoinPool fjPool = new ForkJoinPool();
  public static void main(String[] args) {
    int[] arr = new int[100];
    for (int i = 0; i < arr.length; i++) {
          arr[i] = i;
     }
     int sum = sum(arr);
    System.out.println("Sum: " + sum);
   }
}
```