Fork/Join Framework

Split up into smaller tasks

Outline

- Objectives
- Fork/Join
- ForkJoinTask
- Important Methods
- Method calling order
- Example

Objectives

Use parallel Fork/Join Framework.

Fork/Join

Splitting up the problem is known as **FORKING** and combining the results is known as **JOINING**.

ForkJoinPool use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work

ForkJoinTask

both implements java.util.concurrent.ForkJoinTask

RecursiveAction
which is the equivalent of Runnable in the sense that it **DOESN'T** return a value.

RecursiveTask<V>
which is the equivalent of Callable in the sense that it **DOES** return a value.

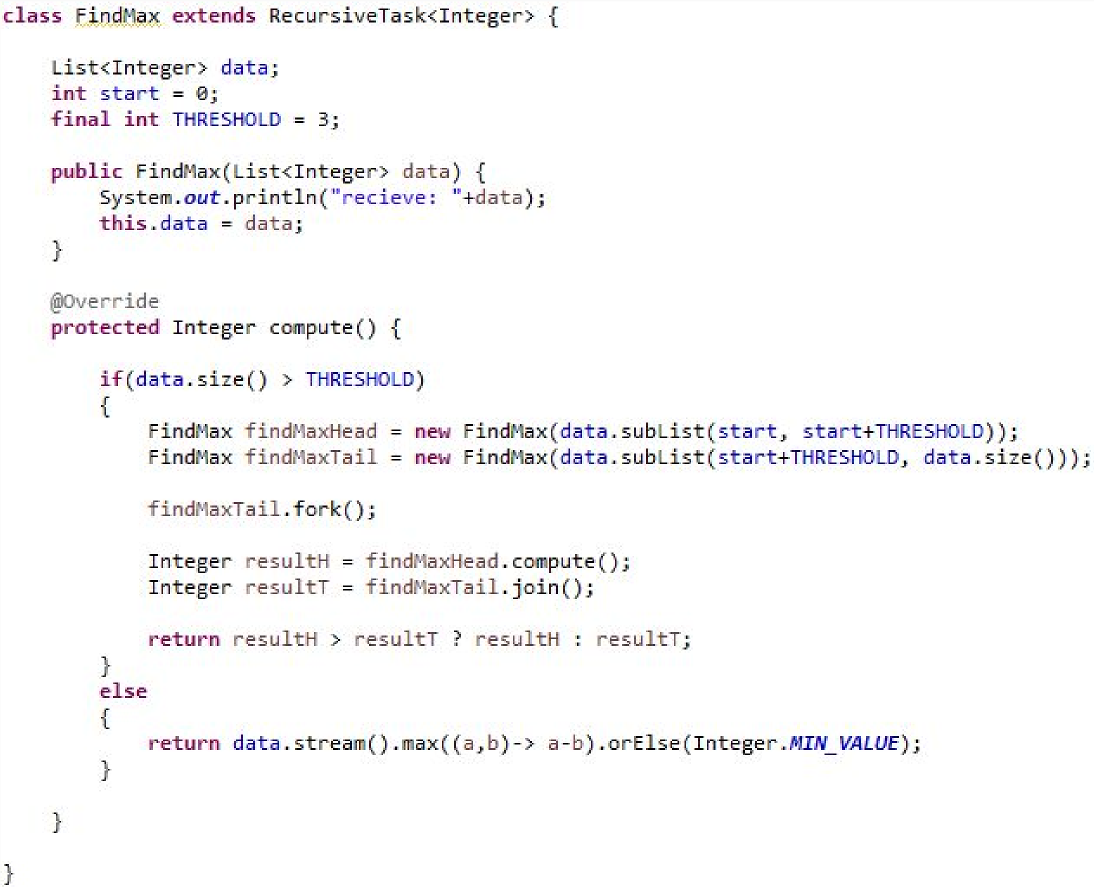
Important Methods

```
ForkJoinTask<V> fork()
V join()
// if you extend RecursiveAction
protected abstract void compute()
// if you extend RecursiveTask
protected abstract V compute()
```

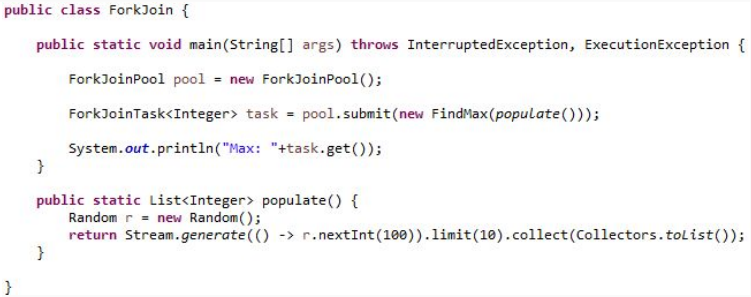
Method Calling Order

Fork > Compute > Join

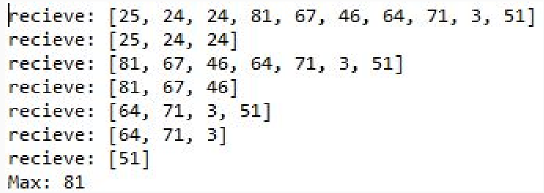
Example



```java
class FindMax extends RecursiveTask<Integer> {

    List<Integer> data;
    int start = 0;
    final int THRESHOLD = 3;

    public FindMax(List<Integer> data) {
        System.out.println("recieve: "+data);
        this.data = data;
    }

    @Override
    protected Integer compute() {

        if(data.size() > THRESHOLD)
        {
            FindMax findMaxHead = new FindMax(data.subList(start, start+THRESHOLD));
            FindMax findMaxTail = new FindMax(data.subList(start+THRESHOLD, data.size()));

            findMaxTail.fork();

            Integer resultH = findMaxHead.compute();
            Integer resultT = findMaxTail.join();

            return resultH > resultT ? resultH : resultT;
        }
        else
        {
            return data.stream().max((a,b)-> a-b).orElse(Integer.MIN_VALUE);
        }

    }

}
```



```java
public class ForkJoin {

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

        ForkJoinPool pool = new ForkJoinPool();

        ForkJoinTask<Integer> task = pool.submit(new FindMax(populate()));

        System.out.println("Max: "+task.get());
    }

    public static List<Integer> populate() {
        Random r = new Random();
        return Stream.generate(() -> r.nextInt(100)).limit(10).collect(Collectors.toList());
    }

}
```



```
recieve: [25, 24, 24, 81, 67, 46, 64, 71, 3, 51]
recieve: [25, 24, 24]
recieve: [81, 67, 46, 64, 71, 3, 51]
recieve: [81, 67, 46]
recieve: [64, 71, 3, 51]
recieve: [64, 71, 3]
recieve: [51]
Max: 81
```

Key Points

- The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.
- This is done through recursion, where you keep splitting up the task until you meet the base case, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.
- Splitting up the problem is known as **FORKING** and combining the results is known as **JOINING**.

Key Points

- The main class of the Fork/Join framework is java.util.concurrent.ForkJoinPool, which is actually a subclass of ExecutorService.
- Just like an ExecutorService executes a task represented by a Runnable or a Callable, in the Fork/Join framework a task is represented by a subclass of either RecursiveAction (that **DOESN'T** return a value) or RecursiveTask<V> (that **DOES** return a value).

Key Points

- However, unlike the worker threads that an ExecutorService uses, the threads of a ForkJoinPool use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work.
- A ForkJoinTask object has three main methods, fork(), join(), and compute(). The **ORDER** in which you call the methods is **IMPORTANT**.

Key Points

- You must first call fork() to queue the first subtask, then compute() on the second subtask to process it recursively, and then join() to get the result of the first subtask.