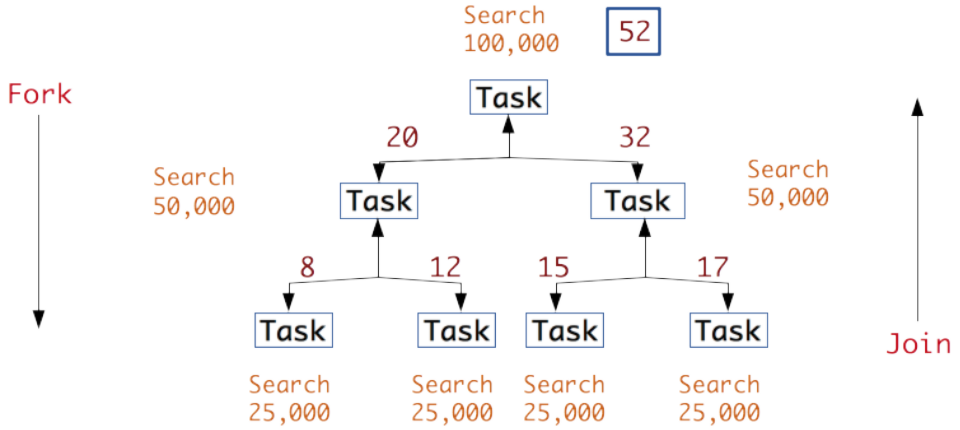
15. Notes - Fork Join Framework

Fork Join Framework -

- Available from Java7
- An ExecutorService for ForkJoinTasks
- It differs from ExecutorService by virtue of employing Work-stealing. i.e. if a worker thread has no tasks in the pipeline it will take the task from the task queue of the other busy thread so that the workload is efficiently balanced.
- To access the pool, A static common pool is available for the application and it can be accessed through commonPool() method of the ForkJoinPool class. Using the commonPool is the preferred approach because creating multiple thread pools might have an adverse impact on the performance of the application.
  `ForkJoinPool pool = ForkJoinPool.commonPool();`
- For applications which need separate thread pools, we can construct the ForkJoinPool using the level of parallelism needed and by default it is equal to the number of processors.
  `ForkJoinPool pool = new ForkJoinPool();`

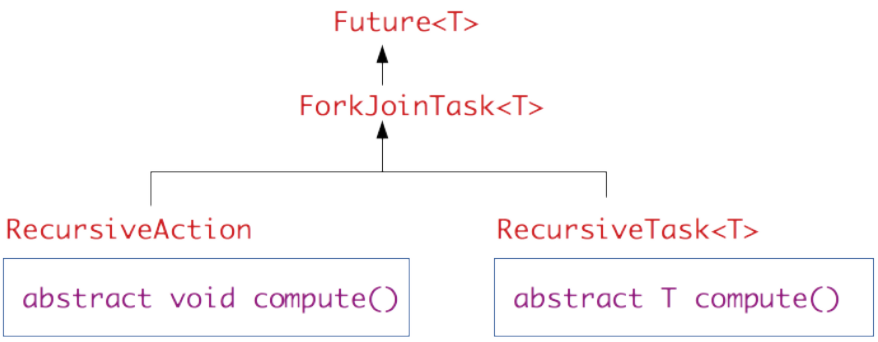
What is fork, join ?



The primary thought process behind fork is that each task is recursively split into subtasks and executed in parallel where as the join operation will wait for the completion of the task and combines the obtained results. To make it simpler, lets assume that the task is to search 100,000 unordered elements and returns the number of occurrences of an element. Assume that, a task will perform the search operation on its own if the list size is 25000. Here the first tasks splits the total elements into two sets of size 50000 and at second level task will split that into two sets of size 25000 each, and at this level it will not fork any subtasks because the size of the dataset is 25000 and as per our condition we shouLd not create sub tasks.

This is how recursion works as well, there will be a base condition and upon reaching that condition recursive call would be stopped. Similarly here we should put a threshold condition where we will not further fork.

Types of ForkJoinTask(s) -



ForkJoinTask implements Future interface so we can use it to extract the results once the task is done. ForkJoinTask is further divided into two subtypes one is RecursiveAction and RecursiveTask. These classes are abstract; and when you extend you need to override the compute method. If it is a RecursiveAction the compute operation doesn't return any value. Where as in case of RecursiveTask we can return a value. You can relate that with Runnable and Callable.

Submitting the tasks from outside non-fork join clients -

```
YourForkJoinTask task = ......;
ForkJoinPool pool = ForkJoinPool.commonPool();

// (1) Arranges for asynchronous execution of the given task.
pool.execute(task);

// (2) Invokes the task, waits for completion and returns the
// result.
result = pool.invoke(task);

// (3) Submits the task for execution
pool.submit(task);

// Later to obtain the result in case of (1) and (3).
Result = task.get();
```

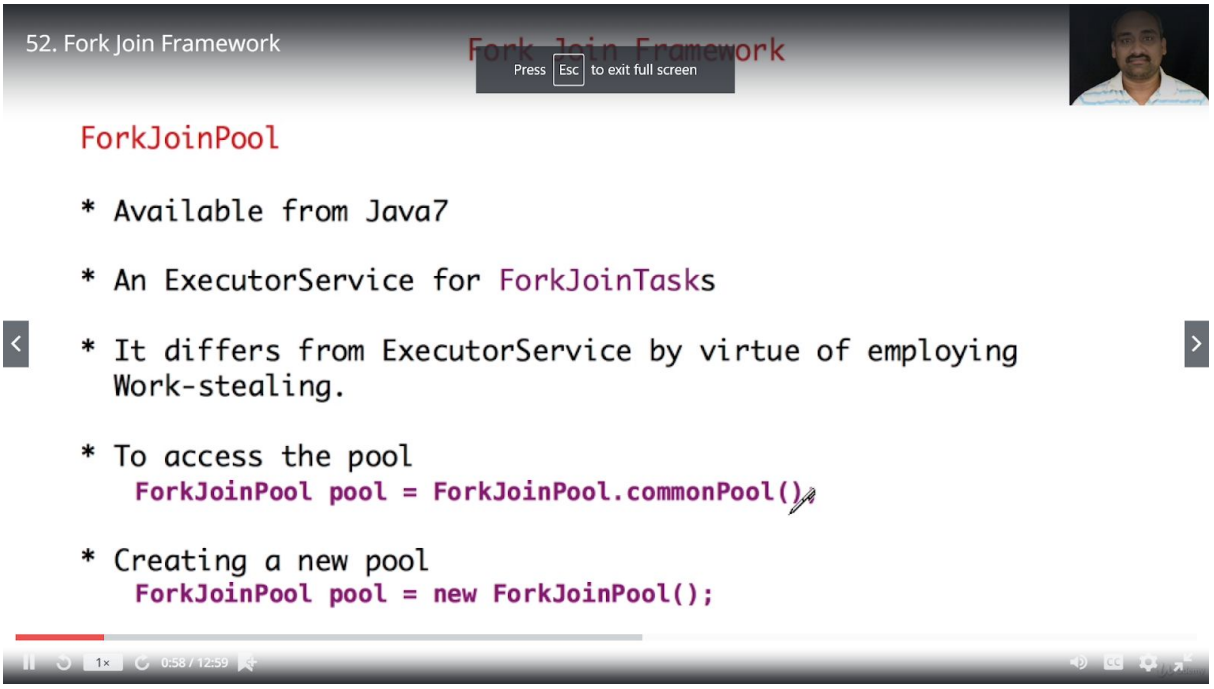
Submitting the tasks from with in the fork join computations -

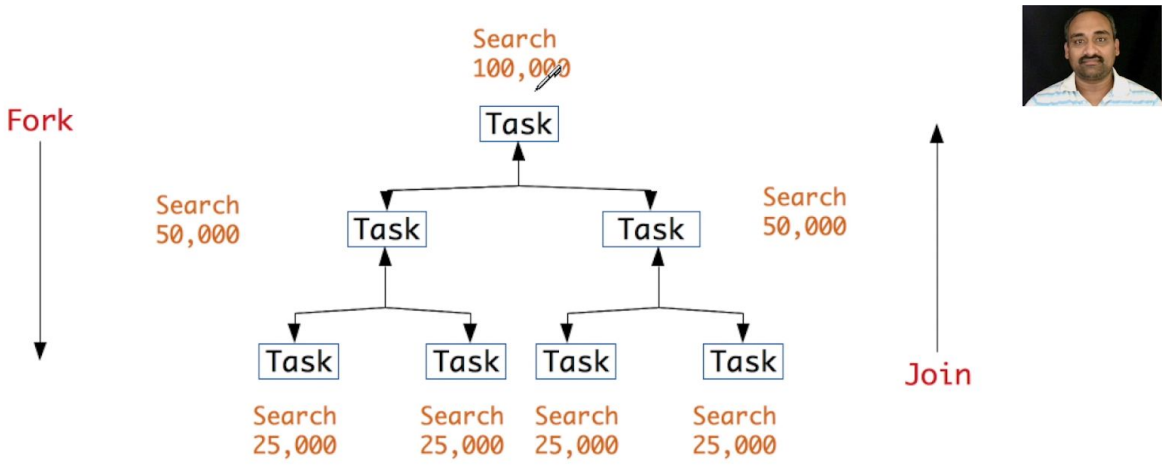
```
YourForkJoinSubTask subTask = ......;

// (1) Submit the task for execution
subTask.fork();

// join - waits for the task to complete the computation
// and returns its result.
result = subTask.join();

// (2) Direct approach - invoke the task and wait for result.
result = subTask.invoke();
```

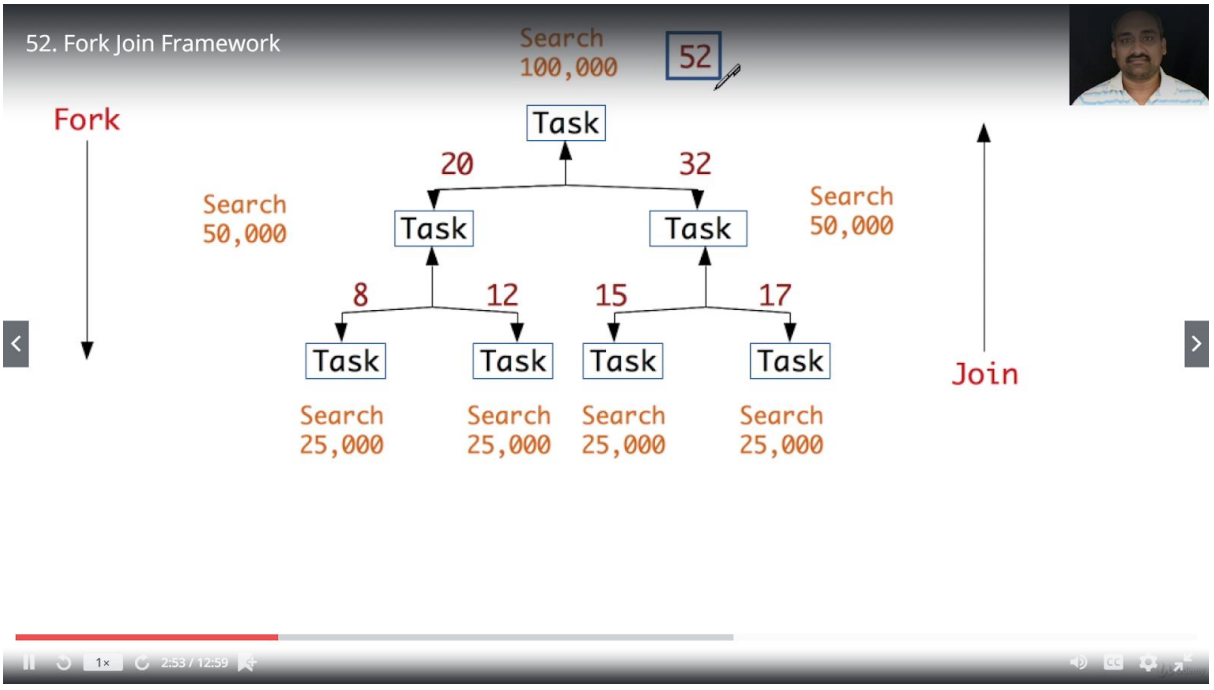
Example -

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class SearchTask extends RecursiveTask<Integer> {

   int arr[];
   int start, end;
   int searchEle;

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

   @Override
   protected Integer compute() {
       System.out.println(Thread.currentThread());
       int size = end - start + 1;
       if (size > 3) {
           int mid = (end + start) / 2;
           SearchTask task1 = new SearchTask(arr, start, mid, searchEle);
           SearchTask task2 = new SearchTask(arr, mid + 1, end, searchEle);
           task1.fork();
           task2.fork();
           int result = task1.join() + task2.join();
           return result;
       } else {
           return processSearch();
       }
   }

   private Integer processSearch() {
       int count = 0;
       for(int i = start; i <= end; i++) {
           if (arr[i] == searchEle) {
              count++;
```

```
            }
        }
        return count;
    }
}

public class Main {

    public static void main(String[] args) {
        int arr[] = {6, 2, 6, 4, 5, 6, 7, 8, 6, 10, 11, 6};
        int searchEle = 6;
        int start = 0;
        int end = arr.length - 1;

        ForkJoinPool pool = ForkJoinPool.commonPool();
        SearchTask task = new SearchTask(arr, start, end, searchEle);
        int result = pool.invoke(task);

        System.out.printf("%d found %d times", searchEle, result);
    }
}
```




Search
100,000
Task
Fork
Search
50,000
Task
Task
Search
50,000
Task
Task
Task
Task
Join
Search
25,000
Search
25,000
Search
25,000
Search
25,000


52. Fork Join Framework
Search
100,000
52
Fork
Task
20
32
Search
50,000
Task
Task
Search
50,000
8
12
15
17
Task
Task
Task
Task
Join
Search
25,000
Search
25,000
Search
25,000
Search
25,000
1×
2:53 / 12:59

```
YourForkJoinTask task = .....;

ForkJoinPool pool = ForkJoinPool.commonPool();


// (1) Arranges for asynchronous execution of the given task.
pool.execute(task);

// (2) Invokes the task, waits for completion and returns the
// result.
result = pool.invoke(task);

// (3) Submits the task for execution
pool.submit(task);

// ....
// Later to obtain the result in case of (1) and (3).
Result = task.get();
```

From With in Fork Join Tasks

```
YourForkJoinSubTask subTask = ......;


// Submit the task for execution
subTask.fork();
result = subTask.join();


// Invoke the task and wait for result.
result = subTask.invoke();
```