

Merge Sort - Multithreaded

Divide → Conquer

<https://github.com/KnightKnight27/scaler-os-batch>

Divide → sort both parts
independently

Runnables are used to only run a task thread.

Callables are used to run and return data
from a thread.

Futures in Java

```
main () {  
    print ("something");  
    int i = Thread(55);  
    ...  
}
```

Something
Bye
Thread output

```
} print("bye")  
}
```

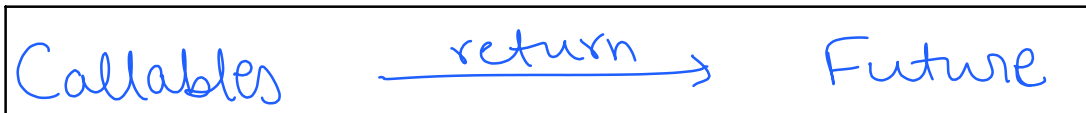
This is because the Thread is different from the thread of the Main() function.

```
main() {  
    print("something");  
    int i = Thread(55);  
    print("Bye")  
    print(i)  
}
```

This will cause an error bcos
the main thread executes it
before Thread completes

```
main() {  
    print("something");  
    FutureObject i = Thread(55);  
    print("Bye");  
    print(FutureObject.get(i))  
}
```

This will stop the further
execution of the whole
program until i is
calculated.



```
MergeSorter leftMergeSorter = new MergeSorter(leftArray, executorService);
MergeSorter rightMergeSorter = new MergeSorter(rightArray, executorService);

Future<List<Integer>> leftSortedArrayFuture = executorService.submit(leftMergeSorter);
Future<List<Integer>> rightSortedArrayFuture = executorService.submit(rightMergeSorter);

List<Integer> sortedArray = new ArrayList<>();|
int i = 0;
int j = 0;

List<Integer> leftSortedArray = leftSortedArrayFuture.get(); // code will not go to the ne
List<Integer> rightSortedArray = rightSortedArrayFuture.get();
```

↳ This callable will return
a Future object

<https://github.com/KnightKnight27/scaler-os-batch/blob/main/MergeSorted2.java>

↳ This will stop further
execution of the program
until the sorted array returns

Adder & Subtractor

Adder

```
for (int i=0; i<100; i++){  
    count += i  
}
```

Subtractor

```
for (int i=0; i<100; i++){  
    count -= i  
}
```

The count variable is commonly shared b/w the two functions

Count actually is an object of count variable.

```
public class Client {  
    public static void main(...){  
        SharedCount sharedCount = new SharedCount();  
  
        Adder adder = new Adder(sharedCount);  
        Subtractor subtractor = new  
        Subtractor(sharedCount);  
  
        Thread t1 = new Thread(adder);  
        Thread t2 = new Thread(subtractor);  
  
        t1.start(); ➡ both started
```

```
public class SharedCount {  
    this.count = 0;  
}
```

But since both start one after another, they are working asynchronously and hence it is possible that race conditions occur and one function's preemption happens more than the other and hence it produces gibberish output.



```

t2.start();
t1.join();
t2.join();
}

```

→ wait for finish → Asynchronous
Parallel execution

↳ join() function also stops further execution

```

public class Client {
    public static void main(...){
        SharedCount sharedCount = new SharedCount();

```

```

        Adder adder = new Adder(sharedCount);
        Subtractor subtractor = new
        Subtractor(sharedCount);

```

```

        Thread t1 = new Thread(adder);
        Thread t2 = new Thread(subtractor);

```

```

t1.start();
t1.join();

```

→ first start & finish t1 completely

```

t2.start();
t2.join();

```

→ then move on to t2

Synchronous
No parallelization

The problem with the gibberish output due to race conditions is because of how the increment operation works.

count += 1 is actually comprised of 3 different ops:

1. Read count
2. calculate count+1

t1	t2
read count	read count

3. Overwrite count with count+1

So during execution, preemption may occur before any of these ops actually occur.

Assignment:

1. Implement Multithreaded Quick Sort
2. Read about Generics in Java
3. Read about Locks and Semaphores

