

## Agenda:

- Callables, Futures
  - Merge sort multithreaded.
- ≡ { sync ← why?  
ideal sol<sup>n</sup>?

direct image  $\xrightarrow{\text{Rotate}}$  Read  $\xrightarrow{\text{Rotate}}$  Read ...

main

image  
0°

90°

180°

270°

return some info

How to get the data back to paint thread!

callable, just like runnable but it allows a thread to return data.

How to use callables:

① Task?

class \_\_\_\_\_ implements Callable {  
 ↑  
 String call() {  
 }  
 }  
 you are going to return a String?  
 <T> call() {  
 }

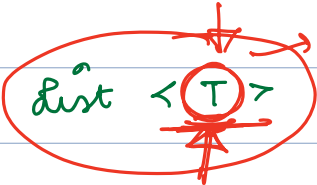
# Generics

List < Animal >

List < int >


List < List < Animal > >

class List < T >



placeholder

interface callable < T > {



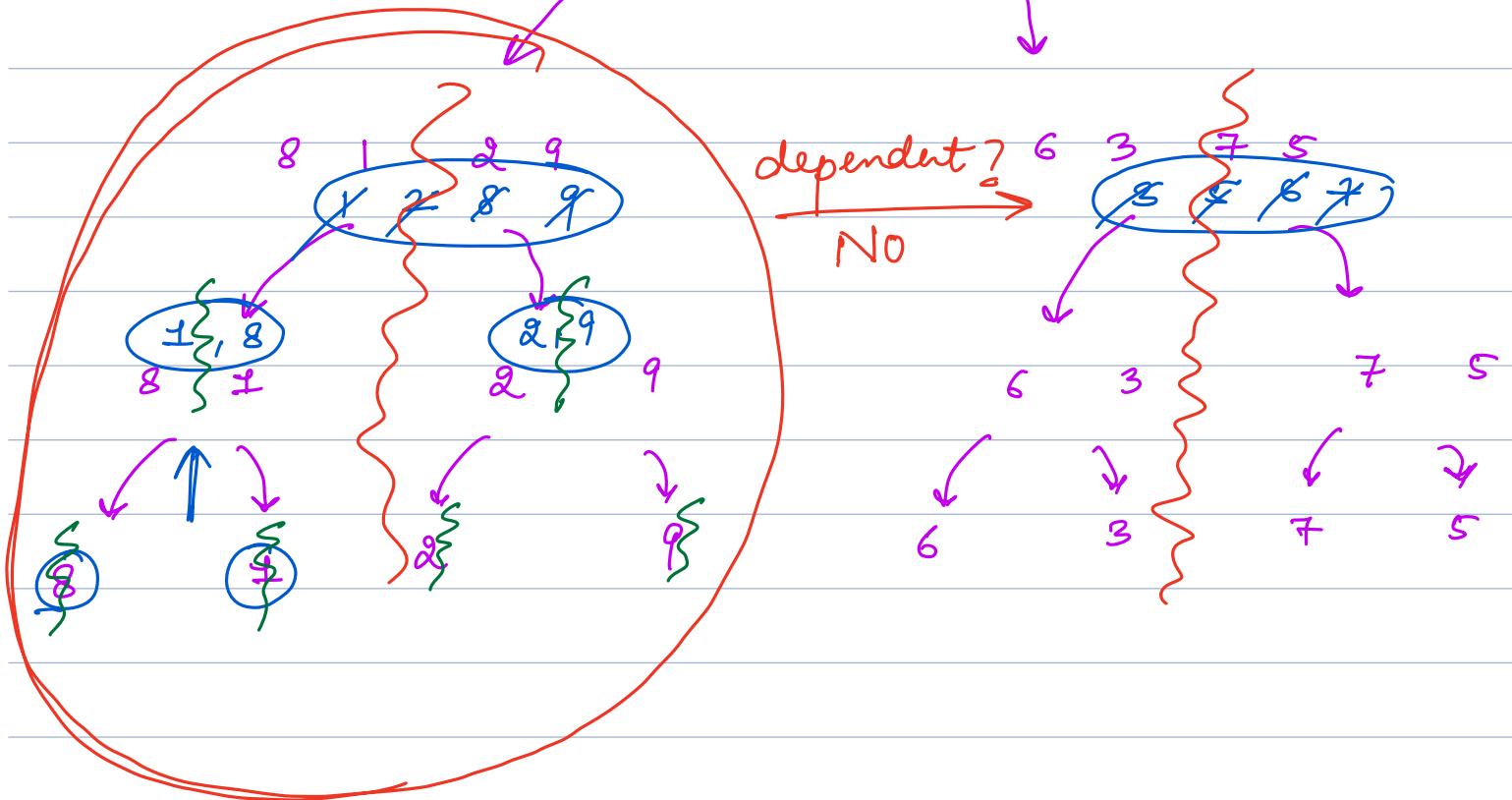
T call();

}

# Merge-sort

8 1 2 9 6 3 7 5

1 2 3 5 6 7 8 9



SortArray (array):

left Array = —

Right Array = —

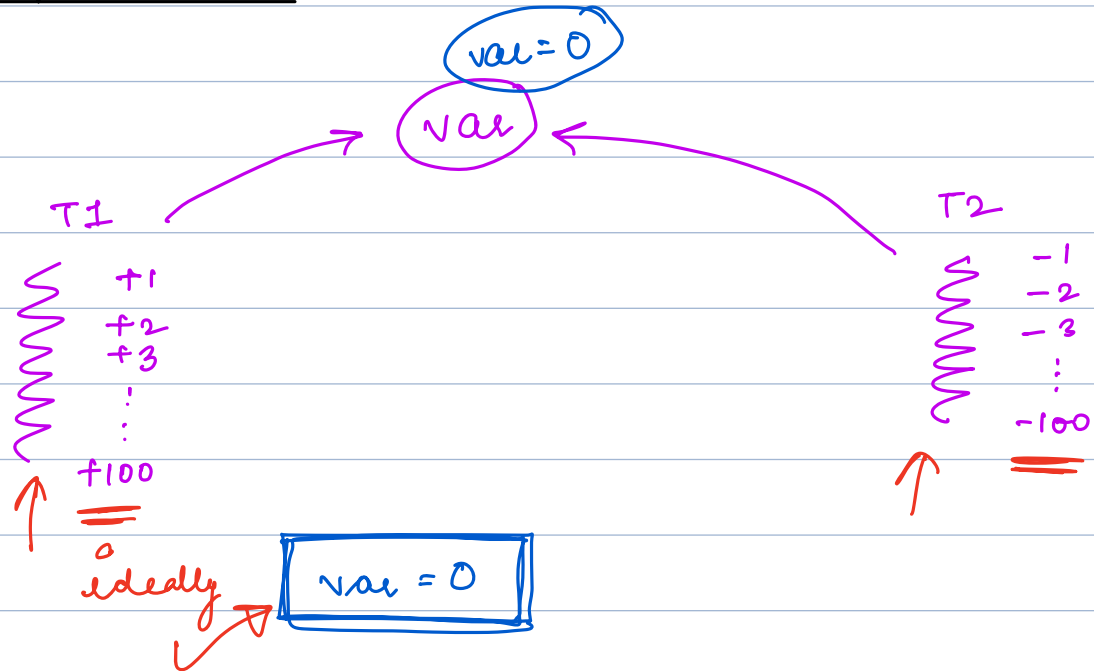
{ — left array

{ — right array

merge the data from both trees

# Synchronization

## Adder / subtractor



## Identify the tasks

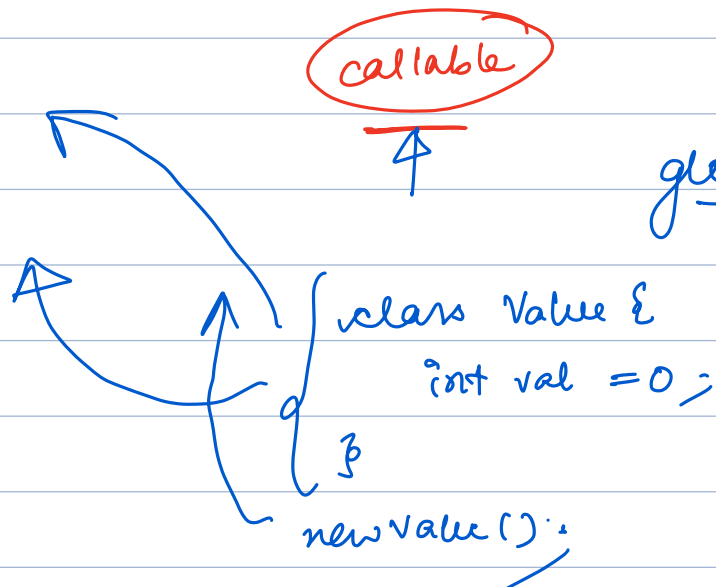
- Addition
- Subtraction

class adder

class subtractor

callable

global variables



value += i; → is it single operation?

$t \xrightarrow{0} \leftarrow \text{get}(\text{value})$   
 $t = t + i; \quad t = 1$   
~~value = 1~~  $\text{value} \leftarrow t$

$t \xrightarrow{0} \leftarrow \text{get}(\text{value})$   
 $t = t - i; \quad t = -1$   
~~value = -1~~  $\text{value} \leftarrow t$

## Data sync problem

some variable → shared → two threads  
↓

quite a lot of chances → simultaneously  
→ inconsistent data

# Why sync problem happen :-

①

critical section  
important / careful about

— part / portion of the code where the concurrency issues can happen.

— part of code which is working on shared data

```
1 print("Hello");  
2 val += i;  
3 print("Hi");  
4 val *= i;  
5 print("Bye");
```

```
1 print("Hey!");  
2 val -= i;  
3 print("Bye");
```

```
addq"(4)  
print(que[size-1])
```

```
delete(que[size-1]);
```

1 2 3 ~~4~~

②

Race condition

Mutex  
Semaphore

"Race of completing the task"

Two threads enters the CS at the same time.

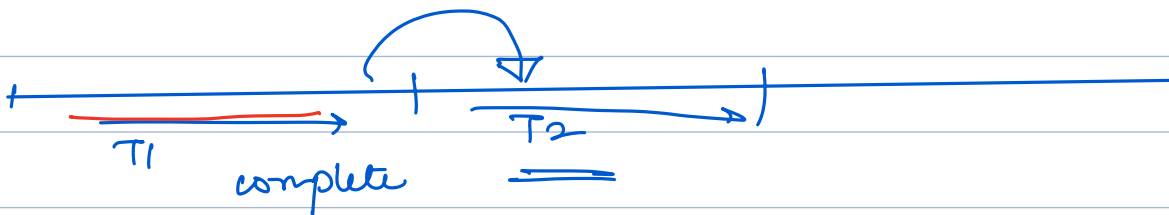
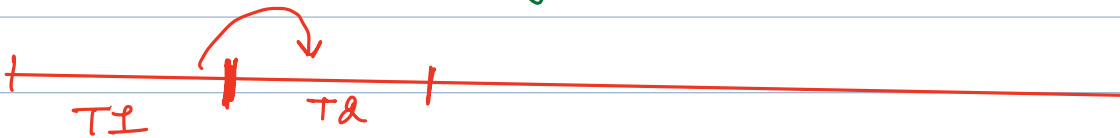
③

preemptiveness

life  
core

we move from one task to  
another before even  
completing the task

concurrent  
module





# Properties of a good sol<sup>n</sup>

## ① Mutual Exclusion

→ must have

→ only 1 thread to be allowed  
to enter CS at the  
same time  
→ remove race cond<sup>n</sup>

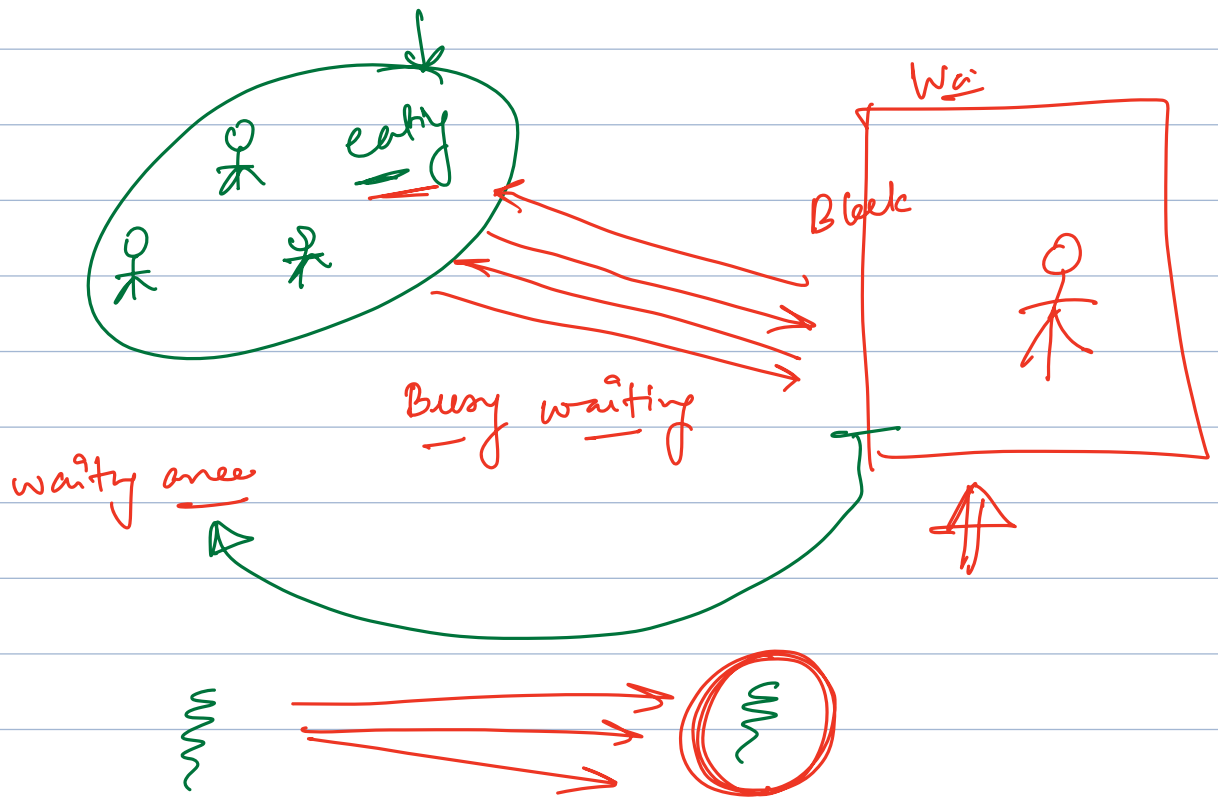
## ② Progress

→ overall system should  
make progress.  
→ overall system should not get to  
a halt

## ③ Bounded waiting:

- No thread should have to wait infinitely.
- after how many tasks your  
task will be allowed.

## ④ No Busy waiting:-



while ( true ) {

    x = check if allowed()

    if ( x is true )

        enter;

}