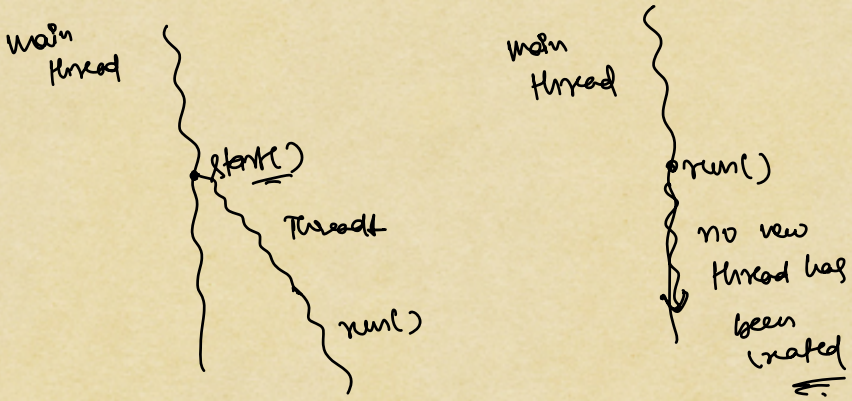
* Run and Start method
* Coding - examples
* Executors
* Coding - examples
* Callables
* Coding - examples

→ Run and Start :

run() → adds the task that we want to do in a different thread, but it runs on current thread.

start() → creates the thread, and calls the run() method on the new thread.



→ A CPU executes a process ⇒ NO ⇒ executes a thread

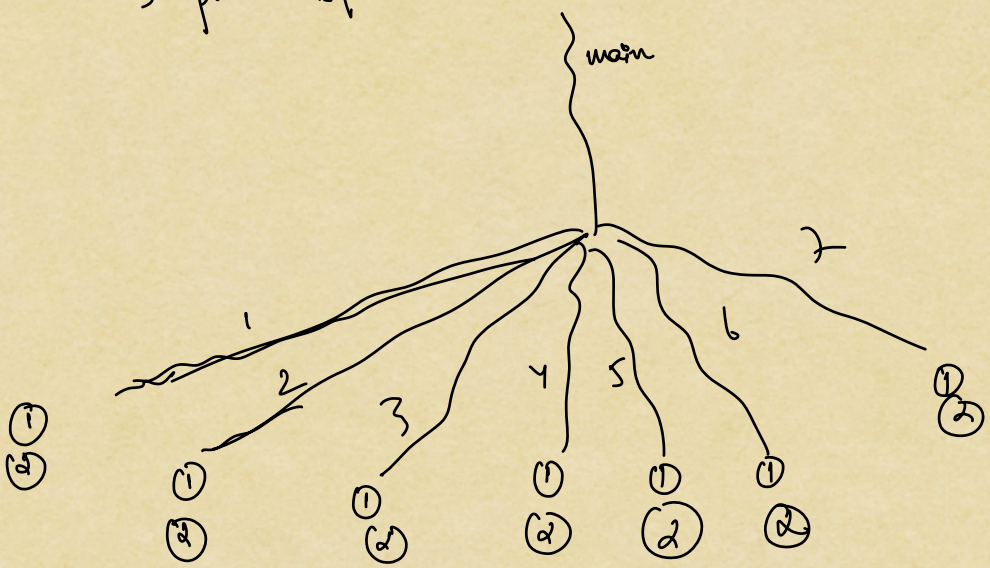
→ All processes have thread ⇒ Yes

→ A thread is slow context switching ⇒ No.

→ A thread occupies more memory than process ⇒ NO

i) print name

ii) print sq.



* Multi-threading in general is undeterministic
* Debugging is very difficult

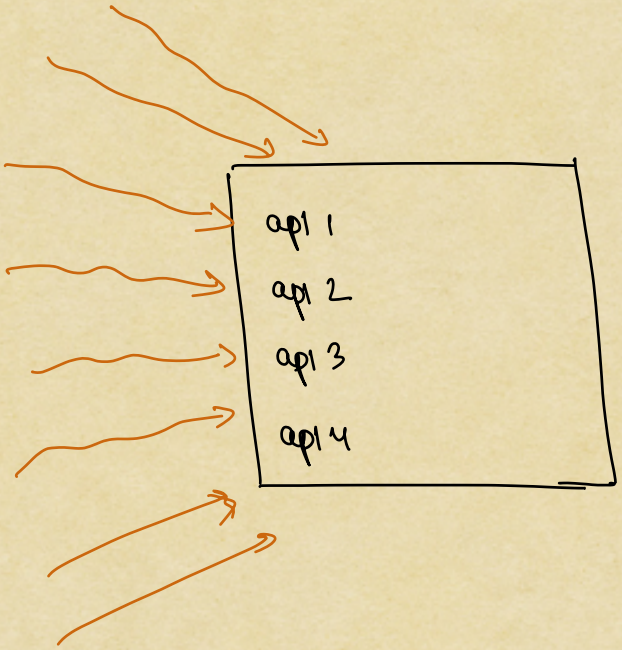
⇒ Client

* Create a task
* Create the thread
* Decide when the thread will run

↓

t.start()

⇒ Building a server:-



for every request, we create a new thread.

⇒ problems

* too many things to handle ← (executor framework ↑ thread pool)
* thread creation is expensive ← (executor framework ↑ thread pool)

Client

* knows about the things that need to be built

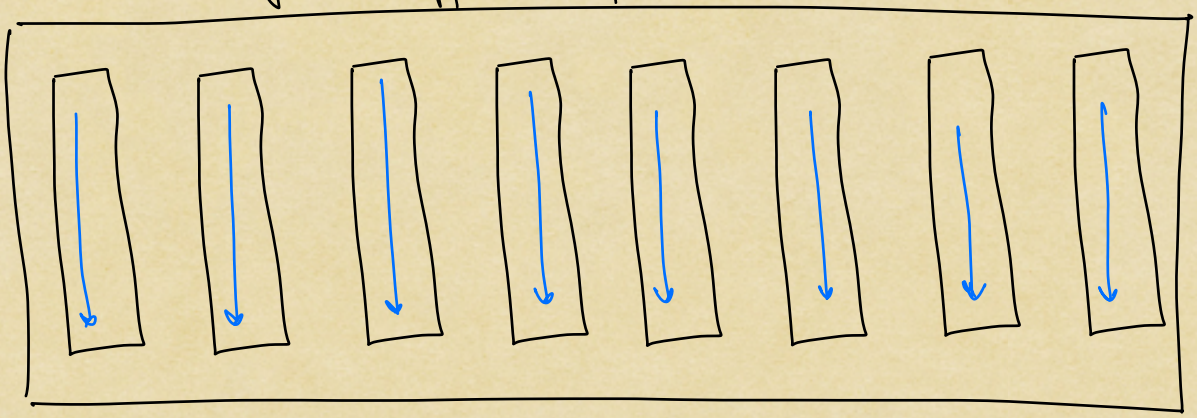
Executors Service

* how to efficiently run multiple threads parallely.

Car factory

→ 2000 cars.

factory can support 8 production lines



* After every car is made, production line get destroyed. and we need to recreate the production line to create a new car.

→ Not efficient

* We need to somehow, recycle the production lines.

↓

Similarly, threads should not be destroyed, and should be reused.

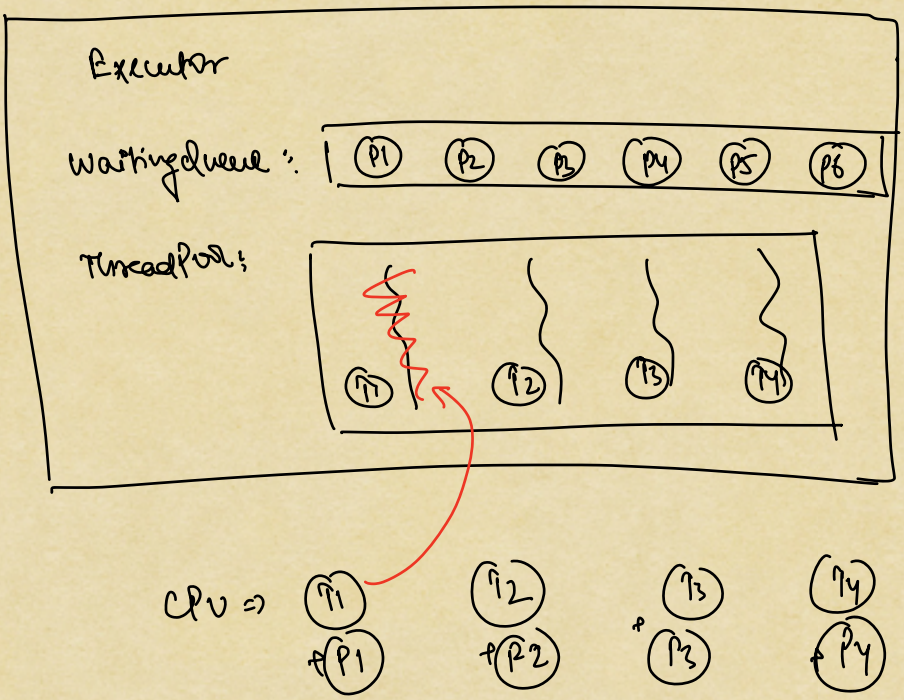
↓

maintained by ExecutorService

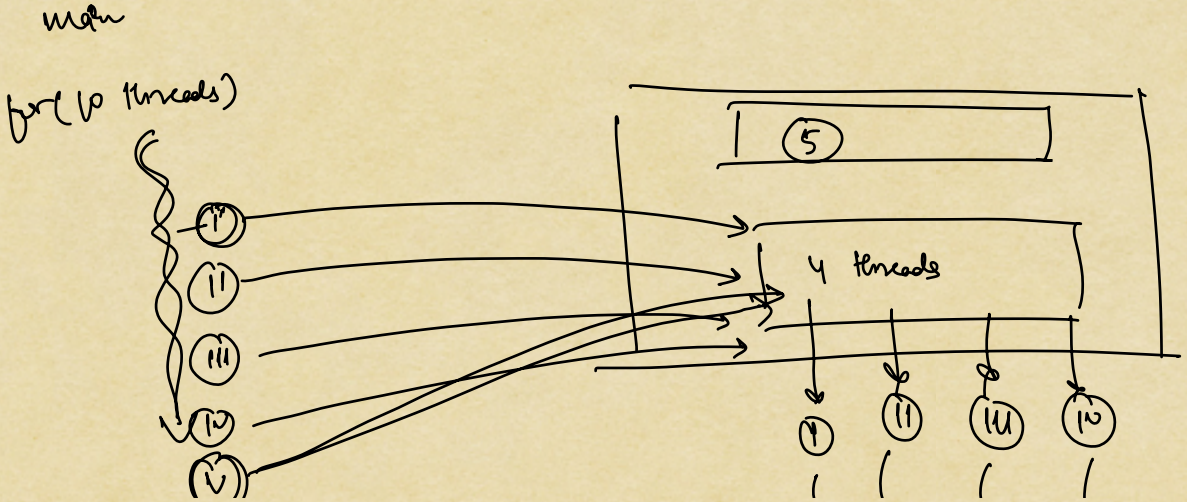
↓

maintains a thread Pool.

Thread Pool:- Collection of threads that can be used to execute tasks parallelly



→ Here we are reusing the threads, and not recreating them.

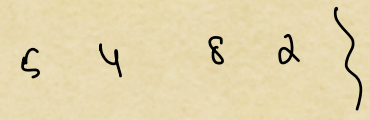


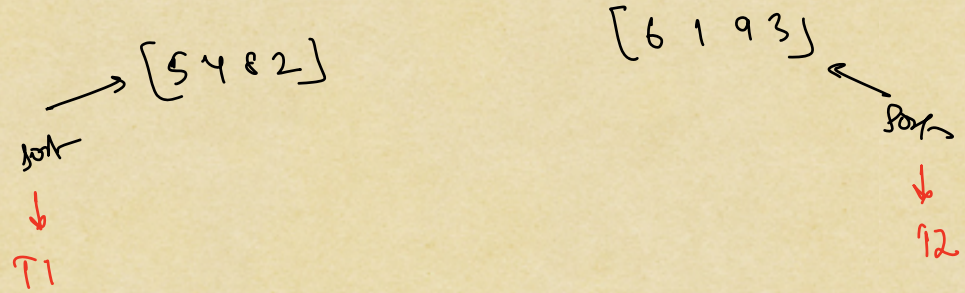
→ Runnable
↓
run

Callable
↓
call

⇒ **Callable :-** In case the task which we want to handle concurrently (as in multi-threaded), we want to return something we use callable | call()

⇒ Merge Sort :-

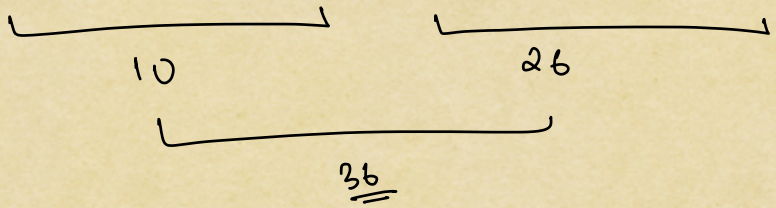




→ SOP for using Callable Interface :-

⇒ You are given 2 arrays, find the sum of all elements of both arrays

eg ⇒ [1 2 3 4] [5 6 7 8]



S1. Identify the task that you want to run in a parallel thread.

→ task class name should be a noun doing a verb.

ex: ArrayAdder

S2. Choose the return type for each task

↑ → Integer

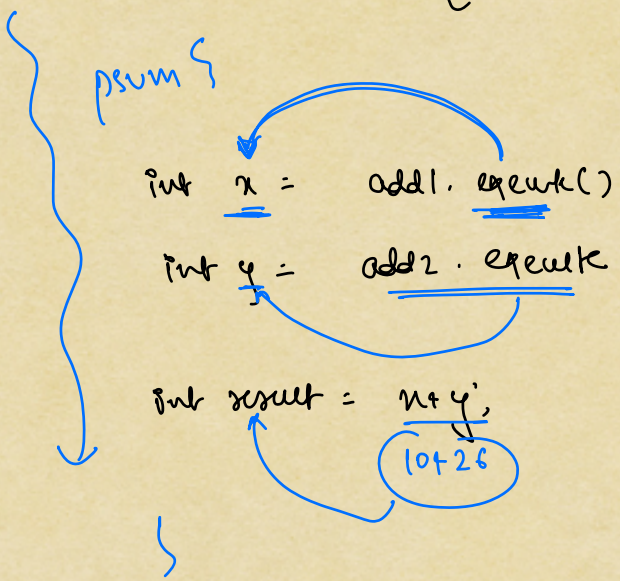
S3. Implement the Callable interface with return type

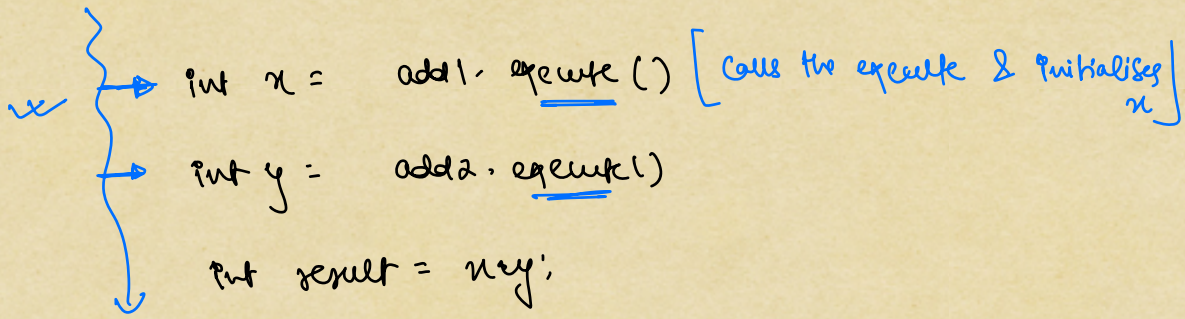
class ArrayAdder implement Callable<Integer>

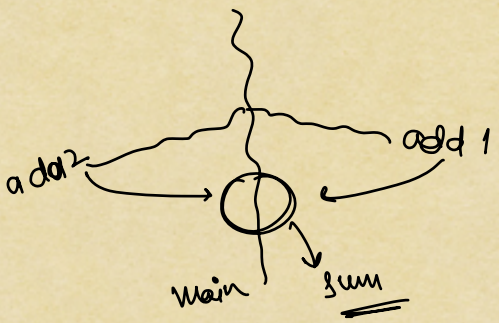
S4. Write the task logic in call method.

adder1 ⇒ [1 2 3 4] ⇒ 10

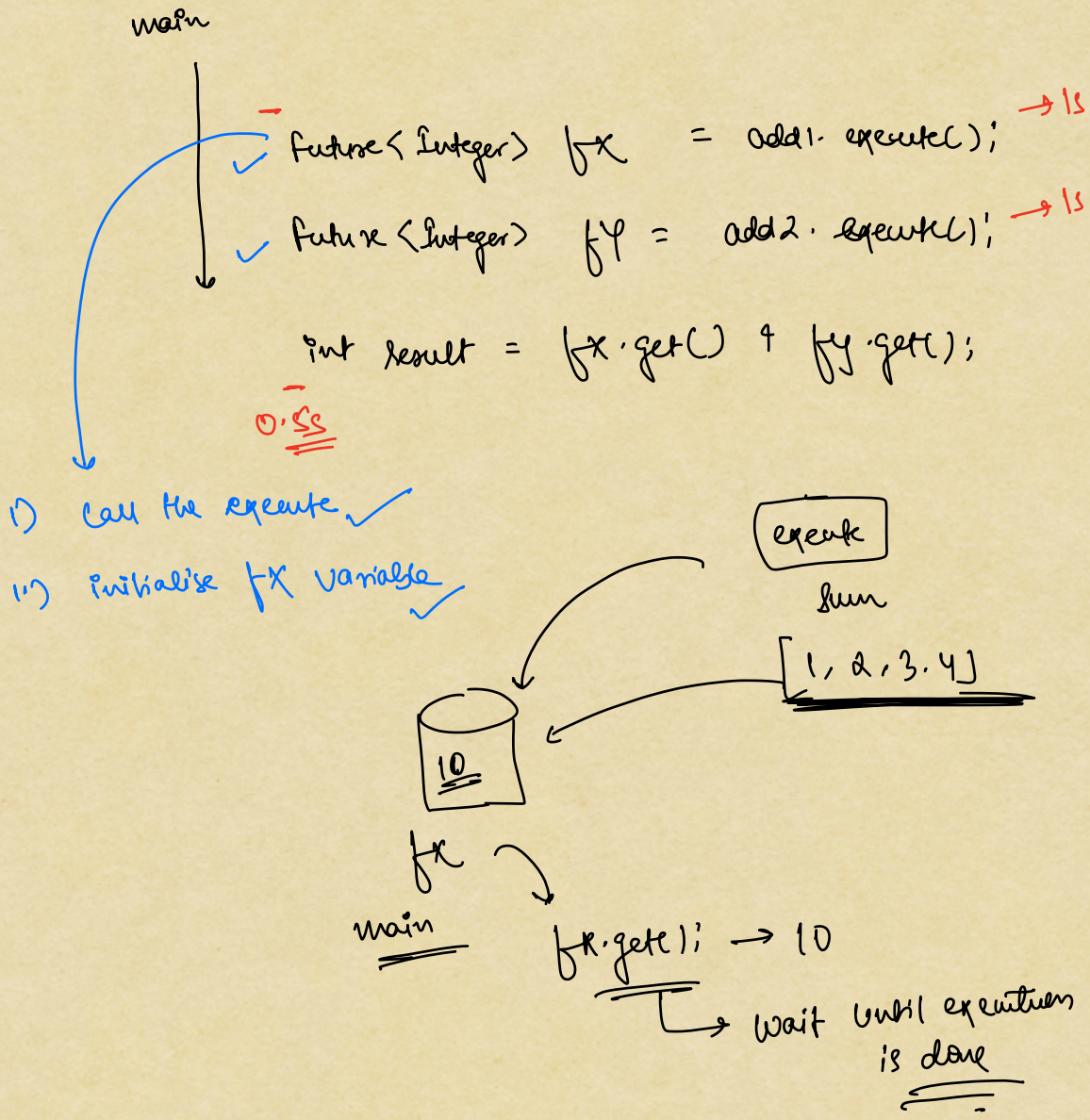
adder2 ⇒ [5 6 7 8] ⇒ 26







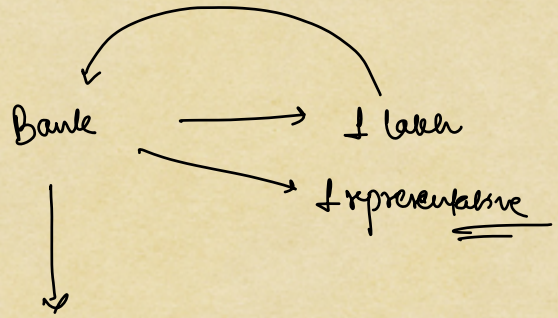
⇒ FUTURE :



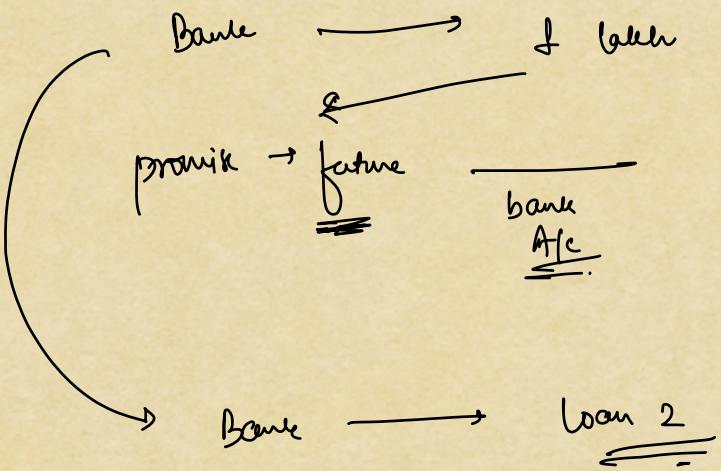
Business

Bakery Shop ⟶ IL Loan ⟶ Bank

Scenario 1



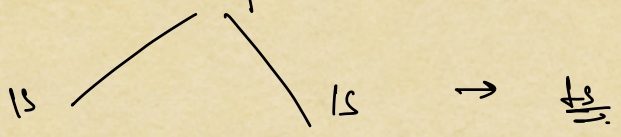
Scenario 2



In Callable we return a future, which ultimately holds the actual result, once the thread execution is completed.

ex ⇒ adding | arrayList ⇒ 1s

2 arrayList ⇒ 2s



Query → doubts