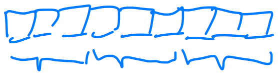
Important: Compilable code* → 5 mins left
→ Source code should compile completely.

PE2 → unit 36 → monad
Lab 6 → "Lazy" lab (not incl. infinite List)
Rec 9.
⇓
may be building upon it
Should be simpler

PE1
Avg = 19.4
Median = 22
203 ⇒ 28/30. ✓

Thread

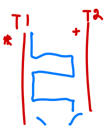
→ normally single thread. (single flow).
→ wait for one method to finish first.
→ one after the other.

Parallel stream.
→ multiple threads.
→ 
1 thread 1 thread 1 thread.

→ Java API: Thread.
↳ Thread (Runnable)
functional Interface
↳ void run()
↳ create various tasks.
↳ When created → not executing.
→ use new Thread .start();

→ each Thread has a name
⇒ getName()
→ also have currentThread(). (static)
⇒ (currently executed) Thread Object.
→ main method run in Main thread

Multiple threads
↳ different order will be obtained. everytime.



JVM jumping between.
⇒ nothing in the Thread API say what should wait for what

eg. Thread findPrime = new Thread(() → {

Sysout (Stream.iter (2, i → i+2)
. filter(x → isPrime(x))
. limit(500000)
. reduce ((x,y) → y)
. orElse(0));

});

findPrime . start();

Note: tasks now concurrently

```
while (findPrime.isAlive()) {
    try {
        Sysout (".");
        Thread.sleep(1000)
    } catch (InterruptedException e) {}
}
```

low level abstraction

Static → sleep()

→ causes current thread to sleep.

→ Must handle InterruptedException

isAlive()

→ check if a thread is alive (i.e. doing computation).

Coordinate between 2 threads.

→ send a message between them.

→ returning the value

do smthing the value.

Higher level Abstraction

Synchronous vs Asynchronous

⇒ Thread's Draw-backs.

↳ overhead

↳ single-use

↳ handling of Exception.

same thread? Thread that created this Thread?

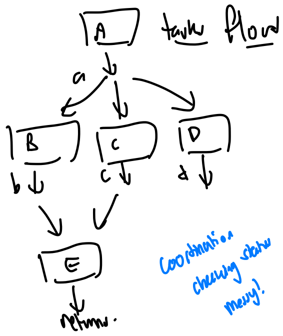
handling?

↳ not easy to pass information around

get around the issue of Overhead

use.

```
int foo (int x) {
    int a = taskA(x);
    int b = taskB(a);
    int c = taskC(a);
    int d = taskD(a);
    int e = taskE(b,c);
    return e;
}
```



coordination, checking status messy!

⇓

Maybe return none();?

```
int foo (int x) {
    Lazy<int> a = Maybe.of(taskA(x));
    Lazy<int> b = a.flatMap(i → taskB(i));
    . int c = taskC(a);
etc. int d = taskD(a);
    . int e = b.combine(c, (i,j) → taskE(i,j));
    return e;
}
```

flow of the result of execution.

⇓

CompletableFuture <T>. (Monad)

- a promise to do the computation
- side info if has been computed.

explain the flow of the execution, capture the dependencies. Passes the values around. Handle exceptions with .handle();

- . completedFuture() ⇒ create the computation flows,
  . thenComposeAsync()
  . thenCombineAsync().

. supplyAsync() ⇒ Takes in a supplier to calc. concurrently.

must be completed,

→ CompletableFuture <Integer> diff = ithPrime. thenCombine (jthPrime, (x,y) → x-y);

blocking call → wait until computation has happened, i.e. call at last minute when you need it.

→ diff.join() to get result.

→ Chain them and not worry about dependencies.

thenApplyAsync() ↔ map

thenComposeAsync() ↔ flatMap

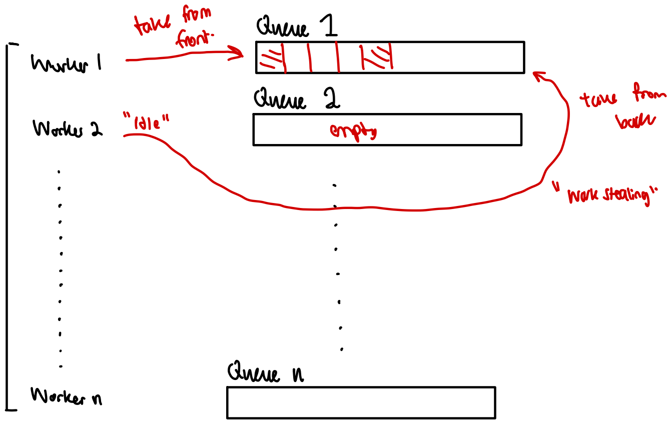
thenCombineAsync() ↔ combine.

Fork And Join

↳ ThreadPool → minimises overhead in creation of Thread

↳ Divide & Conquer model of computation. (In parallel).

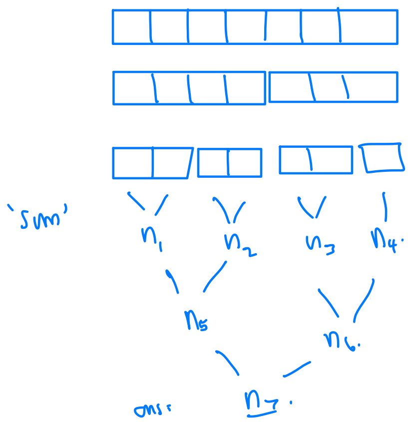
↳ fork() → divides the problem. → put to front of queue.
join() → combine the results.



Java API

RecursiveTask<T>

↳ compute() returns V.
does computation.



Order of forking = reverse order of joining

⇒ more efficient.

Why? — also called "ThreadPool"

Task Queue front

left.fork()

front

right.fork().

need to wait for R to be computed first.

∴ need to call join() on front of the queue.

or ⇒ work stealing. ⇒ worker to grab a task on the back.

END OF SEM 2!