```
static class Box {
  int i = 0;
}

public static void main(String[] args) throws I
  Box box = new Box();

  Runnable r1 = () -> {
    for (int i=1; i <= 10_000; i++) box.i++;
  };

  Runnable r2 = () -> {
    for (int i=1; i <= 10_000; i++) box.i--;
  };
```

```
Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);

t1.start();
t2.start();
```

```
Thread.sleep( millis: 1000);

System.out.println(box.i);
```

```
Runnable r1 = new Runnable() {
  @Override
  public void run() {
    for (int i=1; i <= 100_000; i++) box.i++;
  }
};
```

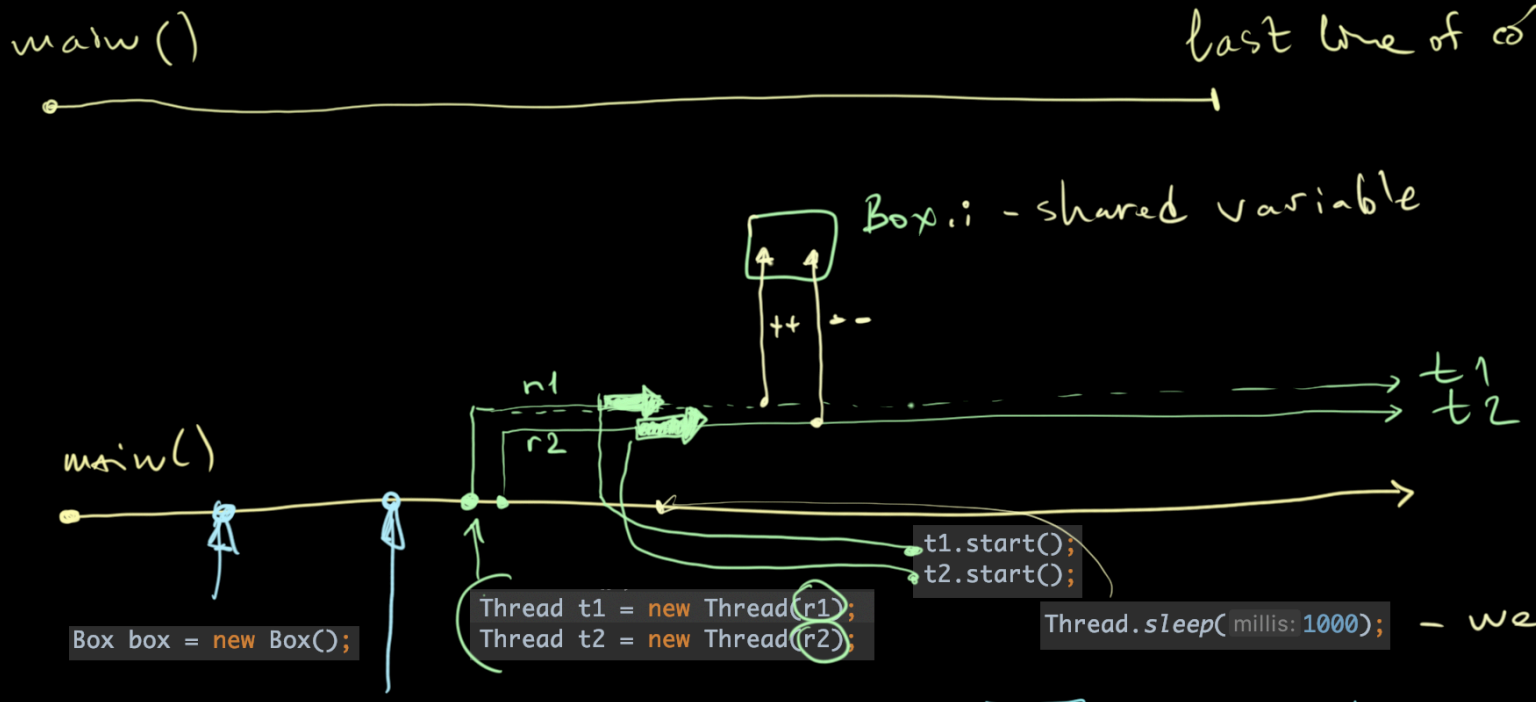
0 + 1 + 1 ... (10.000) ≈ 10.000

10.000 − 1 − 1 ... − 1 (10.000) ≈ 0

they hold deferred computation

here we actually run them

Why non-zero is printed ?

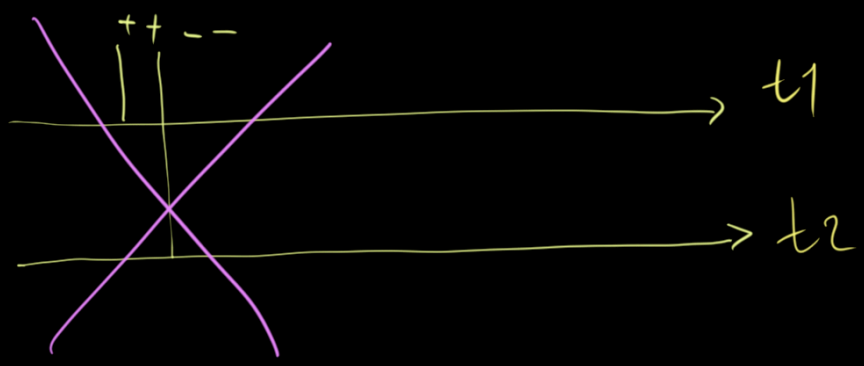



```
Runnable r1 = new Runnable() {
  @Override
  public void run() {
    for (int i=1; i <= 100_000; i++) box.i++;
  }
};

Runnable r2 = () -> {
  for (int i=1; i <= 100_000; i++) box.i--;
};
```

i++ - is not one line - but three lines

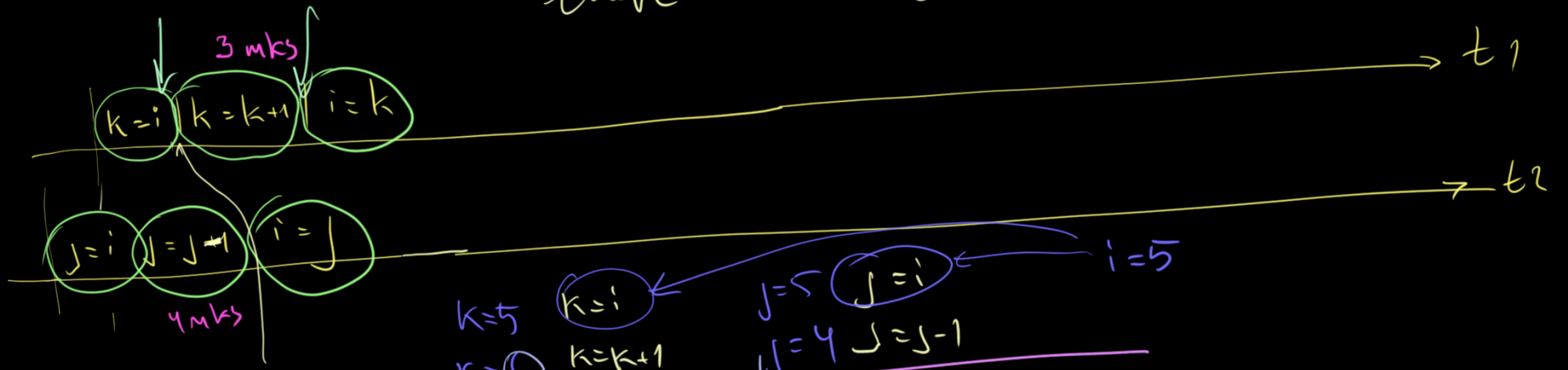
```
k=i
k=k+1
i=k
```

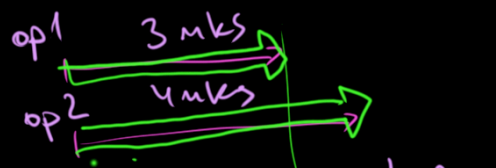
isn't atomic



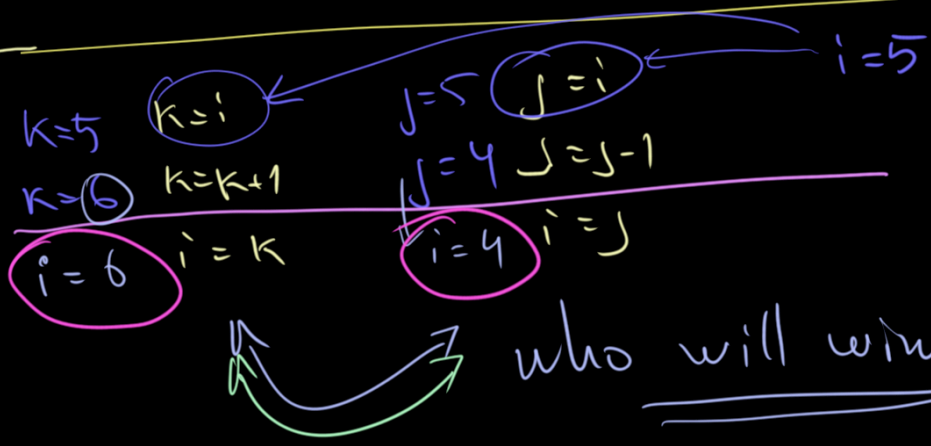
threads compete for the CPU time and all resources

there is no sync between threads



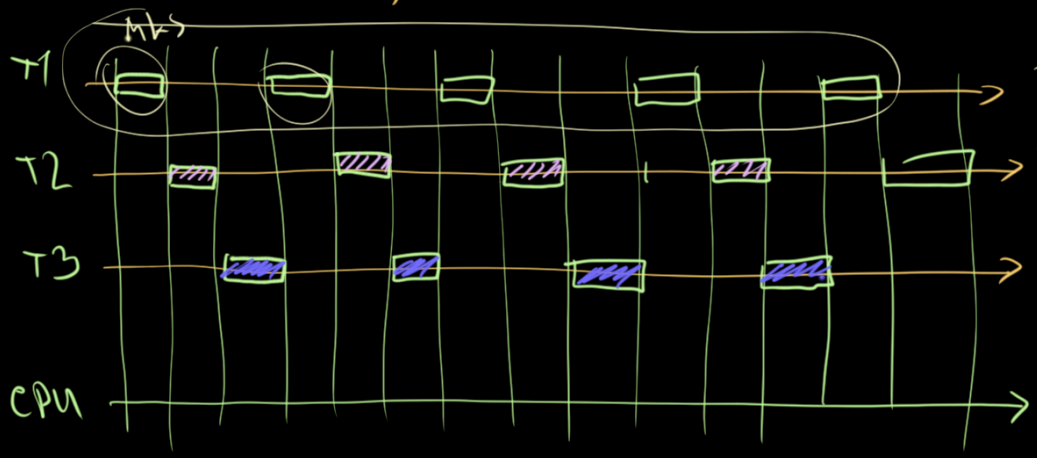


we can't guarantee precise parallelism



who will win? => nobody knows

Concurrency

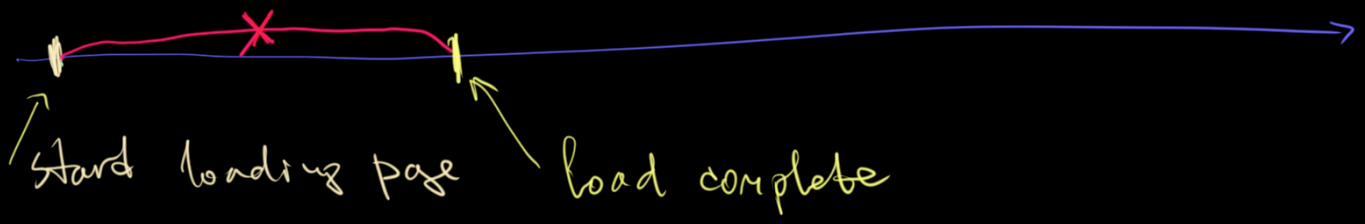


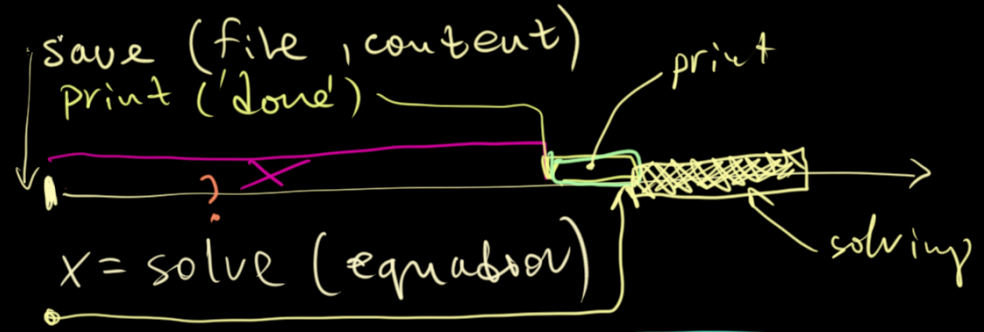
we need to share 1 CPU among 3 threads

one hand => the time is the same

on the other => we have kind of illusion

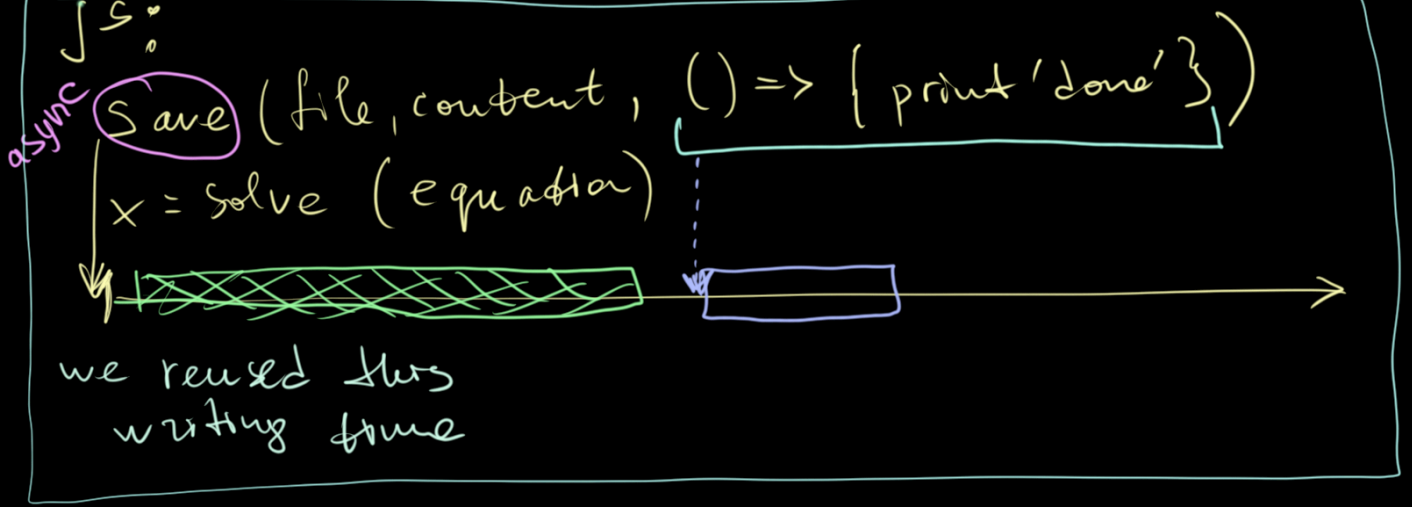
this CPU resource can be reused





C++ Python Java — code is running line by line

JS:



we reused this writing time

JS is better than Java?

JS is easier to write concurrent code

reading the value from DB. print result

solve equation

```java
// database reading
CompletableFuture<Integer> cfi =
    // don.t care about implementation
    CompletableFuture.completedFuture( value: 10);

cfi.thenApply(x -> {
  System.out.println(x);
  return x;
});

int x = solve();
```

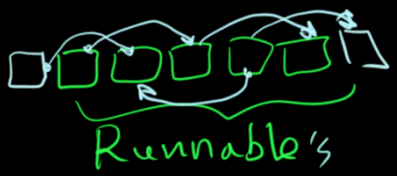
here we described computation
described chaining (.thenApply)

here we started solving

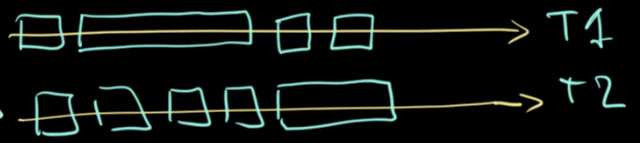
```
Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);

t1.start();
t2.start();
```

← manually create
← manually start them

hard complicated



ExecutorService Job's




```
Runnable r1 = () -> {
  for (int i = 1; i <= 100_000; i++) box.i++;
};

Runnable r2 = () -> {
  for (int i=1; i <= 100_000; i++) box.i--;
};

ExecutorService es = Executors.newFixedThreadPool( nThreads: 10);
es.submit(r1);
es.submit(r2);
```

does everything automatically

we just submit deferred computation to EXECUTOR described

- allocates threads
- release threads
- submits tasks to these threads

(1) represent pieces of your code in terms of Runnable

(2) just submit your pieces of code to ES.

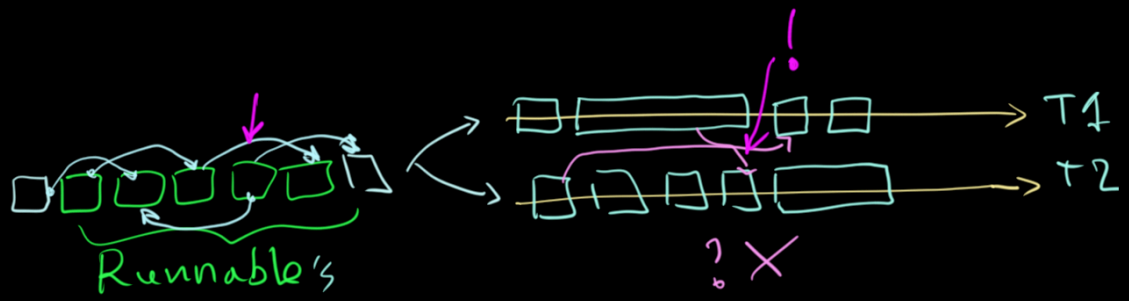
Runnable – Java

() -> {...} – JS

Runnable is just a Java's ~~$yntax~~ to represent callback

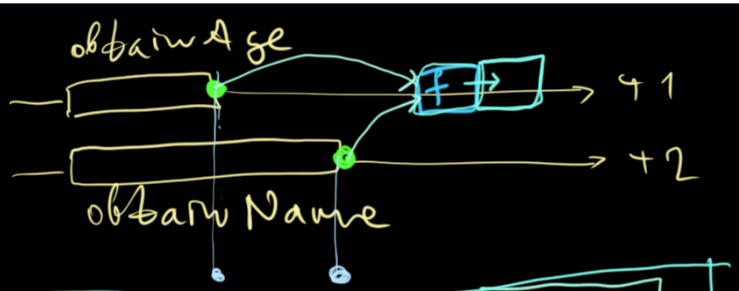
complicated -> shared state

```
Runnable r1 = () -> {
  for (int i = 1; i <= 100_000; i++) box.i++;
};
```



with Runnable is too hard to chain

1. shared state,
2. pass this shared state between computation stages



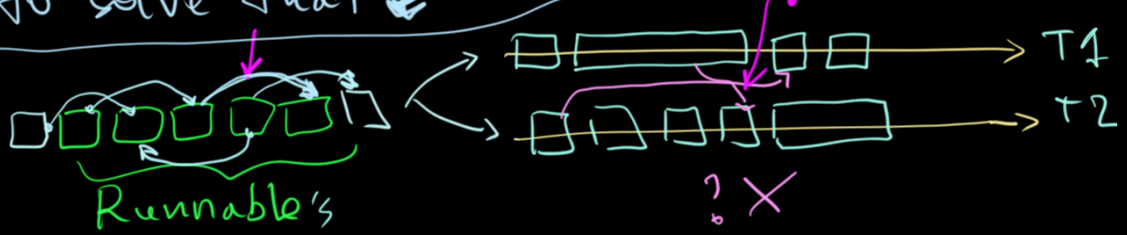

```java
@Data
static class Person {
  int age;
  String name;
}

// `age` is going to be taken from the one db
CompletableFuture<Integer> obtainAge() { throw

// `name` is going to be taken from the another
CompletableFuture<String> obtainName() { throw
```

```java
CompletableFuture<Integer> ageF = obtainAge();
CompletableFuture<String> nameF = obtainName();
```

task:
1. run ageF
2. run nameF
3. wait for complete ageF
4. wait for complete nameF
5. combine them synchronize
6. print out the result

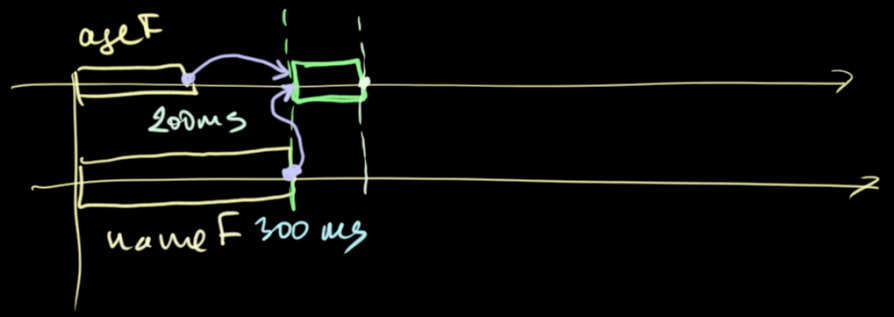
I don't know
- when they start
- when they finish
- how to synchronize them

Completable Future is the way to solve that



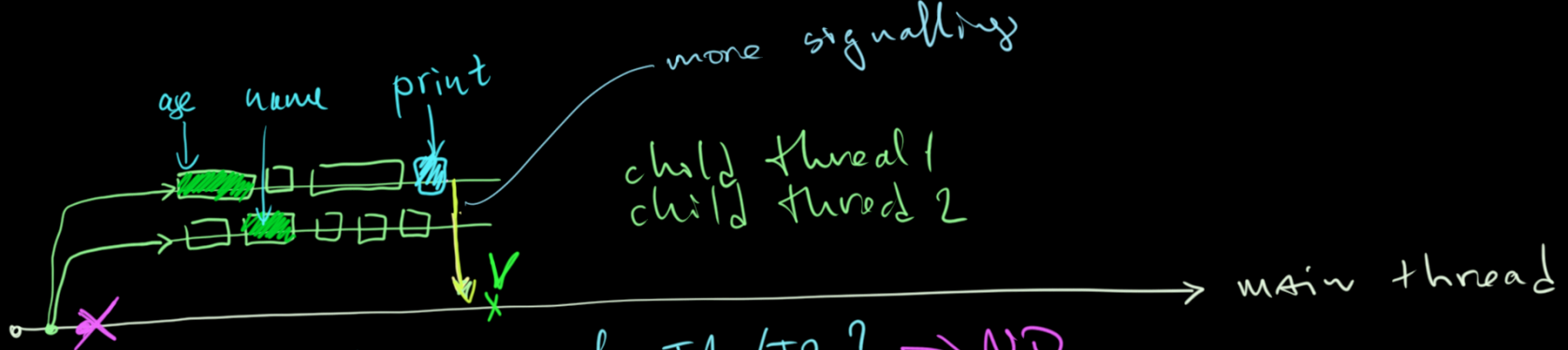
```
CompletableFuture<Integer> ageF = obtainAge();
CompletableFuture<String> nameF = obtainName();
CompletableFuture<Person> personF =
    ageF.thenCombineAsync(
        nameF,
        (age, name) -> new Person(age, name)
    );
```

← one server
← another DB



when ageF and nameF will be done, the function will be applied

.thenCombine

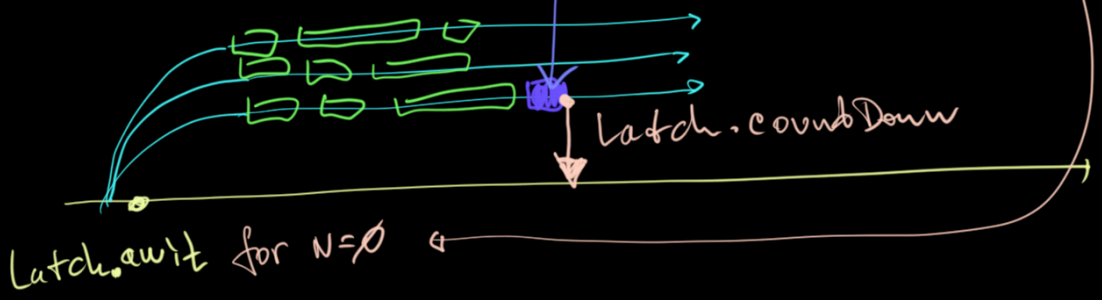


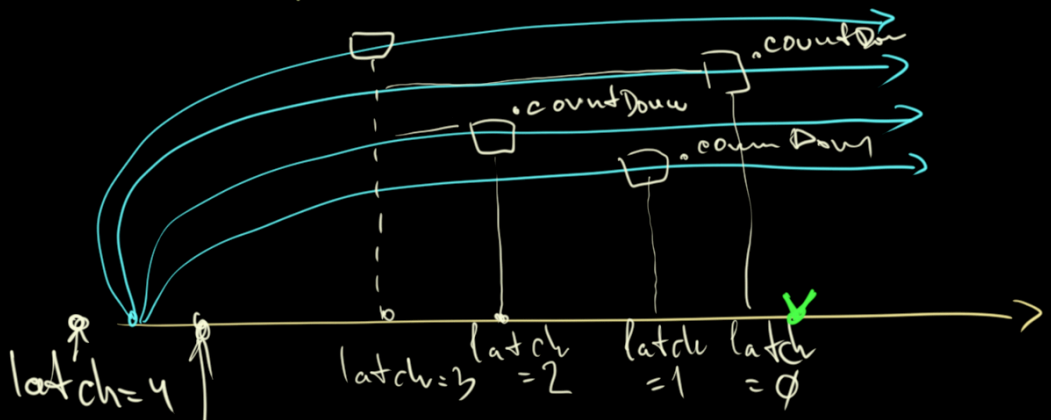
Is main thread aware of T1 / T2 ? → NO

```java
CountDownLatch latch = new CountDownLatch(1);

CompletableFuture<Integer> ageF = obtainAge();
CompletableFuture<String> nameF = obtainName();
CompletableFuture<Person> personF =
    ageF.thenCombineAsync(
        nameF,
        (age, name) -> new Person(age, name)
    );
CompletableFuture<Void> voidF = personF.thenAcceptAsync(p -> {
  System.out.println(p);
  latch.countDown();
});

latch.await();
```

setup the latch

start waiting latch.countDown





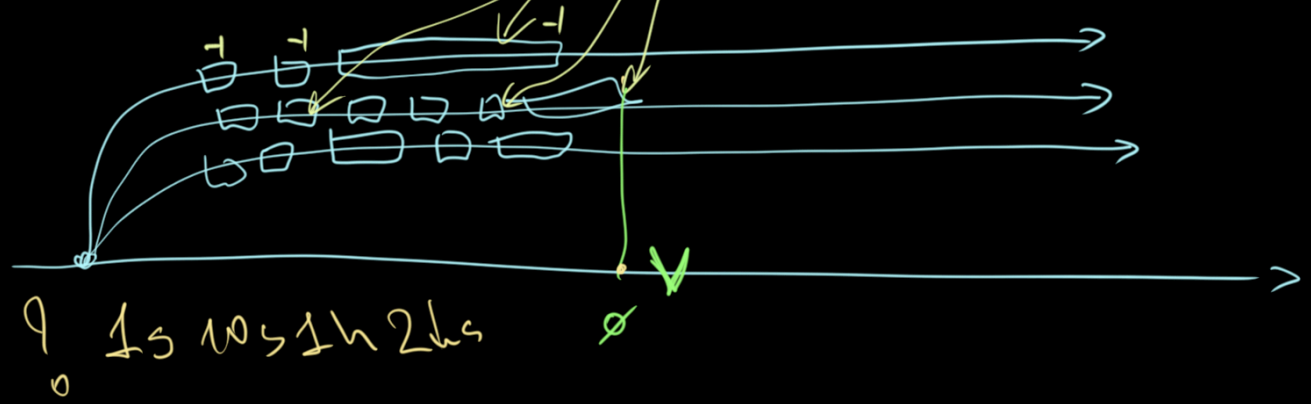
latch can be combined with everything

latch.await for zero

w/ latch we can guarantee everything we started is finished

```java
public long run_parallel_smart() throws InterruptedException {
    final int nCores = Runtime.getRuntime().availableProcessors();
    System.out.printf("Running on %d cores\n", nCores);
    ExecutorService es = Executors.newFixedThreadPool(nCores);

    List<String> fileNames = new AvailableFiles(args).get();
    CountDownLatch cdl =new CountDownLatch(fileNames.size());
    fileNames.forEach(s -> es.submit(new RunnableConverted(s, cdl)));
    cdl.await();
/       fileNames.forEach(s -> es.submit(new ConvertedFile(s)::convert));
    es.shutdown();
    return System.currentTimeMillis()-start;
}
```

```java
@Override
public void run() {
    new ConvertedFile(file).convert();
    cdl.countDown();
}
```



1. Thread / Runnable
2. mutual mutable state
3. atomic operation i++
4. executor service es.submit(r)
5. CompletableFuture
6. Synchronization Primitive : CountDownLatch.

describe your app in terms of callbacks