Coordination of threads
normally we can't manage the thread status directly. We could have thread that collaborate each other.
Locks can't be used for thread coordination. The solution is the barrier, that is active until all task reach that.
Wait: the thread leave the intrinsic lock acquired and wait until it not receive a notify of wake up.
Notify(): wakes up the first threads that called wait().
NotifyAll(): wakes up all threads that called wait inside synchronized block or method that use intrinsic lock.
The call of all methods must be done inside a synchronized or intrinsic lock method.
Wait must be called inside try catch with InterruptedException catch block. Wait must called on the object that call notify.
wait must called in the case of a condition is verified, USE the following implementation

```
1 while(!condition){
2         wait();
3 }
```

while is mandatory, because it can manage is condition is acceptable and put on wait status thread that don't wake up in that moment, because we use notifyAll that wake up all waiting threads on that object.
these methods must be coordinate on a common object, give correct visibility. These need to modify and verify flags.

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(timeout);
    // Perform action appropriate to condition
}

synchronized (obj) {
    <modify condition>
    obj.notify();
}
```

The wait operation forces the thread to release the lock before going to sleep. Release of the lock is mandatory to avoid deadlock!!!!
There is an alternative to use with reetrant locks.
Lock.newCondition() create a Condition object that has await() method to put in the same while of the previous solution.
Signal(), SignalAll() wake up the threads.
This solution allows to have more communication channels for threads.

Synchronizer
they are a concurrent building blocks useful for coordinating the flow of execution of concurrently executing threads.

- Thread ask to synchronizer to check internal state
- According to the internal state the synchronizer forces the thread to wait or let it pass
- The synchronizer provides methods for other threads to manipulate the state

Queues
They are FIFO, these queues haven't blocking system if it empty it will return null
Blocking queues : block when the queue is empty and when the queue is full.
Blocking queues are useful to implement the producer-consumer design pattern.

Types of BlockingQueues:

Queue	Data Structure	Capacity	Insertion Policy
LinkedBlockingQueue	Linked nodes	Optionally bounded	FIFO
ArrayBlockingQueue	Fixed size array	Bounded	FIFO
PriorityBlockingQueue	-	Unbounded	Comparable
DelayQueue	-	Unbounded	Delayed elements

Queues' methods:

Method Behaviour	Insert	Remove	Examine
Throw exception	add(e)	remove()	element()
Returns value	offer(e)	poll()	peek()
Time out	offer(e, time, timeunit)	poll(time, timeunit)	-
Block*	**put(e)**	**take()**	-

Synchronous Queue
it is a queue without capacity that allow pass a data between thread and if there is no data, it put thread in wait until data arrives.
Take and put methods resolve visibility, wait and notify ecc.
It allow to have alternation between thread.
Semaphore
it is useful to manage the access to an shared resource.
It release n virtual access permissions via acquire() and release().
Semaphore can manage accesses to resource with limit access available.

- **CountDownLatch**: threads have to wait until the latch reaches its terminal state. Similar to a door. Example of use: to simultaneously start the execution of threads.
- **CyclicBarrier**: similar to the CountDownLatch, but it is possible to reset it for reuse.
- **Exchanger**: allows threads to exchange objects at a rendez-vous. Blocks until another thread calls exchange(). Useful for sharing data buffers. Similar behaviour of the SynchronousQueue.
- **Phaser**: similar functionality to CountDownLatch and CyclicBarrier, but more flexible. If there's the need to wait hat a thread completes some work before continuing with other operations, the phaser is an ideal solution.

An alternative to yield is priority of threads
default priority is 5.

```
1  Thread aThread = Thread. currentThread();
2  int currentPriority = aThread.getPriority();
3  aThread.setPriority(currentPriority + 1);
```

the effect of priority can differ depending on the execution platform.

Stopping threads

A Java application only completes when all its (non-deamon) threads have completed.
There is a method interrupt() it assign true to a boolean flag called interrupt status.
We can have the value using interrupted() or isInterrupted(). Thread.interruputed() is a static method. Returns the value of the flag for the thread that is active at that moment(current thread) and clears the flag (set the value to 'false'). At the practical level, is used by the running thread to understand if an interruption has been requested and to stop the execution if needed.

Liveness

There are 3 types of problem

- DeadLock
- Starvation
- LiveLock

These problems don't permit to program to go forward.
Thread safety has 2 big categories:

- Safety: nothing bad ever happens
- Liveness: at least something good happens

Liveness problems can't be predicted, and they are difficult to reproduce.
Deadlock: Deadlock occurs when a thread locks a lock forever-→ The solution is **maintain lock in the same order**..
if we work with one lock deadlock will not appear, performance problems in this case.
Lock-ordering are always fixable. Lock in another lock is a bad practice.
Starvation: Starvation occurs when a thread execution is hindered(ostacolato). It could be generated by a bad use of thread priorities, priority inversion.

Livelock

it is different from deadlock, livelock occurs when a thread is alive but is not able to progress, because it failed always the same operation. It is similar to starvation.

Synchronization problem

Some of these problems are famous:

- Dining Philosophers Problem
- Sleeping Barber's Problem
- Producer-Consumer Problem
- Reader-Writer Problem

Dining Philosophers Problem

the solution to deadlock here is change the acquisition order.
The robust solution consists in:
A possible improvement is to release the lock before eating, but with both forks already taken.
This solution seems efficient, but only shifts the problem: the other philosophers end up fighting for the remaining forks, with the risk of running again into a liveness problem.
This doesn't solve starvation.
Solution:

- if the philosopher is able to take both forks, then he'll eat.
- Otherwise, he have to signal that he's hungry and wait.
- He will be awakened by his neighbor when releasing the forks.

Producer and Consumer

we have some threads that produce something and the final product will be consumed by other threads.
The product will be put in a buffer. Solution in Java, **BlockingQueue,** non robust solutions could cause deadlock.

The Barber problem

it is a problem that needs lock and conditions.
It possible that someone enter while barber is cutting hairs and the new one enter in the waiting room, but when he is going in the waiting room barber finishes to cut and start sleep, new customer will be locked in the waiting room, because no new customer awake barber.
Lightweight-tasks are work units (operations + data) that execute in a "deterministic way"
The running time of lightweight-tasks is limited and usually short
The occupation of resources (such as memory), is also normally subject to constraints.
Executions behind are done by thread, but we work with a **thread-pool**
that execute lightweight-tasks. With lightweight-tasks overheads are few.
To implement a lightweight we have to implement Runnable or Callable.
A Callable is similar to a Runnable, but runnable has some defects that callable resolved.

Callable accept generics that call method return. Runnable has run void, not useful.
Create a thread pool

```
1 final ExecutorService executorService = Executors.newFixedThreadPool(10);
2 final List<Future<double[][]>> futures = new ArrayList<>();
3 ...futures.add(executorService.submit(new CallableObject());
4 ....ris= future.get();
```

Task boundaries
Task have to be independent of each other, no link between states and task operations.
The best way is task work in stack-confinement, avoid shared and mutable states of the application.
Important: if it is strictly required to access a shared and mutable state of the program from the lightweight-tasks, synchronization tools have to be used!
The executor framework guarantees safe publication only when sending lightweight-tasks and when retrieving results, NOT for all operations internally performed by the tasks

- The following pre-configured thread pools are available:
  - *newFixedThreadPool* (LinkedBlockingQueueue): creates threads until a maximum is reached, then try to keep the pool size constant.
  - *newCachedThreadPool* (SynchronousQueue): adapts dynamically on the amount of requests. Has no upper limit.
  - *newSingleThreadExecutor* (LinkedBlockingQueueue): processes tasks sequentially.
  - *newScheduledThreadPool* (DelayQueue): supports the execution of delayed and periodic tasks (similar to a Timer).

Future<T>
it is returned from submit method, it represent the result in the future. It allows to “anchor” the result for the future retrieving.
Future is an interface, the concrete class is FutureTask, with get I can try to retrieve result, if it is not ready put in wait status.
Without ExecutorService submit method doesn’t exist, all Java executors implements ExecutorService.

The submission of a Runnable or Callable to an executor is a safe-publication, also the return of a result from a Future is a safe-publication.
CompletionService with poll retrieves and removes the future the next completed task, it isn't blocking, null if there aren’t present.
With take method use loop to get the right number of task.
Thread poll on task can delete using cancel method, we can call from future through cancel().
We can develop custom thread-pool extending ThreadPoolExecutor or ScheduledThreadPoolExecutor.
Fork join
it is a framework that use and executor, behind, it is design for problems that can be decomposed recursively.
It implements a work-stealing approach in order to have good performance: Threads in the pool that have completed the assigned tasks are allowed to steal the tasks of other busy threads.
Fork-join model is widely used in parallel programming.
The **ForkJoinPool** executes **ForkJoinTask**, a light-weight version of a Future.
ForkJoinTask has two specializations: **RecursiveTask** (is allowed to return a result) and **RecursiveAction**.
A ForkJoinTask starts its execution when it is sent to a ForkJoinPool.
The task must extends RecursiveTask<...>
Latency+ProceccingTime=ResponseTime
Latency=ProceccingTime
Speed-up
S=SerialTime/ParallelTime
Amdhal: SpeedUp = 1/(S+((1-S)/n))
Gustafson: Scaled SpeedUp = n + (1-n) * s
n = num. cores, S = serial part
ParallelStreams
We can use parallel-stream also for not-thread-safe collections, with the constraint of don’t modify data in the collection, use immutable objects.
Strongly recommended to avoid the acquisition locks in the lambda functions.
With parallel streams, it's important to avoid any type of modification on the source collection during the execution of the higher-order functions. It is therefore strongly advised to apply a programming approach inspired on functional programming.
In addition, it is strongly recommended to avoid the acquisition of locks in the code of the lambda functions
Futures are limited because If there’s the need to perform additional operations on the result provided by a future, it is required to retrieve the result with get(), then submit additional tasks for the second part of the work.
Completable Futures
they implement Future interface as well as the CompletionStage interface.
Completable futures use listener, the are based on event-based callbacks.
Completable fetuwre can be used like "streams" and allow to program a chain of asynchronous operations.

```
1 CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> "Hello");
2 CompletableFuture<String> future = cf.thenApply(s -> s + " World");
```

thenApply() is excecuted after the Async execution. (Java doc for other functions). SuplyAsync or runAsync accept the Supplier or Runnable parameter and/or the executor that execute the task, if it is not passed it will be used java default pool, the ForkJoinPool.commonPool().

Race condition

Definition:

A race-condition is present when a program reads the value of a variable/field and needs then to perform an operation that depends on that value (e.g. check-then-act or read-modify-write).

- **Check then act:** it's a problem that concern decision instructions, they took decisions based on possibly old values, example if condition.
- **Read modify write**: it's a problem that concern edit instructions, it updated date based on not updated data, example var++
- ***Compound actions:*** *they are sequences of instructions that must be execute atomically to maintain the integrity, so be* ***thread-safe****.*

Immutable collections

they are unmodificable collections are similar to immutable objects (but are not completely immutable).
It provide read-only views of collection objects.
This collections don't guarantee correct memory visibility. If modified the read-only views should be safely republished or outdated values have to be tolerated.
To create an immutable collection use Collection class static methods, example: Collections.unmodifiableList(lst);
Unmodifiable collections are collection useful for read only and for confinement of a collection in one writer many reader approach.

Client side locking

```
synchronized (myList) {
        Interator<Integer> it=myList.iterator();
        while(it.hasNext())
                doSomething(it.next)
}
```

volatile variable will never be cached and all writes and reads will be done to and from the main memory.

Philosophers solution

```
void putForks(int i) {
    lock.lock();
    release_fork(i);
    release_fork((i + 1) % N);
    state[i] = THINKING; // philosopher has finished eating
    checkAndResume(LEFT); // resume left neighbor
    checkAndResume(RIGHT); // resume right neighbor
    lock.unlock();
}

void checkAndResume(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING
                           && state[RIGHT] != EATING) {
        state[i] = EATING;
        take_fork(i);
        take_fork((i + 1) % N);
        lock.notifyAll();
    }
}
```

Phaser important methods

construtor put how many parties partecipate to phaser, register increment this.
register()
arrive()//open the barrier
arriveAndAwaitAdvance();
arriveAndDeregister();

REMEBER JOIN AT THE END OF THE PROGRAM

Today, general-purpose operating systems implement preemptive scheduling as a solution for time-sharing.

In an operating system there are 3 main categories of schedulers:

- Long-term scheduler: responsible to select which process is accepted for execution and with which priority.
- Medium-term scheduler: temporarily removes processes from the main-memory and saves them to disk ("swap out") – in particular processes that are not running.
- Short-term scheduler (CPU scheduler): select which process/thread is allowed to execute. If it is a preemptive scheduler, it is activated at least for each time-slice or for interrupts, synchronization primitives (locks, waits, ...), OS calls or other signals.

Dispatcher: is the module that takes care of assigning the CPU control to the processes/threads. It is responsible for context-switches.
Possible scheduling disciplines:

- First-come, first-served (FIFO): processes/threads are scheduled in the order of arrival.
- Round-robin: a fixed amount of time is assigned to each process/thread. Execution is performed in cycle.
- Fixed-priority preemptive: the scheduler organizes the execution of the processes/threads according to their priorities.