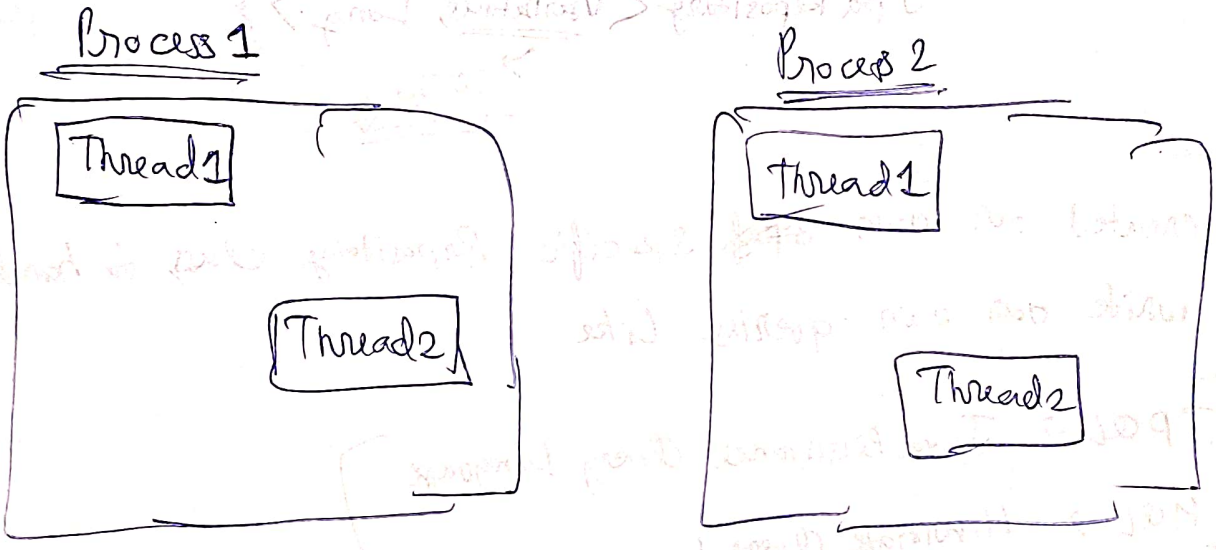
Multithreading & Concurrency. Video 29.

Process → Its an instance of a program.

Program → (Blueprint) (has logic and instructions)

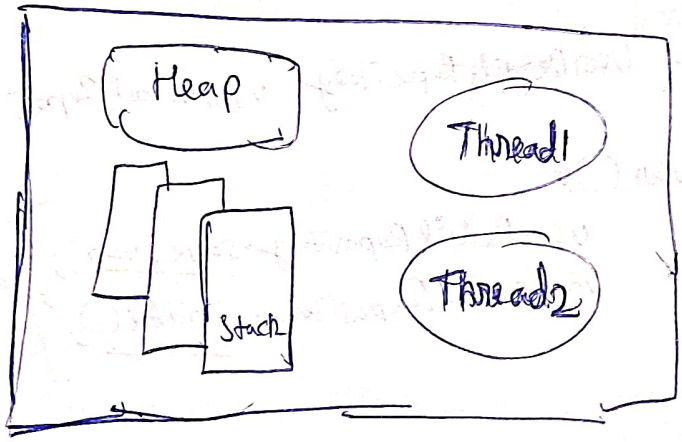
Process → running this Blueprint.

~~Thread~~ A process can have multiple threads.



Each process has its own resources.

Process →



You have a file Test.java

[javac Test.java] → generates bytecode to be executed by JVM.

java Test ⟹ (JVM starts the new process).

You will see processID (PID) created.

Q) How to decide, how much memory to allocate to the process.

A) In JVM args we provide arguments.

java -Xms256m -Xmx2g

-Xms256m → (initial size of heap. 256MB.)

-Xmx2g → (max heap size can go to 2GB.)

Thread

A process can have multiple threads.

when process created it only has main thread.

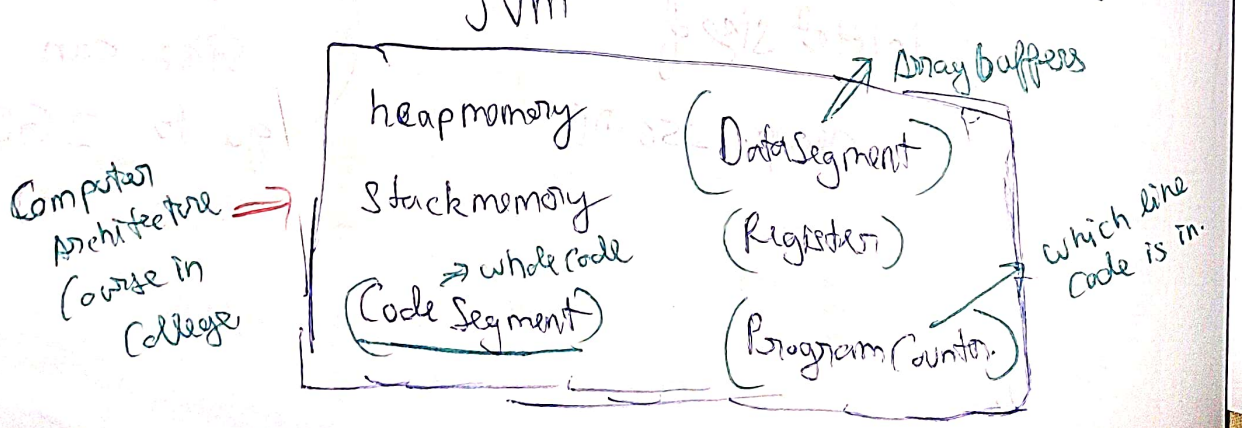
Thread.currentThread().getName() → name of current thread.

(How it actually happens.)?

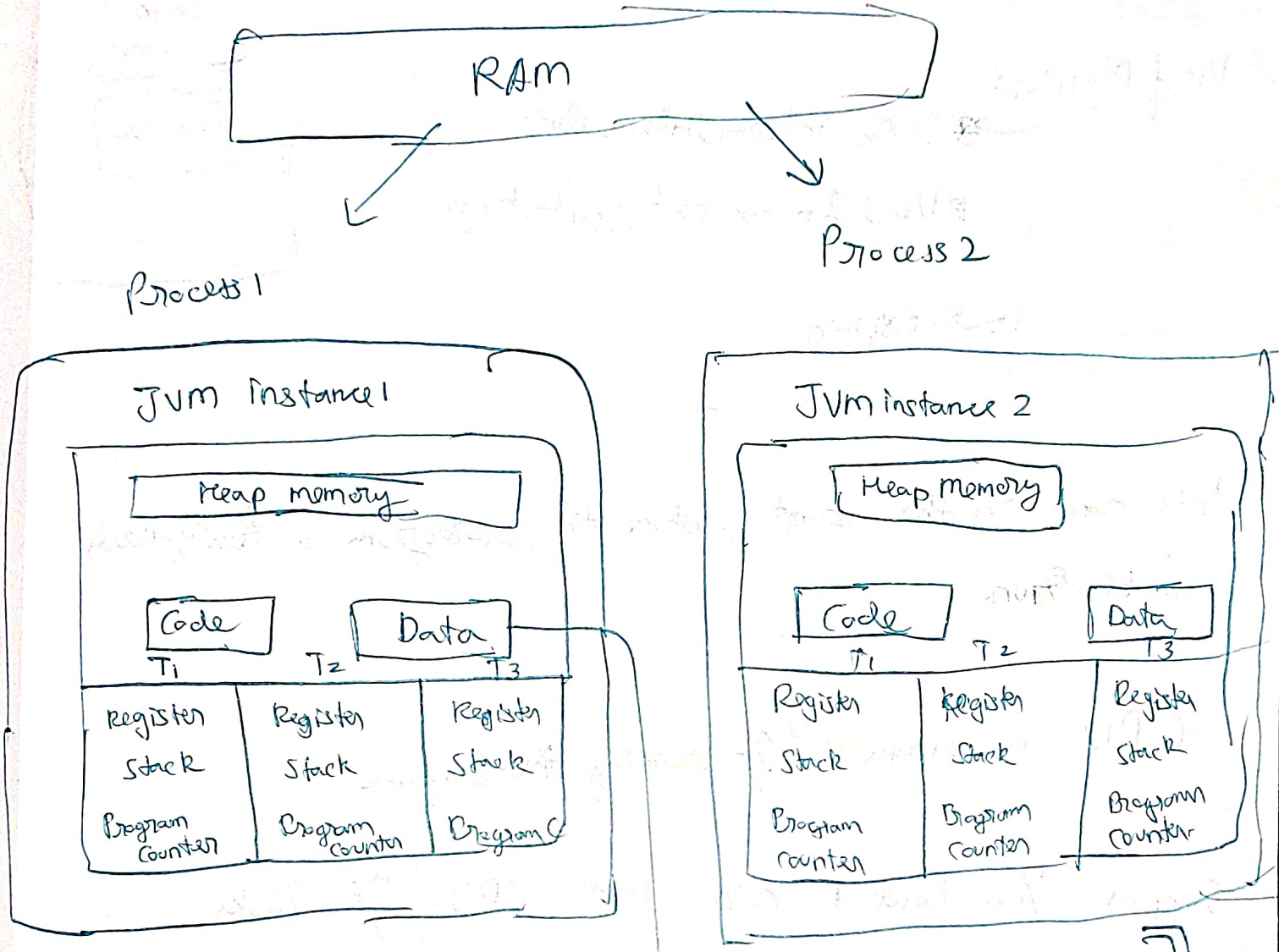
when You do (java Test) ⇒ JVM Creates a new process and allocates it memory.

(-Xms 256MB -Xmx 2G.) → decides how much memory JVM gets.

Since its a machine code that has to be run.



Now You decided to divide the task among 3 threads.



Code { has values which are common to all threads

Code Segment 1 [Thread 1 →

Code Segment 2 [Thread 2

Code segment 3 [Thread 3

}

So Each thread needs its Program Counter, Register, Stack to execute its part of code.

We saw that each thread has its own register →

Use of Register:

→ Store intermediate values

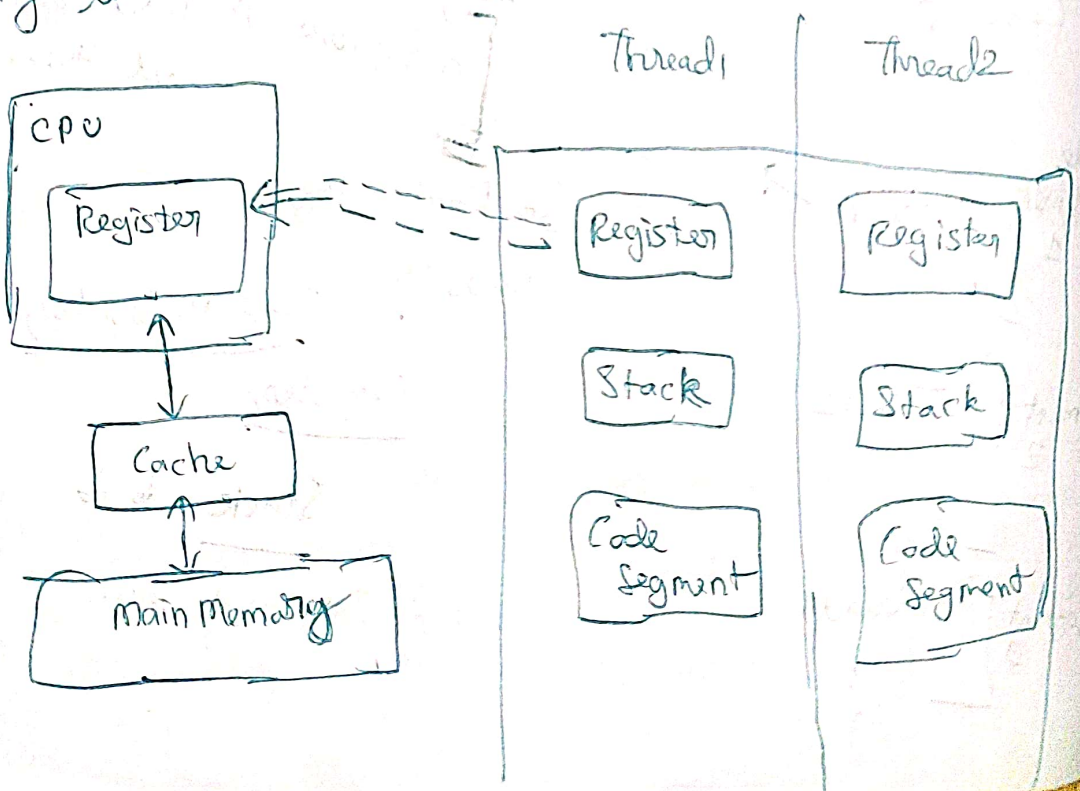
Used for context switching

~~Will Explain~~

Lets come to the part, where the program actually needs to be run.

CPU is responsible for running the program.

Suppose You have 1 core in the CPU. It looks something like:



You know about OS Scheduler right?

When OS gives a particular Thread, a chance to execute in the CPU.

It takes the program Counter, Code Segment and everything

Now ~~Sent~~ After sometime (Context Switch).

The State (programCounter, intermediate values, State in b/w.). are ~~stored~~ in the (Register of the Thread.).

Another thread gets the CPU.

(Video of Functional Interface)

```
@FunctionalInterface
public interface Bird{
    void canFly(String val);
}
```

Interface with only 1 Abstract method. is Functional Interface

Imp (offtopic) :

By default public abstract, gets added to the signature

void canFly() → public abstract void canFly()

Default.

Case 1

@FunctionalInterface → Will give error, Cannot have the annotation with 2 abstract methods.
public interface Bird {

    void canFly();
    void canSwim();

}

Case 2 (Valid Scenario).

@FunctionalInterface
public interface Bird {

    void canFly() → 1 Abstract method.

    default void ...() { → Default method. Has implementation
        // some impl
    }

    static void canEat() { → Static method.
        // some impl
    }

}

New thing I Learnt to-day.

We know Object Class in java.

By Default every class / interface extends the Object Class.

@FunctionalInterface
public interface Bird{ ⟹ Valid

String toString(); → method coming from Object Class.

void canFly(); → Abstract method.

}

⟹ We should be good.

(Lambda Expression)

Easier, cheeky & crisp looking code - To increase swag.

Lets say you want to implement the @ Bird Interface

① Normal Implements.

② Anonymous Class

(we will learn about it in depth. For now just check the syntax

```
Bird newObj = new Bird(){
    @Override
    public void canFly(){
        // some imple
    }
};
newObj.canFly();
```

3rd way Lambda expression.

```
Bird birdObj = (String value) -> {
        sout("some implementation" + value);
};

birdObj.canFly("Lambda");
```

For now ignore the syntax, we will logically understand, what is going on.

Types of Functional Interface

① Consumer

```
@FunctionalInterface
public interface Consumer<T>{
    void accept(T t);
}
```

Accept single input parameter and return no result

```
Consumer<Integer> logObj = (Integer val) -> {
        sout("value = " + Val);
};

logObj.accept(10);
```

② Supplier.

Accepts no input but returning result

```
@FunctionalInterface
public interface Supplier<T>{

    T get();

}
```

```
Supplier<Integer> sp = () -> {

        return Integer.MAX_INT;
    };

sout(sp.get());
```

③. Function.

Accepts a value and returns a value.

```
@FunctionalInterface
public interface Function<T,R>{

    R apply(T t);

}
```

```
Function<String, Integer> lenfunction = (String s) -> {

        return s.length();
    }.

lenfunction.apply("Function");
```

4) Predicate.

→ Accepts a value, returning boolean.

```
@FunctionalInterface
public interface Predicate<T>{

    boolean test(T t);

}.
```

```
Predicate<Integer> isEven = (Integer val) ->{

    if (val%2==0){

        return true;
    }

    return false;

}.
```

```
isEven.test(10);
```

Unique Case of ~~Extend~~ inheritance in Functional Interface.

```
interface Bird{

    void canFly();

}
```

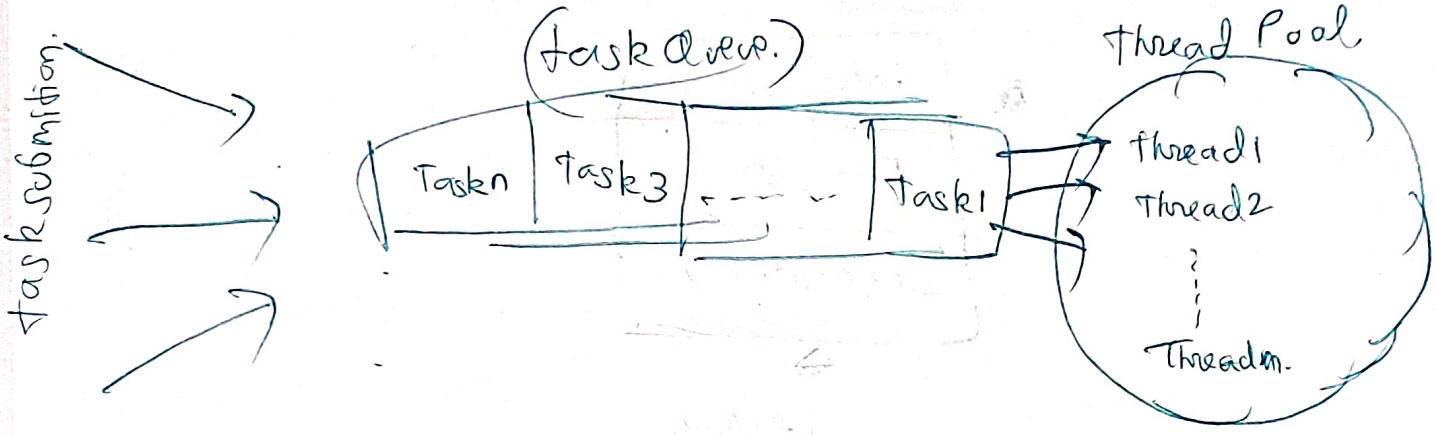
```
@FunctionalInterface
interface NewBird{

    void canFly();

}
```

[This is getting overridden so its 1 abstract method only]

Video 34. Thread Pools in Java | ThreadPool Executor Framework.



whenever a thread in the pool is free, it picks up a task from the queue

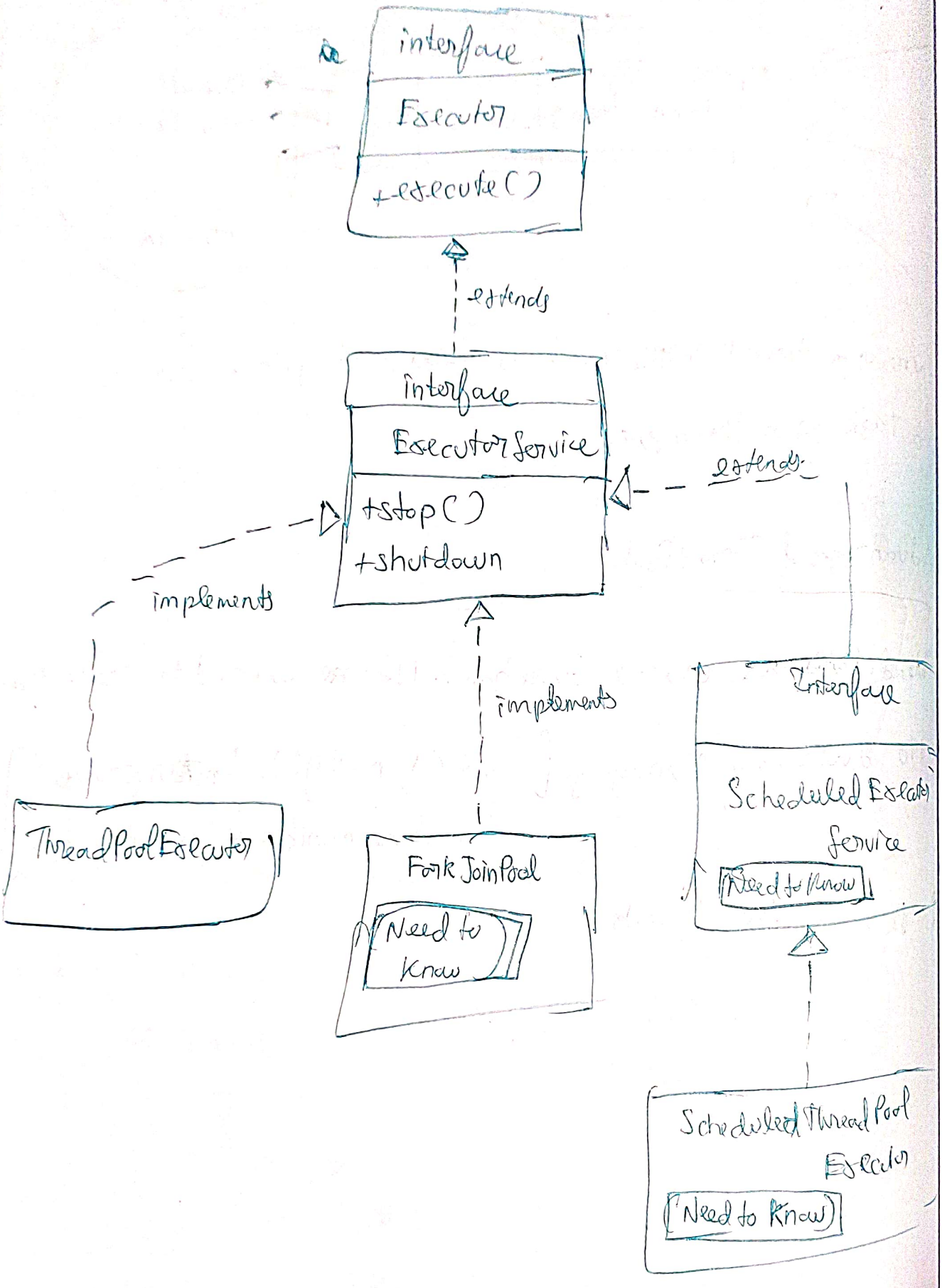
Advantages of ThreadPool.

→ Threads are created beforehand. No time wasted to create them.

→ No overhead of Managing. [wait(), notify(). Inter process thread communication is taken care].

→ Reusing the threads.

Hierarchy of the Concurrent Package



Arguments of ThreadPoolExecutor.

* corePoolSize → no of threads to initialize with, even if idle.
* maximumPoolSize → max no of threads it can have.
* keepAliveTime → time till which idle thread is allowed to live.
* unit → [milliseconds, Hours, Seconds].
* BlockingQueue<Runnable> workQueue,
* ThreadFactory,
* RejectedExecutionHandler.

* If queue is full & all threads are busy then new thread is created. One by one new thread is created till we reach maximumPoolSize.

* BlockingQueue →
  - BoundedQueue [ArrayBlockingQueue] (FIXED SIZE)
  - UnBoundedQueue [LinkedBlockingQueue]. (NO FIXED SIZE)

* ThreadFactory → It allows us to customize creation of threads, like [name, setDaemon(), givePriority]

* RejectedExecutionHandler →

→ new ThreadPoolExecutor.AbortPolicy
(throws RejectedExecutionException)

→ new ThreadPoolExecutor.CallerRunsPolicy
(Execute the Rejected task in
caller thread. main thread
basically

→ new ThreadPoolExecutor.DiscardPolicy
(Simply Reject the task).

→ new ThreadPoolExecutor.DiscardOldestPolicy
[Discards the oldest task in the
queue, to accomodate new
task