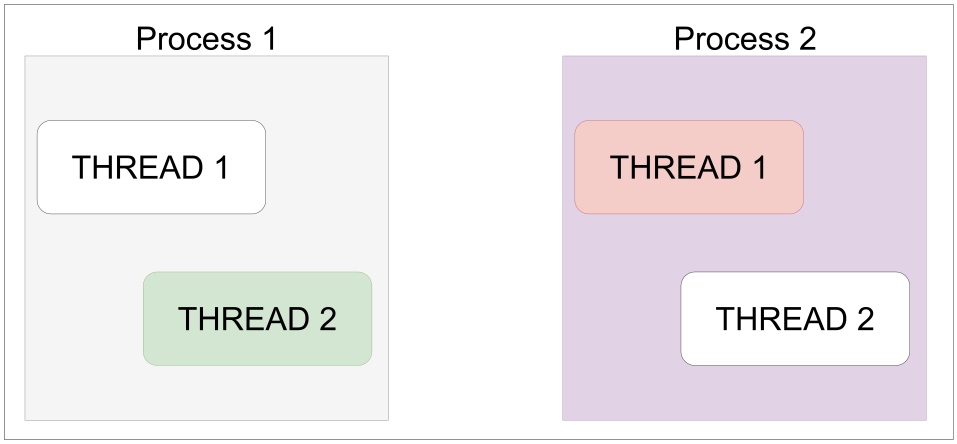
1. **Introduction Of Multithreading**.

- Before we understand what is Multithreading, let's first understand Thread and Process.



2. **Process.**

- Process is an instance of a program that is getting executed.
- It has it's own resources like memory, thread etc,
- OS allocate these resources to process when it's created.
- Compilation [javac Test.java] → Generates Bytecode that can be executed by JVM.

→ Excution [java Test]: at this point , **JVM** starts the new Process, here Test is the class which has "public static void main(String[] args)" method.

3. **How much memory does process get's?**

- while creating the process "java MainClassName" command, a new JVM instances will get's created much heap memory need to be allocated.

java -Xms256m Xmx2g MainClassName
→Xms <size>: - this will set the initial heap size, above, i allocated 256MB
→Xmx<size>: - Max heap size can get, above, i allocated 2GB, if tries to allocate more memory, "OutOfMemory"

4. **Thread.**

- Thred is also known as lightweight process
- or Smallest sequence of instruction that are executed by CPU independently.
- And 1 process can have multiple threads.
- JVM make process for bytecode→macinecode and within that process it creates one initial thread which is **main** thread.
- When the process is created, it start with 1 thread and that inintial thread know as "main thread" and
- form **main** thread that we can create multiple threads to perform the task concurrently.

```
package Multithreading;
```

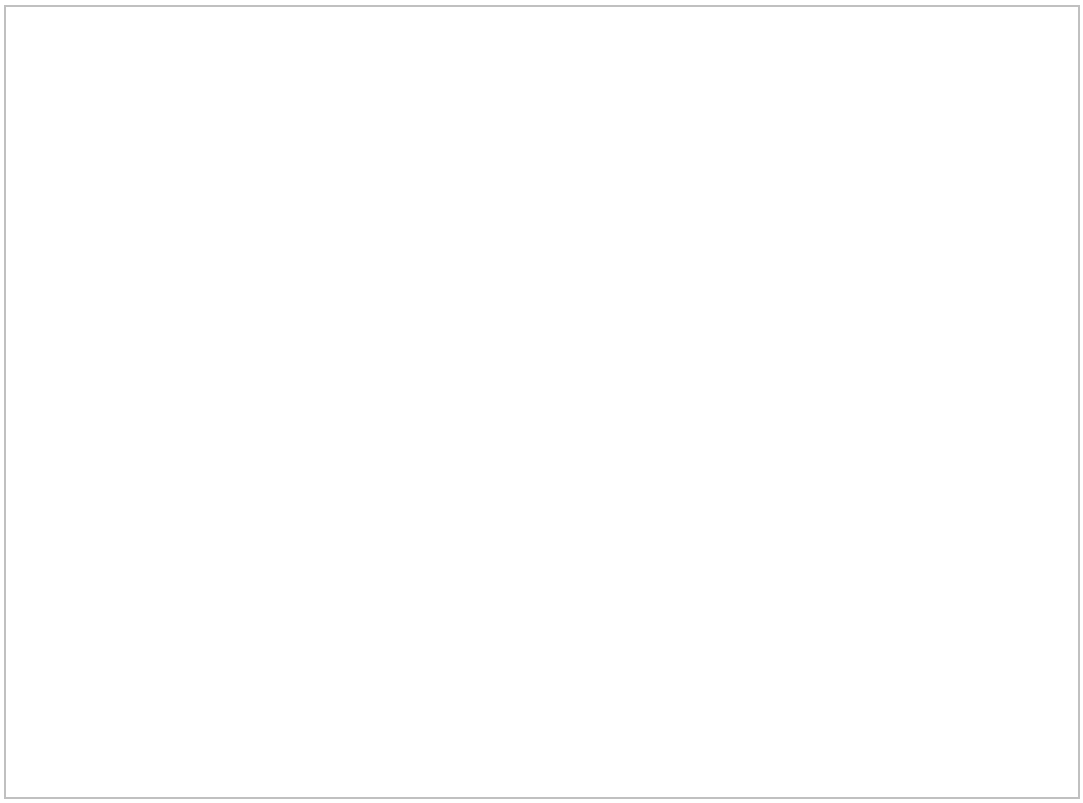
```
public class MultiThreadingLearning {
    public static void main(String[] args){
        System.out.println("Thread Name: "+Thread.currentThread().getName());
    }
}
output:
Thread Name: main

Process finished with exit code 0
```

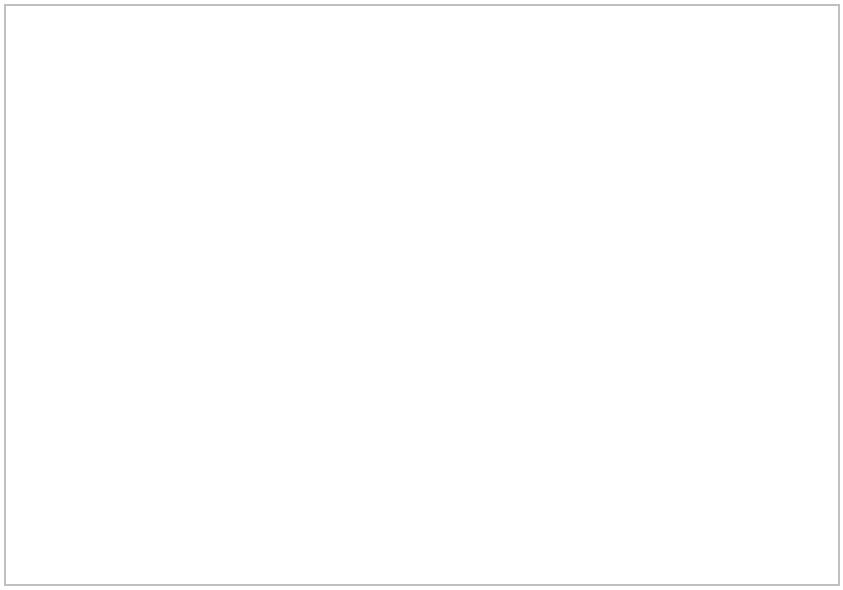
5. **Let's Deep dive into Process and Treads.**

Note:

JVM Consist of heap, stack, code segment, data segment, Register , Program Counter.

- When program started and after creation of process a new JVM instance created allocated to this process.
- When we run java program how JVM internally gonna handle let's try to understand.
- Phycical Memoary 10 GB [total JVM heap Memory.
- At the time of process execution JVM will decide how much memory required for JVM instances and we can also do programatically.
- java -Xms256m Xmx2g MainClassName
- →Xms <size>: - this will set the initial heap size, above, i allocated 256MB
- →Xmx<size>: - Max heap size can get, above, i allocated 2GB, if tries to allocate more memory, "OutOfMemory"





6. **Code Segment:-**
   - javac Main.java

   →Byte Code

   →java Main

   →process → JVM INstances → Interperated / JIT Compiler → convert it to machine code
   - Contains the compiled BYTECODE [i.e machine code] of the java prgram.
   - It's read only.
   - All threads within the same process. share the same code segment.

7. Data Segment.

- Contains the GLOBAL AND STATIC variables.
- All threads within the same process, share the same data segment .
- Threads can read and modify the same data.
- Synchronization is required multiple between multiple thread.

8. Heap

- Objects created at runtime using "new" keyword are allocated to heap.
- Heap is shared among all threads of the same process. [BUT NOT WITHIN THE PROCESS]

→let say in process 1 , X8000 heap memory pointing , to some location in physical memory , same X8000 heap memory point to different location for Process 2]

- Threads Can read and modify the heap data.
- Synchronization is required multiple between multiple thread.

9. Stack:

- Each Thread has it's own STACK.
- It manages, method calls, local variables.

10. Regiter

- When JIT [Just In Time] compiles converts the BYTECODE into native machine code, its uses register to optimized the generated machine code.
- some time we have to re-suffle the instructions and there are some immediate value, thouse are store in register.
- Some intermediate code has to stored as per program instruction.
- Also helps in **CONTEXT Switching.**
- Each thread has it's own register.

11. Counter.

- Also known as Program Counter, it points to the instruction which is getting executed.
- Increments it's counter after successfully execution of the instruction.

ALL these are Manage By JVM

12. Let's Understand the Whole flow:-

- Let's we have Main.java
- in that code we haveMain thread and within main thread we have two more threads T1 and T2 with it's own code base.
- javac Main.java

→BYTECODE

- java Main

→ Process Will created

→JVM Instances allocated with 1GB size.

- This JVM start interperating or JIT compiling BYTECODE to MACHINE CODES
- Including Main thrad and t1 , t2 we have total of 3 threads created.
- each thread assign with **counter**, **register**, and **stack** which is loacal to that thread only.
- what ever the machine code [CPU UNDERSTAND] created by **JIT it will store into code segment.**
- let's T1 thread executing it's codes base, so T1 **counter** start pointing to **code address** in **code segment machine code** hey this has be start executing.
- and same process will goes for remaining code as well.
- program counter has address to the code segment where code execution has to be start.
- what ever the induvidual thread point the code address in code segment. that code block and data segment as well, will assign to thread register which is very much similar to cpu register, and that thread register code segment and data segment, will assign and execute by the **CPU and it's register**.
- and after that **OS** comes into the picture that handle the **CPU** execution.
- let's we have only one core CPU if t1 will execute by cpu and it take's 1 second then other threads needs to wait until t1 execution has to be completed.
- All result generated by **CPU** after t1 threads execution it will store into **thread register** from **CPU register**.
- After t1 execution co CPU has to do CONTEXT Switching for other threads execution SO, CPU Load It's Register from t2 thread register and start execute them .
- After Context switching CPU has start executing other threads let's t2.
- If we have multiple core CPU then they can execute multiple therads in parllel.

13. **Definition of Multithreading**.
    - Allow a program to perform multiple task at the same time.
    - Multiple threads share the same rsources such as memory space but still can perfom task independently.

14. **Benefit's and Challenges of MultiThreading.**
    - Immproved performace by task parallelism
    - Responsiveness
    - Resource sharing

Challenges:

- Concurrency issue like deadlock, data inconsistency etc.
- Synchronized overhead
- testing and debugging is difficult.

15. **MultiTasking vs MultiThreading.**

- Process 1 and Process 2 is multitasking

→In multitasking the do not shared resorces.

- Inside particular task we can execute multiple thread in parallel which is called multi threading.

→ In multithreading they do share the resources and they can execute indepent.

16. **Thread Creation Ways.**
    - implementing 'Runnable' interface
    - Extending "Thread" class.

17. **Why We have two ways to create thread.**
    - class A extends B→ at this point you cannot extend Thread class.
    - class A extends B implements **Runnable**.

18. Runnable is a functional interface.
    - Runnable → run() which has on abstract methods.
    - Thread :- init(), run(), sleep(), start(), stop(), interrupt() etc..
    - Thread class implements Runnable interface.
    - MyClass Implements Runnable run() method, which is invoked by thread class.

Step 1: Create a Runnable Object.
    - Create Class that implements 'Runnable' interface.
    - Implement the 'run()' method to tell the task which thread has to do.

step2:
    - Create an instance of class that implement 'Runnable'
    - pass the Runnable object of the Thread Constructor.
    - Start the thread.

```
public Thread(Runnable task) {
    this(null, null, 0, task, 0, null);
}

public void start() {
    synchronized (this) {
        // zero status corresponds to state "NEW".
        if (holder.threadStatus != 0)
            throw new IllegalThreadStateException();
        start0();
```

```
        }
    }

    @Override
    public void run() {
        Runnable task = holder.task;
        if (task != null) {
            Object bindings = scopedValueBindings();
            runWith(bindings, task);
        }
    }

    @Hidden
    @ForceInline
    final void runWith(Object bindings, Runnable op) {
        ensureMaterializedForStackWalk(bindings);
        op.run();
        Reference.reachabilityFence(bindings);
    }
```

```
package Multithreading;

class MyClass implements Runnable{

    @Override
    public void run() {
        System.out.println("code executed by thread in runnable run method of MyClass:
"+Thread.currentThread().getName());
    }
}

public class CreatingThreadByImplementingRunnable {

    public static void main(String[] args){

        System.out.println("Going inside the main method thread:
"+Thread.currentThread().getName());
        MyClass myClassRunnableObj = new MyClass();
```

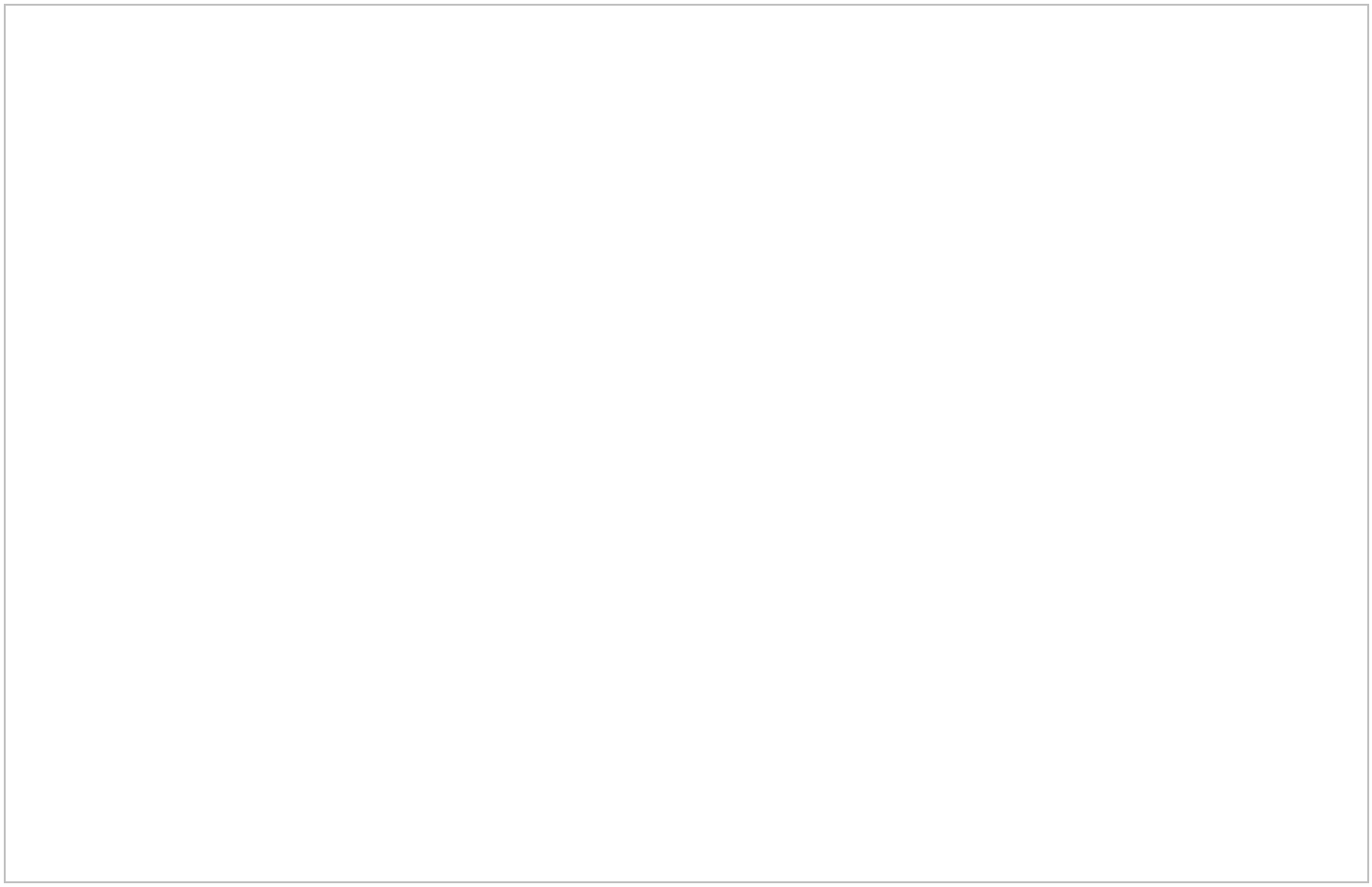
```
        Thread thread = new Thread(myClassRunnableObj);
        thread.start();
        System.out.println("Finish main method: "+Thread.currentThread().getName());
    }
}
output:
Going inside the main method thread: main
Finish main method: main
code executed by thread in runnable run method of MyClass: Thread-0

Process finished with exit code 0
```

19. **Creating thread by extending Thread class.**

```
package Multithreading;

class MyThreadClass extends Thread{
    @Override
    public void run(){
        System.out.println("code executed by thread in extending thread class:
"+Thread.currentThread().getName());
    }
}
public class CreatingThreadByExtendingThreadClass {

    public static void main(String[] args){
        System.out.println("Going inside the main method thread:
"+Thread.currentThread().getName());

        MyThreadClass myThreadClass = new MyThreadClass();
        myThreadClass.start();
        System.out.println("Finish main method: "+Thread.currentThread().getName());
    }
}
output:
Going inside the main method thread: main
Finish main method: main
code executed by thread in extending thread class: Thread-0

Process finished with exit code 0
```

In Industry they prefer Runnable Interface.

20. **Thread Life Cycle.**



Life Cycle State	**Description**
New	• Thread has been created but not started • Its just an Object in memory.
Runnable	• Thread is ready to run. • Waiting for CPU time.
Running	• When thread start executing it's code
Blocked	Different scenarios where runnable thread goes into the blocking state • I/O: like reading from a file or database. • Lock acquired: if thread want to lock on a resource which is locked by other thread, It has to wait. • Rleases all the **MONITORS LOCK**

	→ When thread goes into Blocked state it relases all the **MONITOR LOCKS**.
Waiting	• Thread goes into this state when we call the wait() method, makes it non runnable. • Its goes back to runnable, once we call notify() or notifyAll() mrthod. • Rleases all the **MONITORS LOCK.**
Timed Waiting	• Thread waits for specific period of time and comes back to runnable state, after specific condition met. like: sleep(), join() • Do not releases any **monitor locks**
Terminated	• Life of thread is completed, it can not be started back again.:

21. **MONITOR LOCKS.**
    - It helps to make sure that only 1 thread goes inside the particular section of code [a sunchronized block or method]
    - two threads t1 and t2 using same object's for calling a method1.
    - Here both threads are using same resource.
    - After using Synchronized block this synchronized method put's a **MONITORS LOCK** on the object whose calling this method.
    - Other thread need to wait to acquire **MONITOR LOCK.**
    - After perfoming task MONITOR LOCKS GET UNLOCKED or released.
    - If thread t1 and t2 uses different objects let's o1 and o2 they can make their calling to synchronized method.
    - o1.method1()
    - o2.method1()
    - here each object have MONITOR LOCK.

```
package Multithreading;


class MonitorLock{
    public synchronized void task1(){
        try{
           System.out.println("Inside synchronizes task 1");
```

```
            Thread.sleep(10000);
            System.out.println("Inside synchronizes task 1 completed");
        }catch (Exception e){
            System.out.println("Exception: "+e.getMessage());
        }
    }

    public void task2(){
        System.out.println("task 2, but before synchronized");
        //synchronized block
        synchronized (this){
            System.out.println("task2, Inside synchronized block");
        }
    }

    public void task3(){
        System.out.println("task3 without synchronized block");
    }


}

public class MonitorLocksExample {
    public static void main(String[] args){

        MonitorLock monitorLockObj = new MonitorLock();

        Thread t1 = new Thread(()->{monitorLockObj.task1();});
        Thread t2 = new Thread(()->{monitorLockObj.task2();});
        Thread t3 = new Thread(()->{monitorLockObj.task3();});

        t1.start();
        t2.start();
        t3.start();
    }
}
output:
Inside synchronizes task 1
task 2, but before synchronized
task3 without synchronized block
```

```
//after 10 seconds
Inside synchronizes task 1 completed
task2, Inside synchronized block

Process finished with exit code 0
```

- t1.start() call task1() method which is synchronized and put MONITOR LOCK ON monitorLockObj and print "Inside synchronizes task 1" and start waiting for 10 seconds.
- t2.start() invokes the task2() method which is not synchronized and print :" task 2, but before synchronized", and after that it encounters **synchronized** block. and try to put MONITOR LOCK on monitorLockObj, which is already acquire by thread t1, so t2 needs wait until t1 completed it's execution.
- T3 encounter's nothing like synchronized in task3() it simply printed "task3"
- After 10 seconds thred t1 releases it's lock from monitorLockObj.
- and after that t2 put's MONITOR LOCK on monitorLockObj.

Example 2:

```
package Multithreading;

public class SharedResource {
    boolean itemAvailable = false;

    //synchronized put the monitor lock
    public synchronized  void addItem(){
        itemAvailable=true;
        System.out.println("Item added by: "+Thread.currentThread().getName()+" and invoking
all threads");
        notifyAll();// who ever waiting on this object will be notify
    }

    public synchronized void consumeItem(){
        System.out.println("Consume Item method invoked by:
"+Thread.currentThread().getName());
        //using while loop to avoid "spurious wake-up", sometimes because of system noise
        while (!itemAvailable){
            try{
```

```
                System.out.println("Thread "+ Thread.currentThread().getName()+" is ");
                wait();//it releases the all the monitor lock
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
        }
        System.out.println("Item Consumed by: "+Thread.currentThread().getName());
        itemAvailable=false;
    }
}
```

```
package Multithreading;

public class ProduceTask implements Runnable{
    SharedResource sharedResource;

    ProduceTask(SharedResource resource){
        this.sharedResource=resource;
    }

    @Override
    public void run() {
        System.out.println("Producer Thread: "+ Thread.currentThread().getName());
        try {
            Thread.sleep(5000l);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        sharedResource.addItem();
    }
}
```

```
package Multithreading;

public class ConsumeTask implements Runnable{
    SharedResource sharedResource;

    ConsumeTask(SharedResource sharedResource){
```

```
        this.sharedResource = sharedResource;
    }

    @Override
    public void run() {
        System.out.println("Consumer Thread: "+Thread.currentThread().getName());
        sharedResource.consumeItem();
    }
}


package Multithreading;

public class Main {
    public static void main(String[] args){
        System.out.println("Main Method Start");
        SharedResource sharedResource = new SharedResource();

        //producer thread
        Thread produceThread = new Thread(new ProduceTask(sharedResource));

        //consumer thread
        Thread consumerThread = new Thread(new ConsumeTask(sharedResource));

        //thread is in "RUNNABLE STATE"
        produceThread.start();
        consumerThread.start();

        System.out.println("Main method end");
    }
}
output:
Main Method Start
Main method end
Producer Thread: Thread-0
Consumer Thread: Thread-1
Consume Item method invoked by: Thread-1
Thread Thread-1 is waiting for resource

//wait 5 seconds
```

```
Item added by: Thread-0 and invoking all threads
Item Consumed by: Thread-1

Process finished with exit code 0
```

Assignment : Implement Producer Consumer Problem.

- Two threads, a producer and consumer, share a common , fixed sized buffer as a queue.
- The Producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data fromthe buffer.
- The problem is to make sure that the producer won't produce data if the buffer is full and the consumer won't consume the data if the buffer is Empty,

```
package Multithreading.ActualProsucerConsumerProblem;

import java.util.LinkedList;
import java.util.Queue;

public class SharedResources {
    private Queue<Integer> sharedBuffer;
    private int bufferSize;

    public SharedResources(int bufferSize){
        sharedBuffer = new LinkedList<>();
        this.bufferSize=bufferSize;
    }

    public synchronized void produce(int item) throws InterruptedException {

        //if Buffer is full, wait for the consumer to consume items;
        while (sharedBuffer.size()==bufferSize){
            System.out.println("Buffer is full, Producer is waiting for consumer");
            wait();
        }
        sharedBuffer.add(item);
        System.out.println("Produce: "+item);
        //Notify the consumer that there are items to consume now.
        notify();
    }
```

```
    public synchronized void consume() throws InterruptedException {
        //Buffer is empty, wait for the producer to produce items
        while(sharedBuffer.isEmpty()){
            System.out.println("Buffer is empty, Consumer is waiting for producer");
            wait();
        }

        int item = sharedBuffer.poll();
        System.out.println("Consumed: "+item);
        //Notify the producer that there is space in the buffer now
        notify();
    }
}


package Multithreading.ActualProsucerConsumerProblem;

public class ProducerConsumerMain {
    public static void main(String[] args){

        SharedResources sharedResources = new SharedResources(3);
        //creating producer thread using Lambda expression;
        Thread producerThread = new Thread(()->{
            try{
                for (int i=1;i<=6;i++){
                    sharedResources.produce(i);
                }
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
        });

        //creting consumer thread
        Thread consumerThread = new Thread(()->{
            try{
                for (int i=1;i<=6;i++){
                    sharedResources.consume();
                }
            }catch (Exception e){
```

```
                System.out.println(e.getMessage());
            }
        });
        producerThread.start();
        consumerThread.start();
    }
}

output:
Produce: 1
Produce: 2
Produce: 3
Buffer is full, Producer is waiting for consumer
Consumed: 1
Consumed: 2
Consumed: 3
Buffer is empty, Consumer is waiting for producer
Produce: 4
Produce: 5
Produce: 6
Consumed: 4
Consumed: 5
Consumed: 6

Process finished with exit code 0
```

22. **Why Stop(), Resume(), Suspended() method is depricated** ?

STOP: Terminates the thread abruptly, no lick releases, No resorces clean up happens

- IT may causes dead lock.
- T1 → lock →R1
- T2→READY→R1
- T1.STOP() → it will not going to releases monitor lock as well as resources.

SUSPENDED: Put Thread on hold [suspend] for temporarily. no lock releases too.

RESUME: Used to resume the execution of SUSPENDED THREAD

→Both this operation could led to issues like deadlock let see an example of it.

→Like Wait() and notify() same as suspended() and resume() so, that's why if suspended depricated the what's needs of resume.

time	Main	T1	T2
t1	started		
t2	T1 & T2 created	CREATED	CREATED
t3	started t1 & t2	Acquire lock R1	
t4			try to acquire lock R1 • T2 is waiting in queue for lock to release
t5	T1 Suspended. • Lock NOT Releases	SUPENDED	KEEP ON WAITING DEAD LOCK

```
package Multithreading.DeadLockImplementation;

public class DeadLockExampleMain {
    public static void main(String[] args) throws Exception{
        SharedResources sharedResources = new SharedResources();
        System.out.println("main thread started");

        Thread th1 = new Thread(()->{
            System.out.println("Thread1 is calling");
            sharedResources.produce();
        });

        Thread th2 = new Thread(()->{
            try {
                //here we knowingly wanted to aucquire lock by thread 1
                Thread.sleep(1000);
            }catch (Exception e){
```

```
                System.out.println(e.getMessage());
            }
            sharedResources.produce();
        });

        th1.start();
        th2.start();

        try {
            System.out.println("thread 2 calling produce method");
            Thread.sleep(3000);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("Thread is suspended");
        th1.suspend();

        System.out.println("Main thread finising it's work");

    }
}
output:
main thread started
thread 2 calling produce method
Thread1 is calling
Lock Acquire.
Thread is suspended
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.base/java.lang.Thread.suspend(Thread.java:1809)
    at
Multithreading.DeadLockImplementation.DeadLockExampleMain.main(DeadLockExampleMain.java:33)
Lock Release
Lock Acquire.
Lock Release

Process finished with exit code 1
```

23. **JOIN**

- When JOIN method is invoked on a thread object. Current thread will be blocked and waits for the specific thread to finish.

- It is help ful when we want to coordinate between threads or to ensure we are complete certain task before moving ahead.
- In simle java program what happen if therad 1 created and started by main thread, the main thread got finished off and do not wait for thread t1 to complete it's execution.

```
public class SharedResources {
    boolean isAvailable = false;

    public synchronized void produce(){
        System.out.println("Lock Acquire.");
        isAvailable=true;
        try {
            Thread.sleep(8000);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("Lock Release");
    }
}
```

```
public class Main {
    public static void main(String[] args) throws Exception{
        SharedResources sharedResources = new SharedResources();
        System.out.println("main thread started");

        Thread th1 = new Thread(()->{
            System.out.println("Thread1 is calling");
            sharedResources.produce();
        });

        th1.start();

        System.out.println("Main thread finising it's work");
```

```
    }
}

output:
main thread started
Main thread finising it's work
Thread1 is calling
Lock Acquire.

//wait for 8 sec
Lock Release

Process finished with exit code 0
```

- but in case of JOIN, Thread has wait until other joining thread complete ther task.

```
public class ThreadJoinExampleMain {
    public static void main(String[] args) throws Exception{
        SharedResources sharedResources = new SharedResources();
        System.out.println("main thread started");

        Thread th1 = new Thread(()->{
            System.out.println("Thread1 is calling");
            sharedResources.produce();
        });

        th1.start();


        try{
            System.out.println("Main thread is waiting for thread th1 to complete their
task");
            th1.join();
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("Main thread finising it's work");
```

```
    }
}
output:
main thread started
Main thread is waiting for thread th1 to complete their task
Thread1 is calling
Lock Acquire.
Lock Release
Main thread finising it's work

Process finished with exit code 0
```

24. **Thread Priority.**

- Priorities are integer ranging from 1 to 10
- 1→low priority
- 10→highest priority
- even we set the thread priority while creation, its not guranteed to follow any specific order, it's just a hint to thread scheduler which to execute next. [but it's not strict rule]
- T3→10, t2→2, t0→1
- This order of execution of thread is not guranteed by JVM.
- Some times you can see or may not
- When new thread is created , it inherit the priority of it's parent thread.
- Main therad priority 5 then inherited thread is have a priority of 5.
- We can set custom priority using setPriority(int priority) method.

24 DAEMON THREAD:

- Some thing which is running async or asynchrouslnly.
- two types of thread
- user thread and Daemon thread
- What ever the thread we have created till now it's all are user thread.
- if you want to make it a thread to daemon.
- you need to do like this.
- thread.setDaemon(true)
- Main thread is also consider as user thread.
- inside main thread we have created daemon thera t1.
- after creating and starting t1 daemon thread main thread continue it's further working and t1 daemon thread is also continue it's working.
- so, when main thread get's stop or complete it's work then t1 daemon thread is also get's stopped.

- daemon thread is alive untill any one user thread alive.
- It is **GARBAGE COLLECTOR**.

→ IT'S act as daemon thread when JVM is running GC in background it releases the memory.

- **AUTO SAVE**:

→ While you are typing it's in background save all the changes periodically.

- **Logging**:

→While running of your program it's maintin the log in background.

- Very Big task is handled by **DAEMON THREAD**.

```
public class DaemonThreadExampleMain {
    public static void main(String[] args) throws Exception{
        SharedResources sharedResources = new SharedResources();
        System.out.println("main thread started");

        Thread th1 = new Thread(()->{
            System.out.println("Thread1 is calling");
            sharedResources.produce();
        });
        th1.setDaemon(true);
        th1.start();
        System.out.println("Main thread finising it's work");

    }
}

output:
main thread started
Main thread finising it's work
Thread1 is calling
Lock Acquire.
```

25 LOCK AND CONDITIONS.

Locks and Semaphores.

- Locking do not depends on object like synchronize method.
- in synchronized blockor method they called to be a critical section.
- this synchronized block put MONITOR LOCK on Object.
- only one thread allow to enter in critical section.

- If we have two different object's and thread, and t1 uses o1 and t2 uses o2, so both t1 & t2 apply monitor locks, o1 locks by t1 & o2 locks by t2. because two threads are sharing two different resources so, both t1 & t2 are allow to enter critical section.

```
package Multithreading.Lock_and_semaphores;

public class SharedResources {
    boolean isAvailable = false;
    public synchronized void producer(){
        try {
            System.out.println("Lock acquire by: "+Thread.currentThread().getName());
            isAvailable=true;
            Thread.sleep(4000);
            System.out.println("Lock releases by: "+Thread.currentThread().getName());
        }catch (Exception e){
           System.out.println(e.getMessage());
        }
    }
}

package Multithreading.Lock_and_semaphores;

public class LockONTWoObjects {
    public static void main(String[] args){
        SharedResources resources1 = new SharedResources();
        Thread th1 = new Thread(()->{
            resources1.producer();
        });

        SharedResources resources2 = new SharedResources();
        Thread th2 = new Thread(()->{
            resources2.producer();
        });

        th1.start();
        th2.start();
    }
}
output:
Lock acquire by: Thread-0
```

```
Lock acquire by: Thread-1

//print after 4 second
Lock releases by: Thread-0
Lock releases by: Thread-1

Process finished with exit code 0
```

- But, if we don't want to allow threads to enter in crititcal section whether they are sharing same resources or they have shared different resources.
- Multiple threads working with multiple objects.

There are 4 types of customs Locks.

- ReentrantLock
- Read-Write
- Semaphore
- Stamp

These locks are do not apply or lock the objects.

26. ReentrantLocks:

```
package Multithreading.Lock_and_semaphores;

public class SharedResources {
    boolean isAvailable = false;

    public void reenTrantLockProducer( ReentrantLock reentrantLock){

        try {
            reentrantLock.lock();
            System.out.println("ReentrantLock acquire by:
"+Thread.currentThread().getName());
            isAvailable=true;
            Thread.sleep(4000);

        }catch (Exception e){
            System.out.println(e.getMessage());
```

```
        }
        finally {
            reentrantLock.unlock();
            System.out.println("ReentrantLock releases by:
"+Thread.currentThread().getName());
        }

    }
}


package Multithreading.Lock_and_semaphores;

import java.util.concurrent.locks.ReentrantLock;

public class LockONTWoObjects {

    public static void main(String[] args){
        ReentrantLock reentrantLock = new ReentrantLock();
        SharedResources resources1 = new SharedResources();
        Thread th1 = new Thread(()->{
            resources1.reenTrantLockProducer(reentrantLock);
        });

        SharedResources resources2 = new SharedResources();
        Thread th2 = new Thread(()->{
            resources2.reenTrantLockProducer(reentrantLock);
        });

        th1.start();
        th2.start();
    }
}
output:
ReentrantLock acquire by: Thread-0

//after 4 second
ReentrantLock acquire by: Thread-1
ReentrantLock releases by: Thread-0
```

```
//after 4 sec
ReentrantLock releases by: Thread-1

Process finished with exit code 0
```

27. **Before Learning about Read-write LOCK**

ReadLock: → More than 1 thred can acquire the read lock

WriteLock: Only 1 thread can acquire the write lock.

- **Shared (s) lock & Xclusive (x) Lock**
- Two thread, let's T1 put shared lock then T2 can also put shared lock on some particular resorces in this case T1 & T2 are only allow to read that block.
- If T1 put's **Shared Lock** on particular resorces then T2 or any other thread are not allow to put **Xclusive lock** on that particular resource.
- **Xclusive lock** can only be take by threads on some particular resource when resource is not lock, not even shared lock some threads.
- With Xclusive Lock you can both read and write.
- But with Shared Lock we can only read that resources.
- If any thread acquire **Xclusive lock** then no thread can acquire any other lock, not even Shared Lock and no other thread can read that resources.

```
package Multithreading.Lock_and_semaphores;

import java.util.concurrent.locks.ReadWriteLock;

    public void readWriteProducer(ReadWriteLock readWriteLock){

        try {
            readWriteLock.readLock().lock();
            System.out.println("ReadWritLock For Producer READ LOCK acquire by:
"+Thread.currentThread().getName());
            isAvailable=true;
            Thread.sleep(8000);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        finally {
            readWriteLock.readLock().unlock();
            System.out.println("ReadWritLock For Producer READ LOCK releases by:
"+Thread.currentThread().getName());
```

```
        }

    }

    public void readWriteConsumer(ReadWriteLock readWriteLock){

        try {
            readWriteLock.writeLock().lock();
            System.out.println("ReadWritLock For Consumer READ LOCK acquire by:
"+Thread.currentThread().getName());
            isAvailable=false;
            Thread.sleep(8000);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        finally {
            readWriteLock.writeLock().unlock();
            System.out.println("ReadWritLock For Consumer READ LOCK releases by:
"+Thread.currentThread().getName());

        }

    }
}


package Multithreading.Lock_and_semaphores;

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class LockONTWoObjects {

    public static void main(String[] args){
        ReadWriteLock  readWriteLock = new ReentrantReadWriteLock();
        SharedResources resources1 = new SharedResources();
        Thread th1 = new Thread(()->{
            //resources1.reenTrantLockProducer(reentrantLock);
```

```
            resources1.readWriteProducer(readWriteLock);
        });


        //already shared lock apply on resources1 by thread th1
        Thread th2 = new Thread(()->{
            //resources2.reenTrantLockProducer(reentrantLock);
            resources1.readWriteProducer(readWriteLock);
        });

        //already shared lock apply on rearWriteConsumerBlock by thread th1 and th2 , and we
are try to apply Xclusive lock on re
        SharedResources resources2 = new SharedResources();
        Thread th3 = new Thread(()->{
            //resources2.reenTrantLockProducer(reentrantLock);
            resources1.readWriteConsumer(readWriteLock);
        });


        th1.start();
        th2.start();
        th3.start();
    }
}

output:
ReadWritLock For Producer READ LOCK acquire by: Thread-1
ReadWritLock For Producer READ LOCK acquire by: Thread-0

//after 8 second
ReadWritLock For Producer READ LOCK releases by: Thread-0
ReadWritLock For Producer READ LOCK releases by: Thread-1
ReadWritLock For Consumer READ LOCK acquire by: Thread-2 consumer thread

//after 8 sec
ReadWritLock For Consumer READ LOCK releases by: Thread-2 consumer thread
```

When to use ?

- Read operation is very high as compare to write operation.
- If multiple thread only wants read opertaion then they do not need to wait they can easily perform read operation in case of Shared lock.
- But what if there is very high write operation then thread with read or shared lock need wait for long time to acquired shared lock.

28. STAMPED LOCK.

- Read-Write Lock
- Optimisitic Read

LOCKS

- **Pessimistic LOCK**

→ALL lock's till now are types pessmisitic lock like sahred lock ,Xclusive lock, synchronized and Rwnterrant lock.

- Optimistic Lock

→ There is no lock acquire.

Id	Name	Type
123	SJ	Student
456	Ram	student

time	Thread T1 ID[123]	Thread T2 ID[123]
t1	Read(123) ROW VERSION IS 1	Read(123) ROW VERISION 1
t2	update: type=teacher they are performing some opertation before updating the db	update type:- Ex-Student they are performing some opertation before updating the db
t3		update (123) type =exstudent Update table set type='ex-student' where id=123 and ROW_VERSION=1→at read time row version is 1 so that's why t2 check for row version 1 =>update successfylly

t4	t1 try to update Update Table set type ='teacher' where id=123 and row_version=1; **update failed** becasue row version is incrmented to 2 by t2 thread • t1 again perform read operation and update it's row version to 2 and then again t2 try to update the row with id 123 Update Table set type ='teacher' where id=123 and row_version=2;	

Simple read-write lock using stamped lock.

```
package Multithreading.Lock_and_semaphores;

import java.util.concurrent.locks.StampedLock;

public class SharedResources {
    boolean isAvailable = false;

    StampedLock stampedLock = new StampedLock();


    public void readWriteStampProducer(){
        long stamp= stampedLock.readLock();

        try {
            //it returns some version id

            System.out.println("STAMPReadWritLock For Producer READ LOCK acquire by:
"+Thread.currentThread().getName());
            isAvailable=true;
            Thread.sleep(8000);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
```

```
        finally {
            stampedLock.unlockRead(stamp);
            System.out.println("StampReadWritLock For Producer READ LOCK releases by:
"+Thread.currentThread().getName());

        }

    }

    public void readWriteStampConsumer(ReadWriteLock readWriteLock){
        long stamp = stampedLock.writeLock();

        try {

            System.out.println("StampReadWritLock For Consumer READ LOCK acquire by:
"+Thread.currentThread().getName()+" consumer thread");
            isAvailable=false;
            Thread.sleep(8000);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        finally {
            stampedLock.unlockWrite(stamp);
            System.out.println("StampReadWritLock For Consumer READ LOCK releases by:
"+Thread.currentThread().getName()+" consumer thread");

        }

    }
}
```

Optimisitic Functionality.

```
package Multithreading.Lock_and_semaphores;
```

```
import java.util.concurrent.locks.StampedLock;

public class SharedResources {
    StampedLock stampedLock = new StampedLock();



    int a = 10;
    public void readWriteOptimisticStampProducer(){
        long stamp= stampedLock.tryOptimisticRead();
        try {
            a=11;
            Thread.sleep(6000);
            if (stampedLock.validate(stamp)){
                System.out.println("readWriteOptimisticStampProducer Value Update
Successfully " +Thread.currentThread().getName());
            }else{
                a=10;
                System.out.println("readWriteOptimisticStampProducer Value Update
Successfully " +Thread.currentThread().getName());
            }
            System.out.println("readWriteOptimisticStampProducer Value Update ROLLBACK
"+Thread.currentThread().getName());
            isAvailable=true;
            Thread.sleep(8000);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }


    }

   public void readWriteOptimisticStampConsumer(){
        long stamp = stampedLock.writeLock();
        System.out.println("readWriteOptimisticStampConsumer Write LOCK acquire by:
"+Thread.currentThread().getName()+" consumer thread");
        try {
```

```
            System.out.println("readWriteOptimisticStampConsumer Performing some work
"+Thread.currentThread().getName());
               a=9;
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        finally {
            stampedLock.unlockWrite(stamp);
            System.out.println("readWriteOptimisticStampConsumer write releases by:
"+Thread.currentThread().getName()+" consumer thread");

        }
    }

}

package Multithreading.Lock_and_semaphores;



public class LockONTWoObjects {

    public static void main(String[] args){

        SharedResources resources1 = new SharedResources();

        Thread tryOptimisticThread1 = new Thread(()->{
            resources1.readWriteOptimisticStampProducer();
        });

        Thread tryOptimisticThread2 = new Thread(()->{
            resources1.readWriteOptimisticStampConsumer();
        });

        tryOptimisticThread1.start();
        tryOptimisticThread2.start();
    }
}
output:
readWriteOptimisticStampConsumer Write LOCK acquire by: Thread-4 consumer thread
```

```
readWriteOptimisticStampConsumer Performing some work Thread-4
readWriteOptimisticStampConsumer write releases by: Thread-4 consumer thread

//after 6 second
readWriteOptimisticStampProducer Value Update ROLLBACK Thread-3

Process finished with exit code 0
```

29. SEMAPHORES.

- Connection Poo: l How many therad comes into critical section

30. Conditions.

- wait and notify are used for synchronised block.

await() =wait()
signal() = notify()
sinalAll() = notifyAll()

31. In How Many Ways We can achieve ConCurrency.
In 2 ways

- Lock based Mechanism:

→Synchronised, Reentrant, Stamped, ReadWrite, SemaPhores.

- Lock Free Mechanism.

→Faster as compare to lock based
→Used In Very specific area.
→CAS Operation (Compare - and - Swap)
→AtomicInterger
→AtomicBoolean
→AtomicLong
→AtomicReference

32. Lock Free Mechanism.

- It uses CAS [Compare & Swap]Technique .
- It's Low level operation

→ it supports by cpu.

- It's Atomic.
- And all modern Processor supports it.

It involves 3 main parameteres:

- Memoary location: location where variable is stored.
- Expected value: Value which should be present at the memoary.

→ABA problem solved using version or timestamp.

- New Value: value to be written to memory, if the current value matches the expected value.

```
CAS ( Memory _loc , ExpectedValue , newValue){
        →load/read value from main:memory.
        →compare memory data vs expected
        → if matched : update new value to memory.
}
```

- Most similar to Optimistic read 7 write concurrency.
- But CAS is mostly used with on CPU level.
- Optimistic concurrency used on db level concurrency.

ABA Problem .

- CAS (M1 , 10, 12){

→ REad M1

→ Compare M1 value with expected val=10

→ Update If Matched

}

- if some operation occur first value changes to 10 and after that it changes to 12 and after it's changes back to 10.
- Thread t1 wants to update tat value and it expected value is 10→ which is old one but in memory the value 10 was changed long by multiple threads by their operation 10→12→10
- This Calles ABA Issues.
- This problem is resolved by adding timestamp and version.
- 10 [v1,t1]→12[v2,t2]→10[v3,t3]

Atomic Variabbles:

What ATOMIC means:

- It means Single or "all or nothing"
- int counter =10;
- no matter how many thread will executes this counter it will going to consistent.

```
public class SharedResources {
```

```
    int counter;
    public void increment(){
        //this is not atomic operation
        counter++;
    }

    public int get(){
        return counter;
    }

}
```

- counter++

→Load counter valur

→ Increment By 1

→ Assign back.

- If two threads try t1 & t2 try to reads & update value on concurrent basis , the conter value is not goint to consistent

time	thread t1	thread t2
t1	conter value=0	counter value=0
t2	countr++	counter++

```
//single main therad
public class Main{

    public static void main(String[] args){

        SharedResources resources1 = new SharedResources();


        for (int i=0;i<400;i++){
            resources1.increment();
        }
        System.out.println("main threadresources1.get(): "+resources1.get());



    }
}
```

```
output:
resources1.get(): 400

Process finished with exit code 0



//lt1 & lt2 thread
public class Main{

    public static void main(String[] args){

        SharedResources resources1 = new SharedResources();


        for (int i=0;i<400;i++){
            resources1.increment();
        }
        System.out.println("main threadresources1.get(): "+resources1.get());


    }
}
output:
resources1.get(): 400

Process finished with exit code 0


public class LockONTWoObjects {

    public static void main(String[] args){

        SharedResources resources1 = new SharedResources();
        Thread lt1 = new Thread(()->{
            for (int i=0;i<200;i++){
                resources1.increment();
            }
        });
        Thread lt2 = new Thread(()->{
```

```
            for (int i=0;i<200;i++){
                resources1.increment();
            }
        });
        lt1.start();
        lt2.start();
        try {
            lt1.join();
            lt2.join();

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("lt1 & lt2 thread threadresources1.get(): "+resources1.get());

    }
}
output:
//at my laptop i am getting lt1->200 & lt2->200
lt1 & lt2 thread threadresources1.get(): 400

Process finished with exit code 0


//but for larger value like lt1->10000 & lt2->10000
lt1 & lt2 thread threadresources1.get(): 14543

Process finished with exit code 0
```

SOLUTIONS:

- Using keywork like Synchronized.

```
 public synchronized void increment(){
        //this is not atomic operation
        counter++;
    }
```

- Using lock free operations like AtomicInterger.

→Internally USES CAS [Compare & Swap]

→When you can use CAS

→ READ , MODIFY, AND UPDATE

```
AtomicInteger atomicIntegerCounter = new AtomicInteger(0);
```

```
private volatile int value;

/**
 * Creates a new AtomicInteger with the given initial value.
 *
 * @param initialValue the initial value
 */
public AtomicInteger(int initialValue) {
    value = initialValue;
}
```

volatile :

- Always goes to a particular mainmemoary address and reads it

```
public final int incrementAndGet() {
    return U.getAndAddInt(this, VALUE, 1) + 1;
}
```

```
// The following contain CAS-based Java implementations used on
// platforms not supporting native instructions

/**
 * Atomically adds the given value to the current value of a field
 * or array element within the given object {@code o}
 * at the given {@code offset}.
 *
 * @param o object/array to update the field/element in
 * @param offset field/element offset
 * @param delta the value to add
```

```
     * @return the previous value
     * @since 1.8
     */

    @IntrinsicCandidate
    public final int getAndAddInt(Object o, long offset, int delta) {
        int v;
        do {
            //expected read value from memory
            v = getIntVolatile(o, offset);
        } while (!weakCompareAndSetInt(o, offset, v, v + delta));
        return v;
    }

    /** Volatile version of {@link #getInt(Object, long)}  */
    @IntrinsicCandidate
    public native int getIntVolatile(Object o, long offset);
```

object→memory address
offset→expected value
delta→+1 increment
weakCompareAndSetInt→CAS

```
    @IntrinsicCandidate
    public final boolean weakCompareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x) {
        return compareAndSetInt(o, offset, expected, x);
    }

    /**
     * Atomically updates Java variable to {@code x} if it is currently
     * holding {@code expected}.
     *
     * <p>This operation has memory semantics of a {@code volatile} read
     * and write.  Corresponds to C11 atomic_compare_exchange_strong.
     *
     * @return {@code true} if successful
     */
```

```
@IntrinsicCandidate
public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);
```

```
import java.util.concurrent.atomic.AtomicInteger;
public class SharedResources {
    AtomicInteger atomicIntegerCounter = new AtomicInteger(0);
    public void atomicIntegerCounterIncrement(){
        atomicIntegerCounter.incrementAndGet();
    }

    public int getAtomicInteger(){
        return atomicIntegerCounter.get();
    }

}
```

```
public class LockONTWoObjects {

    public static void main(String[] args){
        Thread lt3 = new Thread(()->{
            for (int i=0;i<10000;i++){
                resources1.atomicIntegerCounterIncrement();
            }
        });
        Thread lt4 = new Thread(()->{
            for (int i=0;i<10000;i++){
                resources1.atomicIntegerCounterIncrement();
            }
        });
        lt3.start();
        lt4.start();
```

```
        try {
            lt3.join();
            lt4.join();

        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("lt3 & lt4 thread threadresources1.get():
"+resources1.getAtomicInteger());

    }
}
output:
lt3 & lt4 thread threadresources1.get(): 20000
```

time	thread t1	thread t2
t1	incrementAndGet() expected=0	ncrementAndGet() expected=0
t2	t1 get CAS 1. read data from memory ehich is 0 2. compare ==expected[0==0] 3. update value	
t3		1. read data from memory which is 1 2. expected value =0 3. apply CAS Operation 4. this will return false. 5. it will go back do while loop again read expected value from loop 6. again apply CAS Operation 7. it will return true.

		8. and then update it to final increment is 2.

Many engineer's got confused with volatile and atomic.

33. VOLATILE

Volatile	Atomic
1. thread safe	1. not thread safe and not relation with concurrency.

34. Concurrent collection.

collection	Concurrent Collection	Lock
PriorityQueue	PriorityBlockingQueue	ReentrantLock
LinkedList	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayDeque	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayList	CopyOnWriteArrayList	ReentrantLock

HashSet	newKeySet method inside ConcurrentHashMap	synchronized
TreeSet	Collections.synchronizedSortedSet	synchronized
LinkedHashSet	Collections.synchronizedSet	synchronized
Queue Interface	ConcurrentLinkedQueue	Compare-and-swap operation

35. ThreadPool,ExecutorFramework, ExecutorService.

Interview question:

- In ThreadPool, why you have taken corePoolSize as 2, why not 10 or 15 or another number, what's the logic.

36. What is ThreadPool ?

- It's a collection of threads [aka worker, which are available to perform the submitted tasks.
- Once task completed, worker thred get back to Thread Pool and wait for new task to assigned.
- Means threads can be reused.
- When threads are busy and task3 is come so, there is no thread available for to entertain this new task , so task 3will put it into queue.
- Now thread 1 completed it's task and comback to pool and try to check queue to find any available task let's task 3 is availale then it will excute the task 3 .

What's the Advantage oF Thread Pool ?

- Thread Creation time can be saved:

→ When each thread created, space is allocated to it [stack, heap, program, counter etc.] and this take time.

→With thread, this can be avoided by reusing the thread.

- Overhead of managing the Thread lifecycle can be removed:

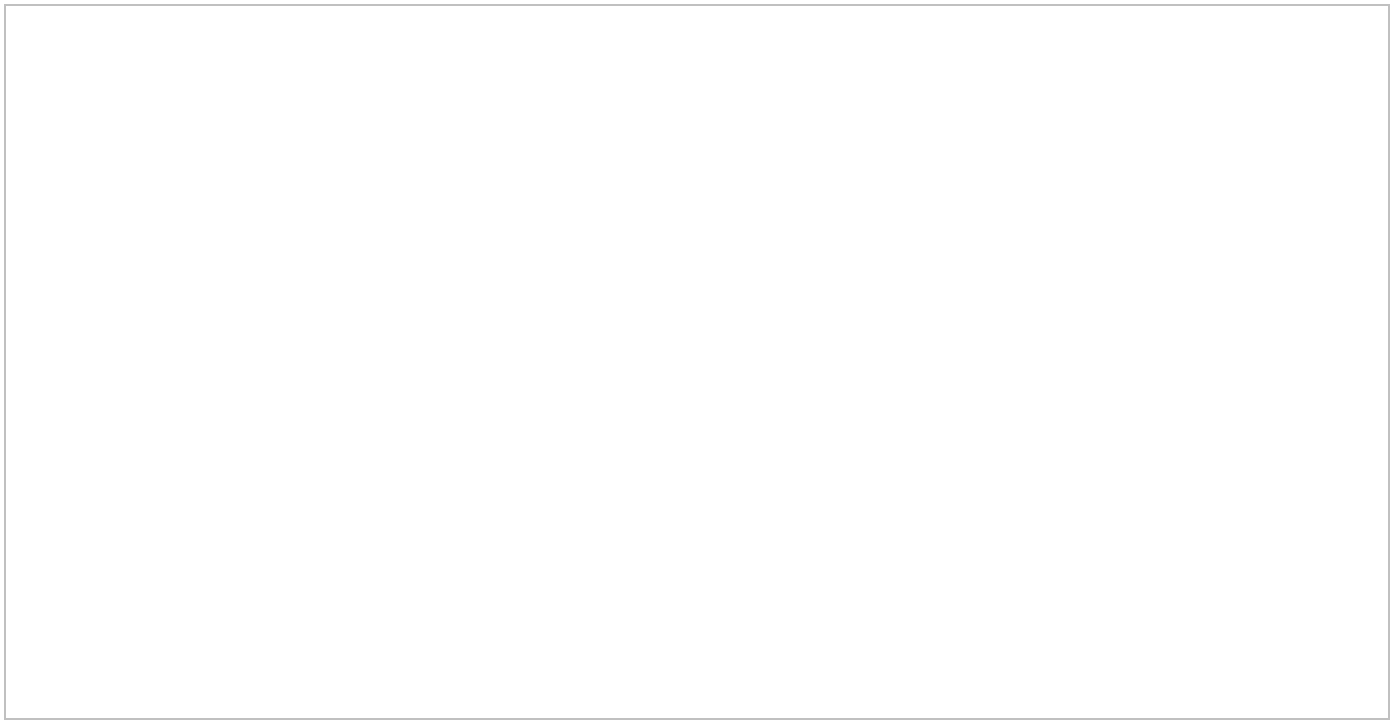
→Thread has different state like Runnning, Waitting, terminate etc. And managing thread state includes complexity.

→Thread Pool abstract away this management.

- Increase the performance

→More Threads means, more Context Switiching time, using control over thread creation, excess context swithching can be avoided.

37. ExecutorFrame Work.



38. Thread Pool Executor:

- It's helps to create a customizable ThreadPool.

Process / Flow

- There are multiple task is comming to thread pool executor .
- first it will check, any thread is free in thread pool.
- There is pool contain 3 thread t1,t2 and t3
- let's t1 → assign task1 →now t1 is busy.
- let's t2 → assign task2 →now t2 is busy.
- let's task3 → comes at time ttt3→ so t3 thread is free, t3→assign task3
- minimum number of thread is 3
- now task 4 is comes , no thread is available, task 4 will goes into queue.
- same for task5, task 6,task 7 and task8.
- queue size is only 5.

- now queue is also full
- all thread is also busy doing their task.
- now task 9 is comes then what happens.
- we can also give maximum number of thread can contain by thread pool at a time.
- maximum size is 5
- and there is a space in thread pool
- new thread t4 will be create and task 9 will assign to it.
- thread t4 →assgin task → task 9
- Now task 10 has comes all 4 thread are busy, check queue, it is also full,
- Now check max size of thread pool , it is 5, and thread pool contain only 4 threads
- we can create one more thread.
- task10 assign →thread t10.
- task 11 has comes, it will rejected this task.
- let's t3 thread is available, now it's check the queue and pick one of the task.

ThreadPoolExecutor Constructor.

public ThreadPoolExecutor(int corePoolSize,
int maximumPoolSize,
TimeUnit unit,
BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory,
RejectedExecutionHandler handler)

corePoolSize:

- Number of threads are initially created and keep in the pool, even if they are idle.

allowCoreThreadTimeOut:

- If this property is set to TRUE [by default its false], idle thread kept alive till time span by "**KeepAliveTime**".

KeepAliveTime:

- Thread, which are idle get terminated after this time.

TimeUnit:

- TimeUnit for the KeepAliveTime, whether Millisecond or secon or hours etc.

maxPoolSize:

- Maximum number of thread allowed in a pool .
- if no. of thread are == corePoolSize and queue is also full, then new threads are created [till it's less than 'maxPoolSize']

- Excess thread, will remain in pool, this pool is not shutdown of if allowCoreThreadTime set to true, then excess thread get terminated after remain idle for keepAliveTime.

When all three threads are busy in their task , and when task 4 is come why we are putting task 4 in queue, why we are not creating new therad t4 and assign that task 4 to t4 thread. And we are doing same thing for task 5, task 6, task 7, untill the queue is full.

- Because average pool size or minimum pool size which is 3 is enough or sufficient to do most of their task on avrage.
- And if we are creating thread as per task is comming then thread pool is getting full.
- Most of the time three thread is enough to do most of the task at that time t4 and t5 will be idle.
- that's why we are defining queue if any task will come they will put it into queue.
- if any task will come in case of queue is full then we can create new therad this is worst case scenario.

BlockingQueue.

- Queue used to hold task, before they got picked by the worker thread.
- Bounded Queue Queue with FIXED capacity. → Like ArrayBlockingQueue.
- Unbounded Queue: Queue with NO FIXED capacity. like →LinkedBlockingQueue.

→Generally this is avoided.

ThreadFactory:

Factory for creating new thread. ThreadPoolExecutor use this to create new thread, this factory provide us an interface to:

- To give custom Thread name
- To give custom Thread priority
- To set Thread Daemon flag etc.

RejectedExecutionHandler:

Handler for tasks that can not be accepted by thread pool. Generally logging logic can be put here. for debugging purpose.

- new ThreadPollExecutor.AbortPolicy

→Throws RejectedExecutionException.

- new ThreadPollExecutor.CallerRunsPolicy

→Executed the rejected task in the caller thread [thread that attempted to submit.

- new ThreadPollExecutor.DiscardPolicy

→Silently discard the Rejected task, without throwing any exception.

- new ThreadPollExecutor.DiscardOldestPolicy

→Discared the oldest task in the queue. to accomodeate new task.

LifeCycle Of ThreadPool:

Running:

- Executor is in running state and submit() method will be used to add new task.

Shutdown:

- Executor do not accept new tasks, but continue to process exisiting tasks, once existing tasks finished, executor moves to terminated state.
- Method used shutdown()

Stop [Force to shutdown]:

- Executor do not accept new tasks.
- Executor forcefully stops all the tasks which are currently executing.
- And once fully shutdown, moves to terminte state.
- Method used shutdownNow()

Terminated:

- End od life for particular ThreadPoolExecutor.
- isTerminated() method can be used to check if particular thread pool executor is terminated or not.

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

class CustomThreadFactory implements ThreadFactory{

    @Override
    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r);
        thread.setPriority(Thread.NORM_PRIORITY);
```

```
            thread.setDaemon(false);
            //thread.setName("lola");
            return thread;

        }
    }

    class CustomRejectHandler implements RejectedExecutionHandler{

        @Override
        public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
            System.out.println("Task Rejected: "+r.toString());

        }
    }


    public class ThreadPoolExecutorExample {
        public static void main(String[] args){
            ThreadPoolExecutor executor = new ThreadPoolExecutor(2,4,10,
                    TimeUnit.MINUTES,new ArrayBlockingQueue<>(2),
                    new CustomThreadFactory(),
                   // new ThreadPoolExecutor.DiscardOldestPolicy()
                    new CustomRejectHandler()

            );
            executor.allowCoreThreadTimeOut(true);

            for (int i=1;i<=7;i++){
                //accepts runnable
                int finalI = i;
                executor.submit(()->{
                    try {
                        Thread.sleep(5000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    System.out.println("Task"+ finalI + " Processed by: "+
    Thread.currentThread().getName());
                });
```

```
        }
        executor.shutdown();
    }
}
output:
C:\Users\DELL\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program
Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.1\lib\idea_rt.jar=62084:C:\Program
Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.1\bin" -Dfile.encoding=UTF-8 -
Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath "E:\java
projects\Data_Structures_And_Algorithms_JAVA_BY_Saurav_Suman\out\production\Data_Structures_A
nd_Algorithms_JAVA_BY_Saurav_Suman"
Multithreading.ExecutorFrameWork.ThreadPoolExecutorExample
Task Rejected: java.util.concurrent.FutureTask@506e1b77[Not completed, task =
java.util.concurrent.Executors$RunnableAdapter@27bc2616[Wrapped task =
Multithreading.ExecutorFrameWork.ThreadPoolExecutorExample$$Lambda/0x0000018001003640@3941a79
c]]
Task1 Processed by: Thread-0
Task5 Processed by: Thread-2
Task6 Processed by: Thread-3
Task2 Processed by: Thread-1
Task4 Processed by: Thread-2
Task3 Processed by: Thread-0

Process finished with exit code 0
```

Interview question:

- In ThreadPool, why you have taken corePoolSize as 2, why not 10 or 15 or another number, what's the logic.

Ans:

- Gnerally, the threadpool min and max size are depend on various factors like:
- CPU Cores

→ If we have two core CPU and we have 100's of thread, they do most of the time context switching it is time taking.

- JVM Memory

→ Each thread have specific memory like counter , resister and stack, and it take space if we have 100 of thread's and JVM size is only of 2MB

- Task Nature [CPU Intensive→ Want more CPU time less threads

or

I/O Intensive →Thread is idle at the time of ongoing I.O operation more thread required more context switching occur.

- Concurrency Requirement [Want high or medium or low concurrency]
- Memory Required to process a request. JVM has limited space to support the task.
- ThroughPut etc.

And it'san iterative process to update the min and max values based on monitoring.

Formula to fina the no. of thread.:

Max No of thread = No of CPU * (1 + Request waiting time[i/o] /Processing time[high cpu intensive])

No. of CPU core =64
request waiting time = 50ms
Processing time = 100ms

64 + (1*50/100) = ~64 approx.

Max no of Active tasks = tasks Arrival rate * Task execution time 64/.15 = ~426 approx. task arrival rate per second can be handled.

But this formula, do not consider Memory yet, Which need to be consider...

JVM : 2GB
{
HEAP SPACE: 1GB
CODE CACHE SPACE: 128 MB
PER THREAD SPACE 5MB* N no of thread stack space]
JVM OVERHEAD: 256MB
}
1000mb + 128mb+256mb+x = 2000mb
left with x= 500mb
1 thread take 5 mb

Per REquest requires 10Mb of space to fullfill the request.

39. Future, Callable and CompletableFuture.

→*//Now what is caller want to know the status of the thread1. Whether it's completed or failed etc.*

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;
public class ThreadPoolExecutorExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,TimeUnit.HOURS,new
ArrayBlockingQueue<>(2),Executors.defaultThreadFactory(), new
ThreadPoolExecutor.AbortPolicy());

        //new Thread will be created and it will perform the task.
        //it will always return Future Object.
        poolExecutor.submit(()->{
            System.out.println("this is the task, which thread will execute.");

        });//Now what is caller want to know the status of the thread1. Whether it's
completed or failed etc.



        //main thread all continue to processing it.


    }
}
```

40. Future:

- Interface which Represents the result of the Async task.
- Means, it allow you to check if:

→Computation is complete.

→Get the result

→Take care of exception if any etc.

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class ThreadPoolExecutorExample {
```

```
    public static void main(String[] args){


        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,TimeUnit.HOURS,new
ArrayBlockingQueue<>(2),Executors.defaultThreadFactory(), new
ThreadPoolExecutor.AbortPolicy());

        //new Thread will be created and it will perform the task.
       Future<?> futureObj= poolExecutor.submit(()->{
            System.out.println("this is the task, which thread will execute.");

        });//Now what is caller want to know the status of the thread1. Whether it's
completed or failed etc.

        System.out.println(futureObj.isDone());

        poolExecutor.shutdown();


        //main thread all continue to processing it.


    }
}
output:
false
this is the task, which thread will execute.
```

S. NO.	Method Available in Future Interface.	purpose
1.	boolean cancel(boolean mayInterruptIfRunning)	• Attempts to cancel the the execution of yhe task. • Return false, if task can notnbe canceled. • [Typically bcoz task already completed]; returns true otherwise.
2,	boolean isCanceled()	• Returns true, if task was cancelled befor it was cancelled before it got completed.
3.	boolean isDone()	• Return true if this task completed.

		• Completion may due to normal termination, an exception, or cancellation → In all of these cases, this method will return true.
4.	V get()	• Wait if required, for the completion of the task. • Locking the caller • Waiting indefinately • Main thread will be block untill tsk execution. • After task completed, retrieve the result if available.
5.	V get(long timeout, TimeUnit unit)	• wait if required, for most of the given timeperiod. • Max it can wait depnds on timeunit. • Throws "TimeoutException" if timeout period finished and task is not yet completed.

```
/**
 * @throws RejectedExecutionException {@inheritDoc}
 * @throws NullPointerException       {@inheritDoc}
 */
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}

protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}

public class FutureTask<V> implements RunnableFuture<V> {

public FutureTask(Runnable runnable, V result) {
    this.callable = Executors.callable(runnable, result);
    this.state = NEW;       // ensure visibility of callable
}

}
```

```
    /**
     * Returns a {@link Callable} object that, when
     * called, runs the given task and returns the given result.  This
     * can be useful when applying methods requiring a
     * {@code Callable} to an otherwise resultless action.
     * @param task the task to run
     * @param result the result to return
     * @param <T> the type of the result
     * @return a callable object
     * @throws NullPointerException if task null
     */
public class Executors {
    public static <T> Callable<T> callable(Runnable task, T result) {
        if (task == null)
            throw new NullPointerException();
        return new RunnableAdapter<T>(task, result);
    }


    /**
     * A callable that runs given task and returns given result.
     */
    private static final class RunnableAdapter<T> implements Callable<T> {
        private final Runnable task;
        private final T result;
        RunnableAdapter(Runnable task, T result) {
            this.task = task;
            this.result = result;
        }
        public T call() {
            task.run();
            return result;
        }
        public String toString() {
            return super.toString() + "[Wrapped task = " + task + "]";
        }
    }

}
```

Example:

```
package Multithreading.ExecutorFrameWork;

import java.sql.Time;
import java.util.concurrent.*;

public class CompletableFutureExampleWithItsMethod {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1, TimeUnit.HOURS,new
ArrayBlockingQueue<>(2), Executors.defaultThreadFactory(),new
ThreadPoolExecutor.AbortPolicy());

        Future<?> futureObj= poolExecutor.submit(()->{
            try {
                Thread.sleep(7000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println("this is the task, which thread will execute.");

        });


        System.out.println("Is Done: "+futureObj.isDone());

        try {
            futureObj.get(2,TimeUnit.SECONDS);
        }catch (TimeoutException | InterruptedException | ExecutionException e){
            System.out.println("Ex->2"+e.getMessage());
        }

        try {
            futureObj.get();
            //here main thread will be blocked
        }catch (Exception e){
            System.out.println("Ex->1"+e.getMessage());
        }
```

```
        System.out.println("is done: "+futureObj.isDone());
        System.out.println("is cancelled: "+futureObj.isCancelled());

        poolExecutor.shutdown();

    }

}
output:
Is Done: false
Ex->2null
this is the task, which thread will execute.
is done: true
is cancelled: false

Process finished with exit code 0
```

41. Callable Interface:

- submit(Runnable)
- submit(Runnable, T)
- submit(Callable<>)
- Callable represents the task which need to be executed just like Runnable.
- Runnable do not have any return type.
- Callable has the capability to return the value.

Runnable Interface	Callable Interface

	@FunctionalInterface
	public interface Callable<V>{
	V call() throws Exception;
	}

USECASE 1 Future<?> fObj = poolExecutor.submit(()→System.out.println("do something")) ?→wildcared→it can be anything Object obj = fObj.get();	Future<Integer> fObj = poolExecutor.submit(()→{ System.out.println("do something"); return 45; })

→Object class is parent of every thing.
System.out.println(obj==null)

- In this case we need future obj so we can able to find it out what is the status of the task execution.

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task,
null);
    execute(ftask);
    return ftask;
}
```

```
package Multithreading.ExecutorFrameWork;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

class MyRunnableForUseCase2 implements Runnable{

    List<Integer> list;
    MyRunnableForUseCase2(List<Integer> list){
        this.list=list;
    }
    @Override
    public void run() {
        list.add(700);

    }
}

public class RunnableVsCallableExample {

    public static void main(String[] args){
```

```
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1, TimeUnit.HOURS,new
ArrayBlockingQueue<>(2), Executors.defaultThreadFactory(),new
ThreadPoolExecutor.AbortPolicy());


        //UseCase 1
        Future<?> futureObj= poolExecutor.submit(()->{
            System.out.println("this is the task, which thread will execute.");

        });
        try {
            Object obj = futureObj.get();
            System.out.println(obj==null);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }


        //UseCase2
        List<Integer> output = new ArrayList<>();
        Future<List<Integer>> futureObjUseCase2 = poolExecutor.submit(new
MyRunnableForUseCase2(output),output);

        try {
            List<Integer> integerList = futureObjUseCase2.get();
            System.out.println("UseCaseResult2: "+integerList);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }


        //UseCase3

        Future<List<Integer>> futureObjUseCase3 = poolExecutor.submit(()->{
            System.out.println("Usecase3 3 With Callable");
            List<Integer> list = new ArrayList<>();
            list.add(300);
            return list;
```

```
        });

        try {
            List<Integer> integerList = futureObjUseCase3.get();
            System.out.println("UseCase3Result3: "+integerList);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }

        poolExecutor.shutdown();


    }
}
output:

this is the task, which thread will execute.
true
UseCaseResult2: [700]
Usecase3 3 With Callable
UseCase3Result3: [300]

Process finished with exit code 0
```

42. CompletableFuture:

- Introduced in java 8
- To help in async programming
- We can considered it as an advanced version of Future provides additional capability like chaining.

```
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {}
```

How to use this:

- **CompletableFuture.supplyAsync:**

public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier)

public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier, Executor executor)

- If we are not passing the executor it uses default ForkJoinPool.
- It will take that thread and pass to the task to execute them.

- You can also use specific thread pool executor.

→supplyAsync method initiates Async operation .
→'supplier' is executed asynchronously in a sperated thread.
→If we want more control on Threads, we can pass Executor in the method.
→By default its uses, shared Fork-Join Pool executor. it dynamically adjust it's pool size processors.

```
 /**
 * Returns a new CompletableFuture that is asynchronously completed
 * by a task running in the given executor with the value obtained
 * by calling the given Supplier.
 *
 * @param supplier a function returning the value to be used
 * to complete the returned CompletableFuture
 * @param executor the executor to use for asynchronous execution
 * @param <U> the function's return type
 * @return the new CompletableFuture
 */
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,
                                                   Executor executor) {
    return asyncSupplyStage(screenExecutor(executor), supplier);
}

static Executor screenExecutor(Executor e) {
    if (!USE_COMMON_POOL && e == ForkJoinPool.commonPool())
        return ASYNC_POOL;
    if (e == null) throw new NullPointerException();
    return e;
}

@SuppressWarnings("serial")
static final class AsyncSupply<T> extends ForkJoinTask<Void>
    implements Runnable, AsynchronousCompletionTask {
    CompletableFuture<T> dep; Supplier<? extends T> fn;
    AsyncSupply(CompletableFuture<T> dep, Supplier<? extends T> fn) {
        this.dep = dep; this.fn = fn;
    }

    public final Void getRawResult() { return null; }
    public final void setRawResult(Void v) {}
```

```
    public final boolean exec() { run(); return false; }

    public void run() {
        CompletableFuture<T> d; Supplier<? extends T> f;
        if ((d = dep) != null && (f = fn) != null) {
            dep = null; fn = null;
            if (d.result == null) {
                try {
                    d.completeValue(f.get());
                } catch (Throwable ex) {
                    d.completeThrowable(ex);
                }
            }
            d.postComplete();
        }
    }
}



@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}

static <U> CompletableFuture<U> asyncSupplyStage(Executor e,
                                                 Supplier<U> f) {
    if (f == null) throw new NullPointerException();
    CompletableFuture<U> d = new CompletableFuture<U>();
    e.execute(new AsyncSupply<U>(d, f));
    return d;
}

public interface Executor {
```

```
    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

EXAMPLE:

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());



        try {
            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(()->{
                //This is the task which need to to be completed by thread.
                System.out.println("asyncTask1 thread name:
"+Thread.currentThread().getName());
                return "Task Completed";
            },poolExecutor);

            System.out.println("asyncTask1.get(): "+ asyncTask1.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
```

```
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }
        poolExecutor.shutdown();

    }
}
OUTPUT:
asyncTask1 thread name: pool-1-thread-1
asyncTask1.get(): Task Completed

Process finished with exit code 0
```

- **thenApply & thenApplyAsync:**

→Apply a function to the result of previous Asunc computation.
→Return a new CompleatableFuture object.

thenApply:

- It's a synchronous execution.
- Means, It uses The same thread, which completed the previous Async task.

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());



        try {
            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(()->{
                //This is the task which need to to be completed by thread.
                System.out.println("asyncTask1 in supplyAsync thread name:
"+Thread.currentThread().getName());
```

```
                return "Saurav ";
            },poolExecutor).thenApply((String val)->{
                System.out.println("asyncTask1 in thenApply thread name:
"+Thread.currentThread().getName());
                return val+"Saxena";
            });

            System.out.println("asyncTask1.get(): "+ asyncTask1.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }
        poolExecutor.shutdown();

    }
}
output:
asyncTask1 in supplyAsync thread name: pool-1-thread-1
asyncTask1 in thenApply thread name: pool-1-thread-1
asyncTask1.get(): Saurav Saxena

Process finished with exit code 0
```

thenApplyAsync:

- it's a Asyncchronous execution.
- Means, it uses different thread [From fork-join pool, if we do not provide the executor in
- If Multiple 'thenApplyAsync' is used , ordering can be guranteed,
- they will run concurrently.

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
```

```
                new ThreadPoolExecutor.AbortPolicy());


        try {
            CompletableFuture<String> asyncTask2 = CompletableFuture.supplyAsync(()->{
                //This is the task which need to to be completed by thread.
                System.out.println("asyncTask2 in supplyAsync thread name:
"+Thread.currentThread().getName());
                try {
                    Thread.sleep(8000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                return "Saurav ";
            },poolExecutor);

            CompletableFuture<String> thenApplyAsync=asyncTask2.thenApplyAsync((String val)->
{
                System.out.println("asyncTask2 in thenApplyAsync thread name:
"+Thread.currentThread().getName());
                return val+"Saxena";
            },poolExecutor);
            System.out.println(" asyncTask2.get(): "+ asyncTask2.get());
            System.out.println("thenApplyAsync thenApplyAsync.get(): "+
thenApplyAsync.get());
        } catch (Exception e) {
            System.out.println("Ex=> "+e.getMessage());
        }


        poolExecutor.shutdown();

    }
}
output:
asyncTask2 in supplyAsync thread name: pool-1-thread-1
asyncTask2 in thenApplyAsync thread name: pool-1-thread-2
 asyncTask2.get(): Saurav
thenApplyAsync thenApplyAsync.get(): Saurav Saxena
```

```
Process finished with exit code 0
```

- thenCompose and thenComposeAsync:

→Chain together dependent Async operations

→Means when next Asunc operation depends on the result of the previous .

→We can tied them together.

→For async tasks, we can bring some ordering using this.

internally maintains a certain kind of ordering.

thenCompose:

```
public <U> CompletableFuture<U> thenCompose(
    //? super T -> String and it's upper
    //? extends CompletionStage -> CompletionStage and it's lower
    Function<? super T, ? extends CompletionStage<U>> fn) {
    return uniComposeStage(null, fn);
}
```

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());



                try {
            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(()->{
                //This is the task which need to to be completed by thread.
                System.out.println("asyncTask1 in supplyAsync thread name:
"+Thread.currentThread().getName());
                return "Saurav ";
            },poolExecutor)
```

```
.thenCompose((val)->CompletableFuture.supplyAsync(()->val+"lola "));// order always
maintained
            //in case of thenComposeAsync we can pass executor.
         CompletableFuture<String> thenComposes =  asyncTask1.thenCompose((String val)->{
                System.out.println("thenComposes thread name:
"+Thread.currentThread().getName());
                 return CompletableFuture.supplyAsync(()->val+"Saxena");
            });

            System.out.println("asyncTask1.get(): "+ asyncTask1.get());
            System.out.println("thenComposes.get(): "+ thenComposes.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }



        poolExecutor.shutdown();

    }
}
output:
asyncTask1 in supplyAsync thread name: pool-1-thread-1
thenComposes thread name: ForkJoinPool.commonPool-worker-1
asyncTask1.get(): Saurav lola
thenComposes.get(): Saurav lola Saxena

Process finished with exit cod
```

thenAccept and thenAcceptAsync:

- Generally end stage, in the chain of Async operations
- It does not return anything.

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
```

```
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());
                try {
            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(()->{
                //This is the task which need to to be completed by thread.
                System.out.println("asyncTask1 in supplyAsync thread name:
"+Thread.currentThread().getName());
                return "Saurav ";
            },poolExecutor)
.thenCompose((val)->CompletableFuture.supplyAsync(()->val+"lola "));// order always
maintained
            //in case of thenComposeAsync we can pass executor.
         CompletableFuture<String> thenComposes =  asyncTask1.thenCompose((String val)->{
                System.out.println("thenComposes thread name:
"+Thread.currentThread().getName());
                 return CompletableFuture.supplyAsync(()->val+"Saxena");
            });
            asyncTask1.thenAccept((val)->System.out.println("All stage complete of asyncTask1
final value is: "+val));
            System.out.println("asyncTask1.get(): "+ asyncTask1.get());
            System.out.println("thenComposes.get(): "+ thenComposes.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }



        poolExecutor.shutdown();

    }
}
output:
asyncTask1 in supplyAsync thread name: pool-1-thread-1
thenComposes thread name: ForkJoinPool.commonPool-worker-1
All stage complete of asyncTask1 final value is: Saurav lola
```

```
asyncTask1.get(): Saurav lola
thenComposes.get(): Saurav lola Saxena

Process finished with exit code 0
```

thenCombine and thenAsync:

- used to combine the result of 2 completable future.

```
public <U,V> CompletableFuture<V> thenCombine(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn) {
    return biApplyStage(null, other, fn);
}
```

```
package Multithreading.ExecutorFrameWork;

import java.util.concurrent.*;

public class RealCompletableFutureExample {
    public static void main(String[] args){
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1,1,1,
                TimeUnit.HOURS,new ArrayBlockingQueue<>(2),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());


         try{
            CompletableFuture<Integer> asyncTask3 = CompletableFuture.supplyAsync(()->{
                return 10;
            },poolExecutor);

            CompletableFuture<String> asyncTask4 = CompletableFuture.supplyAsync(()->{
                return "k";
            });
            CompletableFuture<String> combineFutureObj =  asyncTask3.thenCombine(asyncTask4,
(Integer num,String str)->{
```

```
                return num.toString()+str;
            });

            System.out.println("combineFutureObj.get(): "+combineFutureObj.get());

         }catch (Exception e){
            System.out.println(e.getMessage());
        }


        poolExecutor.shutdown();

    }
}
output:
combineFutureObj.get(): 10k

Process finished with exit code 0
```

Java ForkJoinPool || WorkStealingPool || FixedThreadPool || CachedThreadPool || SingleThreadPool

- Executors[UTILITY CLASS PRESENT IN java.util.concurrent] provides Factory methods which we can use to create Thread Pool Executor.

Fixed ThreadPoolExecutor:

- "newFixedThreadPool" method creates a thread pool executor with a fixed no. of threads.

Min and Max Pool	Same
Queue Size	Unbounded Queue
Thread Alive When Idle	Yes

When to use	Exact Info, how many Async task i needed.
Disadvantage	Not good when worload is heavy, as it will lead to limited concurrency

ExecutorService poolExecutor1 = Executors.newFixedThreadPool(nThreads:5);
poolExectuor1.submit(()→"This is the async task");

Cached ThreadPoolExecutor:
"newCachedThreadPool" methods creates the thread pool that creates a new thread as Needed [fynamically].

Min and Max Pool	Min:0 Max:Interger.MAX_VALUE
Queue Size	Blocking Queue with Size 0 • queue is not being used. • as soon as request has come new thread will create and ne task will assign.
Thread Alive When Idle	60 seconds
When to use	Good for handling burst of short lived tasks.
Disadvantage	Many long lived tasls and submitted rapidly, ThreadPool can create so many threads which might lead to increase memory usage.

CACHED THREAD POOL exector

```
ExecutorService poolExecutor = Executors.newCachedThreadPool();
poolExecutor.submit(()->"This is the async task");
```

Single Thread Exector.

- "newSingleThreadExecutor" creates Exector with just single Worker thread.

Min and Max Pool	Min:1 Max:1
Queue Size	Unbounded Queue
Thread Alive When Idle	Yes
When to use	When need to process tasks sequentially
Disadvantage	No Concureeny at all

VVI

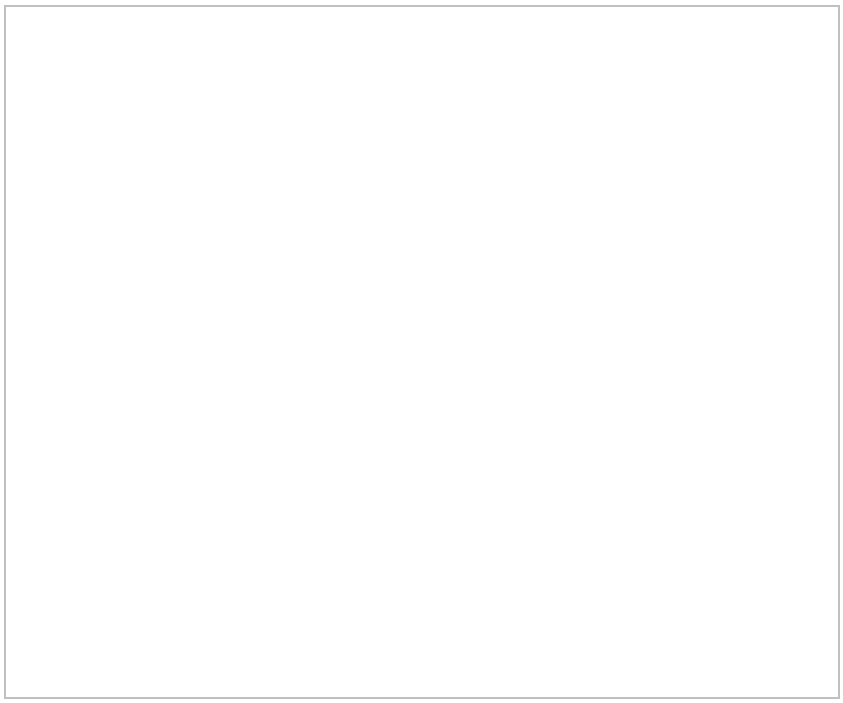
WorkStealing Pool Executor:

- It creates a **FORK-JOIN** Pool Executor.
- Number of threads depends upon the available Processors or we can

```
/**
 * Creates a work-stealing thread pool using the number of
 * {@linkplain Runtime#availableProcessors available processors}
 * as its target parallelism level.
 *
 * @return the newly created thread pool
 * @see #newWorkStealingPool(int)
 * @since 1.8
 */
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool
        (Runtime.getRuntime().availableProcessors(),
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
```

```
}


/**
 * Creates a thread pool that maintains enough threads to support
 * the given parallelism level, and may use multiple queues to
 * reduce contention. The parallelism level corresponds to the
 * maximum number of threads actively engaged in, or available to
 * engage in, task processing. The actual number of threads may
 * grow and shrink dynamically. A work-stealing pool makes no
 * guarantees about the order in which submitted tasks are
 * executed.
 *
 * @param parallelism the targeted parallelism level
 * @return the newly created thread pool
 * @throws IllegalArgumentException if {@code parallelism <= 0}
 * @since 1.8
 */
public static ExecutorService newWorkStealingPool(int parallelism) {
    return new ForkJoinPool
        (parallelism,
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
}
```

- Bring More Parallelism



- There are 2 Queues:

→Submission Queue

→Work - Stealing Queue for each queue for each thread [it's Dequeue]

- Steps:
- If all threads are busy, task would placed in "Submission Queue". [or whenever we call submit method, tasks, tasks goes into submisison queue only]
- Let's say task 1 picked by threadA. And if 2 subtasks created using **fork()** method. Subtask1 will be executed by ThreadA only ans Subtask2 is put into the ThreadA **Deque** [**Work Stealing Queue**]
- If any other thread becomes free, and there is no task in submission queeu, it can "STEAL" the task from the other thread work-stealing queue.
- If Any thread created in pool, each and every thread has it's own **Deque** or **Work Stealing Queue** where we are storing thread subtask.
- If any thread get's free from their task.

→Firstly it's check it's own Work Staling queue if any sub-task left then it will pick that task.

→If Work-stealing Queue is empty then it will goes and check for submission queue if any task is pending in queue thread will pick that task.

→if submission queue is also empty then it will go and check other thread deque or work stealing queue and steal their busu thread sub-task.

- Stealing cannot possible from submission queue.
- Task can split into multiple small sub-tasks. for that Task should extend:
- RecursiveTask

→Whenever Your sub-task want to return something or value.

- RecursiveAction

→Whenever Your sub-task do not return value.

- fork()→WORK STEALING QUEUE→STEALING WORKS. WORKS.
- submit() → SUBMISSION QUEUE→STEALING NOT POSSIBLE
- Thread can consume their sub task from their own deque from front and steal from back some other thread deque sub-task.

```
package Multithreading.Fork_Join_Pool;


import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;

class ComputeSumTask extends RecursiveTask<Integer>{
    int start;
    int end;

    ComputeSumTask(int start,int end){
        this.start=start;
        this.end=end;
    }


    @Override
    protected Integer compute() {
        if(end-start<=4){
            int totalSum = 0;
            for(int i=start;i<=end;i++){
                totalSum+=i;
            }
            return totalSum;
        }else{
            //split
            int mid = (start+end)/2;
            ComputeSumTask leftTask = new ComputeSumTask(start,mid);
            ComputeSumTask rightTask = new ComputeSumTask(mid+1,end);

            //Fork the Subtasks for parallel execution;
```

```
            leftTask.fork();
            rightTask.fork();

            //combine the results of subtasks
            int leftResult = leftTask.join();
            int rightResult = rightTask.join();
            //combine the result
             return leftResult+rightResult;
        }

    }
}


public class ComputeSumTaskUsingRecursiveTask {
    public static void main(String[] args){
        ForkJoinPool pool = ForkJoinPool.commonPool();
        Future<Integer> future = pool.submit(new ComputeSumTask(0,100));

        try
        {
            System.out.println("FINAL RESULT: "+future.get());
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
output:
FINAL RESULT: 5050

Process finished with exit code 0
```

Java ScheduledThreadPoolExecutor || Shutdown Vs AwaitTermination || Multithreading in Java

shutdown vs await Termination vs shutdownNow

Shutdown:

- Initiates orderely shutdown of the ExecutorService.
- After calling "shutdown", Executor will not accept new task submission.
- Already submitted task will continue to execute.

```
public class ComputeSumTaskUsingRecursiveTask {
    public static void main(String[] args){

        ExecutorService poolObj = Executors.newFixedThreadPool(5);
        poolObj.submit(()->{
            try {
                Thread.sleep(5000);
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
            System.out.println("Task completed");

        });

        poolObj.shutdown();
        System.out.println("Main thread completed");

    }
}
output:
Main thread completed
//after 5sec
Task completed

Process finished with exit code 0
```

AwaitTermination.

- it's a optionational functionality. Return true/false.
- It is used after calling 'Shutdown' method.
- Blocks calling thread for specific timeout period, and wait for ExecutorServiceshutdown.
- Return true, if ExecutorService get's shutdown withing specific timeout else false.

```
public class ComputeSumTaskUsingRecursiveTask {
    public static void main(String[] args){
```

```
        ExecutorService poolObj = Executors.newFixedThreadPool(5);
        poolObj.submit(()->{
            try {
                Thread.sleep(5000);
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
            System.out.println("Task completed");

        });

         poolObj.shutdown();


        try {
            boolean isTerminated = poolObj.awaitTermination(2,TimeUnit.SECONDS);
            System.out.println("isTerminated "+isTerminated);

        }catch (Exception e){
            System.out.println(e.getMessage());
        }

        System.out.println("Main thread completed");

    }
}
output:
//after 2 sec
isTerminated false
Main thread completed

//after 5 sec
Task completed

Process finished with exit code 0
```

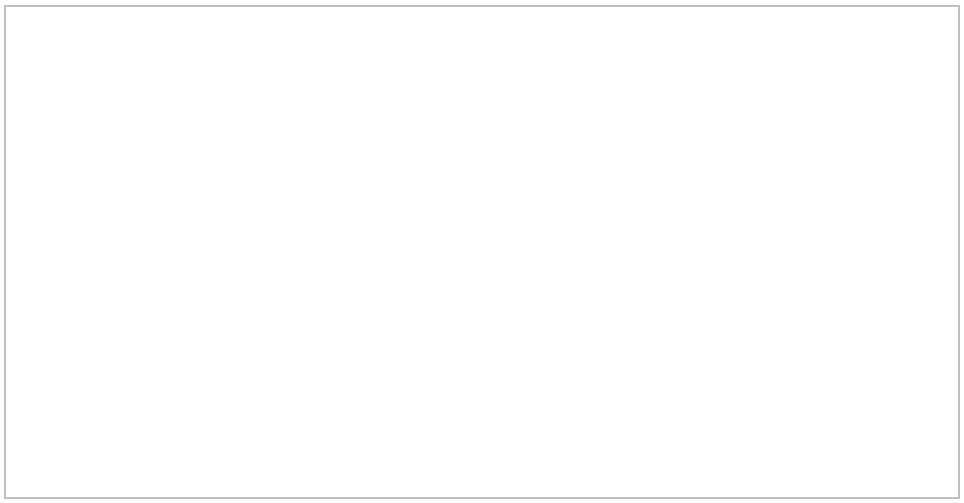
shutdownNow:

- Best effort attempt to stop/intrrrupt the actively executing tasks
- Halt the processing of the tasks which are waiting

- Return the list of tasks which are awaiting execution.

```
package Multithreading.Fork_Join_Pool;


import java.util.concurrent.*;
public class ComputeSumTaskUsingRecursiveTask {
    public static void main(String[] args){


        ExecutorService poolObj = Executors.newFixedThreadPool(5);
        poolObj.submit(()->{
            try {
                Thread.sleep(15000);
            }catch (Exception e){
                 System.out.println("Exception: "+e.getMessage());
            }
            System.out.println("Task completed");

        });

        poolObj.shutdownNow();


        System.out.println("Main thread completed");
    }
}
ouput:
Main thread completed
Exception: sleep interrupted
Task completed

Process finished with exit code 0
```

SchduledThreadPoolExector: Helps to schduled the tasks



Method Name	
schduled(Runnable command, log delay, TimeUni unit)	• Schedules a runnable task after specific delay. • Only one time task runs.
schedule(Callable<v> callable, long delay, TimeUnit unit)	• Schedules a Callable task after specific delay. • Only one time task runs.

```
/**
 * Creates a thread pool that can schedule commands to run after a
 * given delay, or to execute periodically.
 * @param corePoolSize the number of threads to keep in the pool,
 * even if they are idle
 * @return the newly created scheduled thread pool
 * @throws IllegalArgumentException if {@code corePoolSize < 0}
 */
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

/**
 * Creates a new {@code ScheduledThreadPoolExecutor} with the
 * given core pool size.
```

```
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @throws IllegalArgumentException if {@code corePoolSize < 0}
     */
    public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE,
              DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS,
              new DelayedWorkQueue());
    }
```

EXAMPLE:

```
package Multithreading;

import java.util.concurrent.*;

public class ShceduledPoolExecutorExample {

    public static void main(String[] args){
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(5);

        scheduledExecutorService.schedule(()->{
            System.out.println("hello i am From Runnable i am not returning any thing.");
        },5, TimeUnit.SECONDS);

        Future<String> future=scheduledExecutorService.schedule(()->{
            return "hello i am From Callable i am returning String vALUE.";
        },5, TimeUnit.SECONDS);

        try {
            System.out.println("Call callable future.get(): "+future.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }

        scheduledExecutorService.shutdown();
```

```
    }

}
output:
//after 5 sec
hello i am From Runnable i am not returning any thing.
Call callable future.get(): hello i am From Callable i am returning String vALUE.

Process finished with exit code 0
```

schduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

- Schdules a runnable task for repeated execution with fixed rate.
- We can use cancel method to stop this repeated task.
- Also let's say, if thread1 is taiking too much time to complete the task and next task is ready to run , till previous task will not get completed, new task can not be staart [it will wwait in queue]it in queue]

```
package Multithreading;

import java.util.concurrent.*;

public class ShceduledPoolExecutorExample {

    public static void main(String[] args){
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(5);


        Future<?> future=scheduledExecutorService.scheduleWithFixedRate(()->{

            System.out.println("Hello");

        },3,5,TimeUnit.SECONDS);

        try {
            Thread.sleep(10000);
            future.cancel(true);
            System.out.println("Task is cancel abruptly");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
```

```
        }
         scheduledExecutorService.shutdown();
    }

}
output:
Hello
Hello
Task is cancel abruptly

Process finished with exit code 0


Future<?> future=scheduledExecutorService.scheduleAtFixedRate(()->{
    System.out.println("Thread Pick that the task.");
    try {
        Thread.sleep(6000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println("Thread completed the task.");

},3,5,TimeUnit.SECONDS);
//scheduledExecutorService.shutdown();
output:
Thread Pick that the task.
//after 6sec after completion of first task it will take another task.
Thread completed the task.
Thread Pick that the task.

Process finished with exit code 130
```

schduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)

```
Future<?> future=scheduledExecutorService.scheduleWithFixedDelay(()->{
    System.out.println("Thread Pick that the task.");
    try {
        Thread.sleep(6000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
```

```
    }
    System.out.println("Thread completed the task.");

},1,3,TimeUnit.SECONDS);
//scheduledExecutorService.shutdown();
output:
//initial delay for 1 second
Thread Pick that the task.

//sleep for 6 sec
Thread completed the task.

//delay for 3 sec
Thread Pick that the task.

Process finished with exit code 130
```

ThreadLocal VS VirtualThreads.

ThreadLocal

- ThreadLocal class provide access to Thread-Local variables.
- This 'Thread-Local' variable hold the value for particular thread.
- It's type of generic it can hold either string,integer, boolean etc.
- each thread has it's own Thread-Local variable.
- Means each Thread has its own copy of thread -Local Variable.
- We need only 1 object of ThreadLocal class and each thread can use it to set and get it's own Thread - Variable variable.

```
/**
 * Sets the current thread's copy of this thread-local variable
 * to the specified value.  Most subclasses will have no need to
 * override this method, relying solely on the {@link #initialValue}
 * method to set the values of thread-locals.
 *
 * @param value the value to be stored in the current thread's copy of
 *        this thread-local.
 */
public void set(T value) {
    set(Thread.currentThread(), value);
```

```
        if (TRACE_VTHREAD_LOCALS) {
            dumpStackIfVirtualThread();
        }
    }

    private void set(Thread t, T value) {
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            map.set(this, value);
        } else {
            createMap(t, value);
        }
    }

    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
```

about get

```
    /**
     * Returns the value in the current thread's copy of this
     * thread-local variable.  If the variable has no value for the
     * current thread, it is first initialized to the value returned
     * by an invocation of the {@link #initialValue} method.
     *
     * @return the current thread's value of this thread-local
     */
    public T get() {
        return get(Thread.currentThread());
    }

    private T get(Thread t) {
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T) e.value;
                return result;
            }
```

```
    }
    return setInitialValue(t);
}
```

Example:

```
package Multithreading.ThreadLocalVsVirtualThread;

public class ThreadLocalExample {
    public static void main(String[] args){
        ThreadLocal<String> threadLocal = new ThreadLocal<>();

        //mainThread
        threadLocal.set(Thread.currentThread().getName());


        Thread thread1 = new Thread(()->{
            threadLocal.set(Thread.currentThread().getName());
            System.out.println("inside thread1 of task1");
            System.out.println("Inside thread1 threadLocal.get(): "+threadLocal.get());
        });

        thread1.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        System.out.println("Inside main thread threadLocal.get(): "+threadLocal.get());


    }
}
output:
inside thread1 of task1
Inside thread1 threadLocal.get(): Thread-0
```

```
Inside main thread threadLocal.get(): main

Process finished with exit code 0
```

Rember to clean up the thread , If reusing the thread.

- thread1 performing a task, for this you have set threadLocal.set("something relatedd to task1")
- now task 1 completed and we want to assign task 2 so, we need to cleanup the thread.

EXAMPLE Without cleaning Local thread.

```
package Multithreading.ThreadLocalVsVirtualThread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadLocalExample {
    public static void main(String[] args){
        ThreadLocal<String> threadLocal = new ThreadLocal<>();

        ExecutorService poolObj = Executors.newFixedThreadPool(2);
        poolObj.submit(()->{
            threadLocal.set(Thread.currentThread().getName());

        });

        for (int i=1;i<=15;i++){
            poolObj.submit(()->{
                System.out.println(threadLocal.get());
            });
        }
        poolObj.shutdown();
    }
}
output:
pool-1-thread-1
null
null
```

```
null
null
null
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
pool-1-thread-1
null

Process finished with exit code 0
```

Method to clean up the threadLocal.

```
package Multithreading.ThreadLocalVsVirtualThread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadLocalExample {
    public static void main(String[] args){
        ThreadLocal<String> threadLocal = new ThreadLocal<>();


        ExecutorService poolObj = Executors.newFixedThreadPool(2);
        poolObj.submit(()->{
            threadLocal.set(Thread.currentThread().getName());
            //for cleaning the threadLocal variable
            threadLocal.remove();

        });

        for (int i=1;i<=15;i++){
            poolObj.submit(()->{
                System.out.println(threadLocal.get());
            });
```

```
        }
        poolObj.shutdown();
    }
}
output:
null
null
null
null
null
null
null
null
null
null
null
null
null
null
null

Process finished with exit code 0
```

Virtual Thread Vs Platform Thread[normal Thread].

Moto of Virtual Thread:

- To get Higher throughPut not Latency.
- In one Second how many task is going to execute.

```
Thread th1 = Thread.ofVirtual().start(RunnableTask);
Or
ExecutorService myExecutorObj = Executors.newVirtualThreadPerTaskExecutor();
myExecutorObj .submit(new RunnableTask());
```

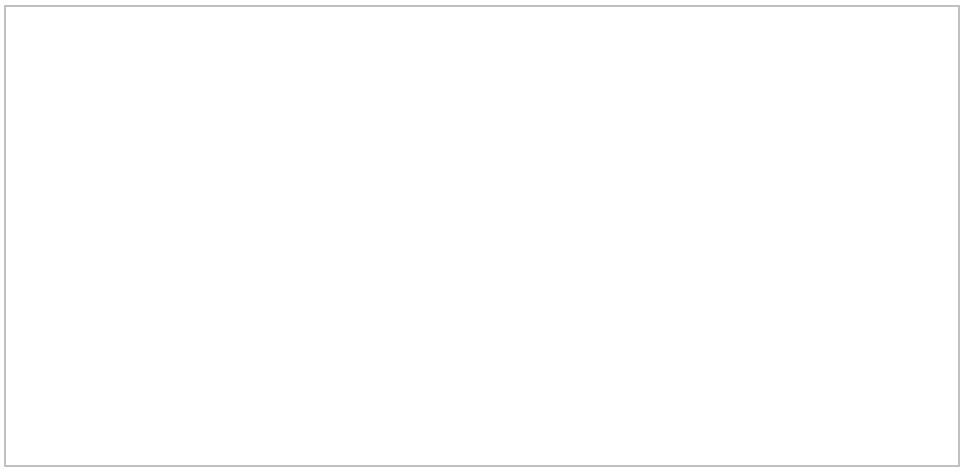
Platform Thread[Normal Thread]

- Platform thread is Just provide a wrapper of **OS [Operating System]** which is provide and manage by **JVM**.
- ONE TO ONE MAPPING.

DisAdvantage

- it's slow , thread creation takes time.
- That's why we are using thread pool executor.

- JVM Call OS to create thread, it is **SYSTTEM CALLS** which takes time→.Expensive TASK
- IN thread pool executor some times we need to create thread, that's system call's not completely removed.
- 2nd advantage→ thread t1 waiting after making DB Call since 4 second, means ultimately OS thread is also waiting because of one to one mapping, OS thread cannot do other task.



To Resolve that problem Virtual Thread comes into picture.

- Mostly available from JDK 19
- We can create as many virtual therads with the help of JVM.
- When virtual thread going to perform some task then it will attach to the OS Thread.
- When virtual thread going into waiting state then JVM detach the link between os thread and virtual thread.
- What ever we are doing with normal thread in Multithreading we can also do same thing with virtual thread.