CPU

The CPU, often referred to as the brain of the computer, is responsible for executing instructions from programs. It performs basic arithmetic, logic, control, and input/output operations specified by the instructions.

A core is an individual processing unit within a CPU. Modern CPUs can have multiple cores, allowing them to perform multiple tasks simultaneously.

A quad-core processor has four cores, allowing it to perform four tasks simultaneously. For instance, one core could handle your web browser, another your music player, another a download manager, and another a background system update.

A program is a set of instructions written in a programming language that tells the computer how to perform a specific task

Microsoft Word is a program that allows users to create and edit documents.

A process is an instance of a program that is being executed. When a program runs, the operating system creates a process to manage its execution.

When we open Microsoft Word, it becomes a process in the operating system.

A thread is the smallest unit of execution within a process. A process can have multiple threads, which share the same resources but can run independently.

A web browser like Google Chrome might use multiple threads for different tabs, with each tab running as a separate thread.

Multitasking allows an operating system to run multiple processes simultaneously. On single-core CPUs, this is done through time-sharing, rapidly switching between tasks. On multi-core CPUs, true parallel execution occurs, with tasks distributed across cores. The OS scheduler balances the load, ensuring efficient and responsive system performance.

Example

We are browsing the internet while listening to music and downloading a file.

Multitasking utilizes the capabilities of a CPU and its cores. When an *operating system* performs multitasking, it can assign different tasks to different cores. This is more efficient than assigning all tasks to a single core.

Multithreading refers to the ability to execute multiple threads within a single process concurrently.

A web browser can use multithreading by having separate threads for rendering the page, running JavaScript, and managing user inputs. This makes the browser more responsive and efficient.

Multithreading enhances the efficiency of multitasking by breaking down individual tasks into smaller sub-tasks or threads. These threads can be processed simultaneously, making better use of the CPU's capabilities.

In a single-core system:
Both threads and processes are managed by the OS scheduler through time slicing and context switching to create the illusion of simultaneous execution.

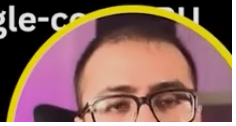
In a multi-core system:
Both threads and processes can run in true parallel on different cores, with the OS scheduler distributing tasks across the cores to optimize performance.

Time Slicing

- Definition: Time slicing divides CPU time into small intervals called time slices or quanta.
- Function: The OS scheduler allocates these time slices to different processes and threads, ensuring each gets a fair share of CPU time.
- Purpose: This prevents any single process or thread from monopolizing the CPU, improving responsiveness and enabling concurrent execution.

Context Switching

- Definition: Context switching is the process of saving the state of a currently running process or thread and loading the state of the next one to be executed.
- Function: When a process or thread's time slice expires, the OS scheduler performs a context switch to move the CPU's focus to another process or thread.
- Purpose: This allows multiple processes and threads to share the CPU, giving the appearance of simultaneous execution on a single-co CPU or improving parallelism on multi-core CPUs.

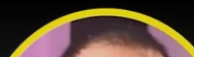


Multitasking can be achieved through multithreading where each task is divided into threads that are managed concurrently.

While multitasking typically refers to the running of multiple applications, multithreading is more granular, dealing with multiple threads within the same app or process.

Multitasking operates at the level of processes, which are the operating system's primary units of execution.

Multithreading operates at the level of threads, which are smaller units within a process

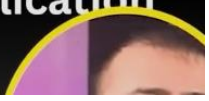


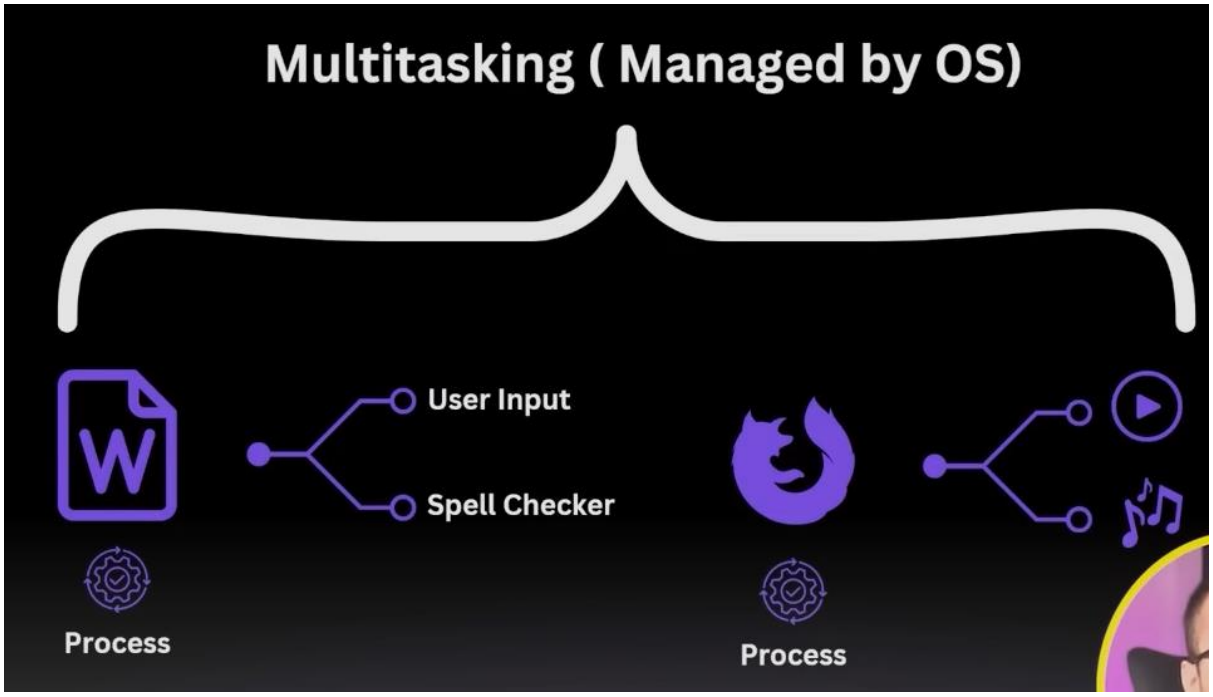
Multitasking involves managing resources between completely separate programs, which may have independent memory spaces and system resources.

Multithreading involves managing resources within a single program, where threads share the same memory and resources.

Multitasking allows us to run multiple applications simultaneously, improving productivity and system utilization.

Multithreading allows a single application to perform multiple tasks at the same time, improving application performance and responsiveness.





In Java, multithreading is the concurrent execution of two or more threads to maximize the utilization of the CPU. Java's multithreading capabilities are part of the java.lang package, making it easy to implement concurrent execution.

In a single-core environment, Java's multithreading is managed by the JVM and the OS, which switch between threads to give the illusion of concurrency.

The threads share the single core, and time-slicing is used to manage thread execution.

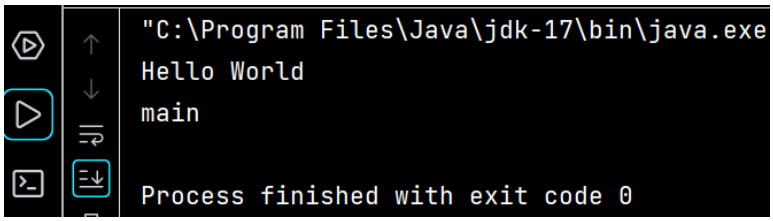
In a multi-core environment, Java's multithreading can take full advantage of the available cores.

The JVM can distribute threads across multiple cores, allowing true parallel execution of thread

A thread is a lightweight process, the smallest unit of processing. Java supports multithreading through its java.lang.Thread class and the java.lang.Runnable interface.

When a Java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program.

```java
public static void main(String[] args) {
    System.out.println("Hello World");
    System.out.println(Thread.currentThread().getName()); // main
}
```



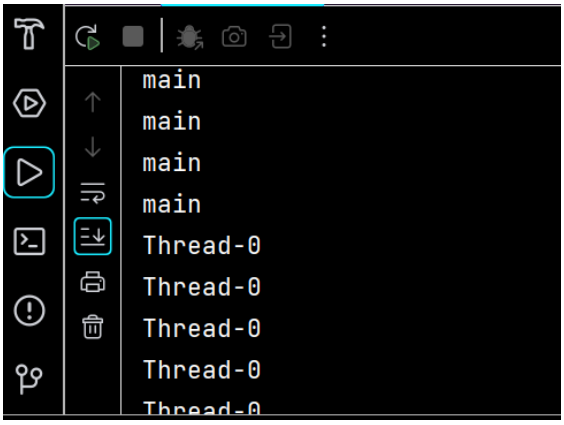
To create a new thread in Java, you can either extend the Thread class or implement the Runnable interface.

- A new class World is created that extends Thread.
- The run method is overridden to define the code that constitutes the new thread.
- start method is called to initiate the new thread

```java
public static void main(String[] args) {
    World world = new World();
    world.start();

    while(true){
        System.out.println(Thread.currentThread().getName());
    }
}
```

```java
public class World extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

O/P -



Method 2 –

```java
public static void main(String[] args) {
    World world = new World();
    Thread t1 = new Thread(world);
    t1.start();

    while(true){
        System.out.println(Thread.currentThread().getName());
    }
}
```

```java
public class World implements Runnable{
    @Override
    public void run() {
        while(true){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

- A new class World is created that implements Runnable.
- The run method is overridden to define the code that constitutes the new thread.
- A Thread object is created by passing an instance of World.
- start method is called on the Thread object to initiate the new thread.

In both cases, the run method contains the code that will be executed in the new thread.

Thread Lifecycle

- New: A thread is in this state when it is created but not yet started.
- Runnable: After the start method is called, the thread becomes runnable. It's ready to run and is waiting for CPU time.
- Running: The thread is in this state when it is executing.
- Blocked/Waiting: A thread is in this state when it is waiting for a resource or for another thread to perform an action.
- Terminated: A thread is in this state when it has finished executing.

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("t1 is running");
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        System.out.println(t1.getState()); // NEW
        t1.start();
        System.out.println(t1.getState()); // RUNNABLE
        Thread.sleep( millis: 100);
        System.out.println(t1.getState()); // TIMED_WAITING

        t1.join(); // main method will wait for t1 to get finished
        System.out.println(t1.getState()); // TERMINATED
    }
```

Thread v/s Runnable

Suppose class A already extends class B in that case we cannot extend it by Thread class also because multiple inheritance is now allowed, in this case class A should implement Runnable interface and override Run() method.

Threads

```
public class MyThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep( millis: 5000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();
        t1.join();
        System.out.println("Hello"); // will get printed after 5 sec
    }
```

Methods –

```
public class MyThread extends Thread{
    public  MyThread(String name){
        super(name);
    }
    @Override
    public void run() {
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName() +
                    " - Priority: " + Thread.currentThread().getPriority() +
                    " - count: " + i);
            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread l = new MyThread( name: "low priority thread");
        MyThread m = new MyThread( name: "medium priority thread");
        MyThread h = new MyThread( name: "high priority thread");
        l.setPriority(Thread.MIN_PRIORITY);
        m.setPriority(Thread.NORM_PRIORITY);
        h.setPriority(Thread.MAX_PRIORITY);

        l.start();
        m.start();
        h.start();
    }
```

In constructor we can pass name of thread.

Using setPriority() we give hints to JVM about relativve im,portance of threads, it is not strictly followed. Maybe thread with low priority execute earlier than threat with medium priority.

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep( millis: 1000);
            System.out.println("Thread is running.......");
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted : " + e);
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
        t1.interrupt(); // stops the threat in whatever state it is
    }
}
```

```
MyThread
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\J
Thread interrupted : java.lang.InterruptedException: sleep interrupted
```

Yield() - A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint.

```java
public class MyThread extends Thread{
    public MyThread(String name){
        super(name);
    }

    @Override
    public void run() {
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName() + " is running");
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread( name: "t1");
        MyThread t2 = new MyThread( name: "t2");
        t1.start();
        t2.start();
    }
}
```

```
Run  MyThread
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Inte
t2 is running
t1 is running
t2 is running
t1 is running
t1 is running
t1 is running
```

User Thread – whatever work we are performing inside run() are done with help of user threads.

Daemon Thread – they run in background. Ex- garbage collector in Java. JVM don't wait for their execution to end. As soon as main thread, all other user threads are done, daemon threads are also terminated and JVM terminates.

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println("Hello world !!");
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setDaemon(true);
        t1.start();
        System.out.println("main done");
    }
}
```

```
MyThread
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent
main done
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!
Hello world !!

Process finished with exit code 0
```

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println("Hello world !!");
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setDaemon(true);
        t1.start();
        MyThread userThread = new MyThread();
        userThread.start();
        System.out.println("main done");
    }
}
```

In this case both threads keep running and JVM will wait for userThread to finish.

Synchronization

```java
public class Counter{
    private int count = 0;
    public  void increment(){
        count++;
    }
    public int getCount(){
        return count;
    }
}
```

```java
public class MyThread extends Thread{
    private Counter counter;
    public MyThread(Counter counter){
        this.counter = counter;
    }

    @Override
    public void run() {
        for(int i=0;i<1000;i++){
            counter.increment();
        }
    }
```

```java
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);
        t1.start();
        t2.start();
        try{
            t1.join();
            t2.join(); // finish both
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(counter.getCount());  // 1845, but we expect 2000
        // becoz both threads run simultaneously and both share common object
        // suppose count = 101 now both threads read at same time and increment
        // then count becomes 102

        // To make sure that at a particular instance only 1 thread can access the method
        // we make that method as synchronized, this will make other threads to wait
    }
```

```java
public class Counter{
    private int count = 0;
    public synchronized void increment(){
        count++;
    }
    public int getCount(){
        return count;
    }
}
```

Now O/P – 2000

Instead of whole method we can make a particular block synchronized –

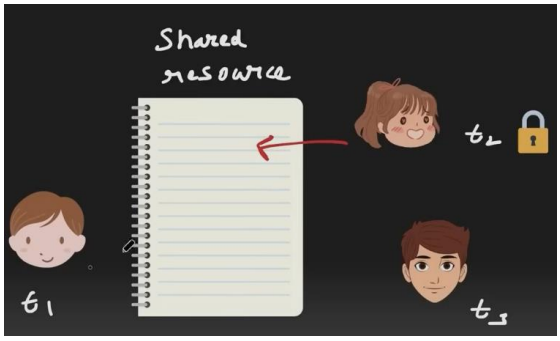
```java
public class Counter{
    private int count = 0;
    public void increment(){
        synchronized (this){ // "this" signifies that if multiple threads try to access
                             // same instance of class, then only 1 thread should be
                             // allowed to access this block
                             // if we have multiple objects both will run independently
            count++;
        }
    }
    public int getCount(){
        return count;
    }
}
```

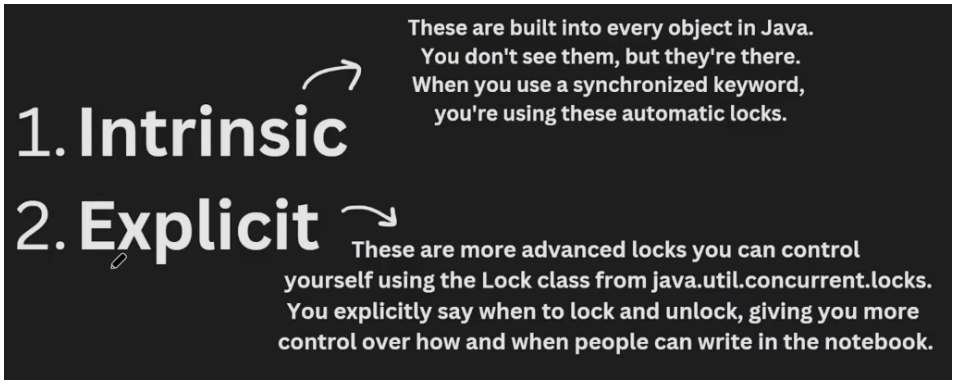
Critical section – Part of our code where shared resources are accessed and modified. Ex- increment() method in above example.

Race Condition – when multiple threads are working on shared resources and if their timing conflicts it can lead to unexpected results.

Mutual Exclusion – condition when only 1 thread can access the critical section

Locks – after using synchronized keyword on method we achieve mutual exclusion it is done by locking. When 1 thread is working on shared resource it puts lock on it and when it is done, it releases the lock.





Synchronized – when a method is declared with synchronized keyword it means this method will acquire the intrinsic lock of the object the method belongs to.

Issue with intrinsic lock is that suppose t1 is running and it puts a lock and it gets stuck due to infinite loop, then other threads attempting to acquire the same lock will be **blocked indefinitely**. Explicit locking can solve it.

Lock is interface and ReentranLock is its impl class.

Lock.lock() → thread will wait for acquiring lock in BLOCKED state, just like synchronized.

.tryLock() → Acquires the lock if it is available and returns immediately with the value true. If the lock is not available then this method will return immediately with the value false.

.tryLock(**1000,** TimeUnit.*MILLISECONDS*) → Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted. If the lock is available this method returns immediately with the value true.

.unlock → it is run by a thread which has already acquired a lock and now want to release it.

Note :- The reason why Sonar suggests to re-interrupt the thread is because by the time the exception occurs, the thread's interrupted state may have been cleared, so if the exception is not handled properly the fact that the thread was interrupted will be lost.

```
Thread t = new Thread(()->{
    try {
        Thread.sleep( millis: 2000);
    } catch (InterruptedException e) {
        if(Thread.currentThread().isInterrupted()){
            System.out.println("thread is interrupted");
        }else{
            System.out.println("interrupted flag is reset");
        }
    }
});
t.start();
t.interrupt();
```

O/P – interrupted flag is reset

```java
public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount)  {
        System.out.println(Thread.currentThread().getName() + " attemp to withdraw " + amount );
        try {
            if(lock.tryLock( time: 1000, TimeUnit.MILLISECONDS)){
                if(balance >= amount){
                    try {
                        System.out.println(Thread.currentThread().getName() + " proceeding with withdrawal");
                        Thread.sleep( millis: 10000); // stuck in very long computation
                        balance -= amount;
                        System.out.println(Thread.currentThread().getName() + " completed withdrawal, " +
                                                                "Remaining Balance: " + balance);
                    }catch (InterruptedException e){
                        Thread.currentThread().interrupt(); // good practice to re-interrupt
                        // we manually restore state so to run some cleanup code
                    }
                    finally {
                        lock.unlock();
                    }
                }else{
                    System.out.println(Thread.currentThread().getName() + " insufficient balance");
                }
            }else{
                System.out.println(Thread.currentThread().getName() + " could not acquire the lock, will try later");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        if(Thread.currentThread().isInterrupted()){
            // do cleanup
        }
    }
```

```java
public static void main(String[] args) {
    BankAccount sbi = new BankAccount();
    Runnable task = new Runnable() {  // anonymous class
        @Override
        public void run() {
            sbi.withdraw( amount: 50);
        }
    };
    Thread t1 = new Thread(task,  name: "t1");
    Thread t2 = new Thread(task,  name: "t2");
    t1.start();
    t2.start();
```

O/P -

```
t1 attemp to withdraw 50
t2 attemp to withdraw 50
t1 proceeding with withdrawal
t2 could not acquire the lock, will try later
t1 completed withdrawal, Remaining Balance: 50
```

Deadlock Prevention in ReentrantLock

```java
public class ReentrantExample {
    private final Lock lock = new ReentrantLock();
    public void outerMethod(){
        lock.lock();
        try{
            System.out.println("Outter method");
            innerMethod();
        }finally {
            // lock will be released when every lock() is matched with its unlock()
            lock.unlock();
        }
    }
    private void innerMethod() {
        lock.lock(); // here main thread will wait for himself only to release lock
                     // acquired in outer method, this is deadlock.
                     // inner method waiting for outer and outer waiting for inner to finish
        // Since the lock is Reentrant so we won't get any deadlock
        // Main thread will acquire lock becoz same thread holds it.
        try{
            System.out.println("Inner method");
        }finally {
            // When we use Reentrant lock, then a count is maintained for how many times
            // we have acquired the lock and unlock it, each lock call must be paired with unlock call
            lock.unlock();  // after this statement, lock is still acquired becoz count is still 1 for lock call
        }
    }

    public static void main(String[] args) {
        ReentrantExample example = new ReentrantExample();
        example.outerMethod();
    }
```

Scenario Setup

- A **lock** is being used to ensure that only one thread accesses a critical section at a time.
- Thread `t1` acquires the lock first and holds it for some time.
- Thread `t2` tries to acquire the lock while `t1` is still holding it.

---

1. Using `lock.lock()`

- `t2` will **wait indefinitely** until `t1` releases the lock.
- If `t2` is interrupted while waiting, it will **ignore the interrupt** and keep waiting for the lock.

2. Using `lock.lockInterruptibly()`

- `t2` will **wait** for the lock like before, but it will **respond to an interrupt** while waiting.
- If `t2` is interrupted, it will stop waiting for the lock and throw an `InterruptedException`.

```java
public class ReentrantExample {
    private final Lock lock = new ReentrantLock();
    public void outerMethod(){
        try {
            lock.lockInterruptibly();
            try{
                System.out.println("Outter method");
                innerMethod();
            }finally {
                // lock will be released when every lock() is matched with its unlock()
                lock.unlock();
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
```

Lock's Fairness

- A fair lock ensures that threads acquire the lock in the order they requested it (i.e., a first-come, first-served basis)
- A thread will not starve (i.e., will not have to wait indefinitely) because threads that arrived earlier in the queue are guaranteed to get the lock before any new threads.
- A non-fair lock (when fairness is set to false) does not guarantee order

```java
public class FairLockExample {
    private final Lock lock = new ReentrantLock( fair: true);
    public void accessResource(){
        lock.lock();
        try{
            System.out.println(Thread.currentThread().getName() + " acquired the lock");
            Thread.sleep( millis: 1000);
        }catch (InterruptedException e){
            Thread.currentThread().interrupt();
        }finally {
            System.out.println(Thread.currentThread().getName() + " released the lock");
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        FairLockExample example = new FairLockExample();
        Runnable task = new Runnable() {
            @Override
            public void run() {
                example.accessResource();
            }
        };

        Thread t1 = new Thread(task,  name: "t1");
        Thread t2 = new Thread(task,  name: "t2");
        Thread t3 = new Thread(task,  name: "t3");

        try {
            t1.start();
            Thread.sleep( millis: 50);
            t2.start();
            Thread.sleep( millis: 50);
            t3.start();
        }catch (InterruptedException e){
        }
    }
```

Problems with Synchronized –

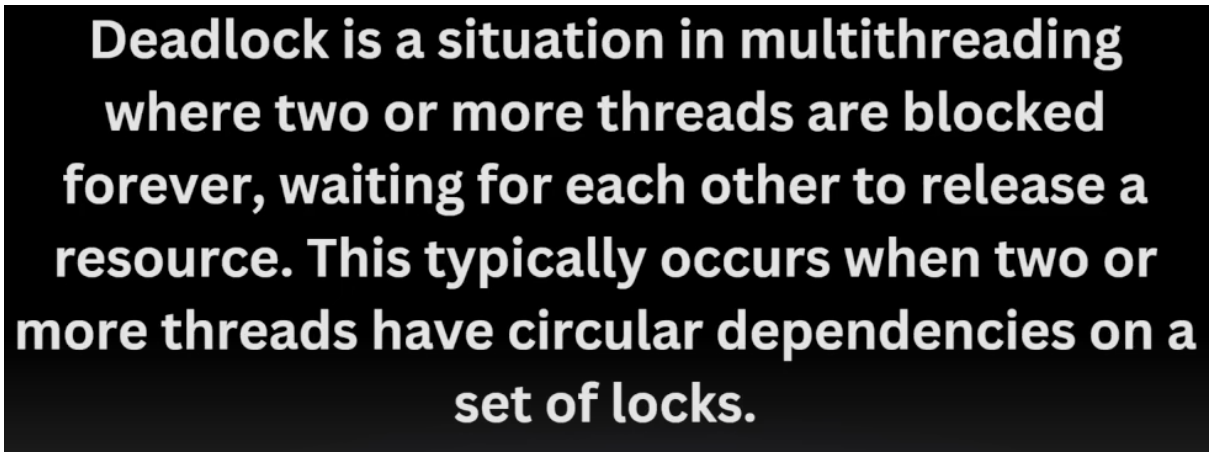
- No fairness
- indefinitely blocking to acquire lock
- No Interruptibility
- No distinguish between read /write operations – synchronized will block every thread.

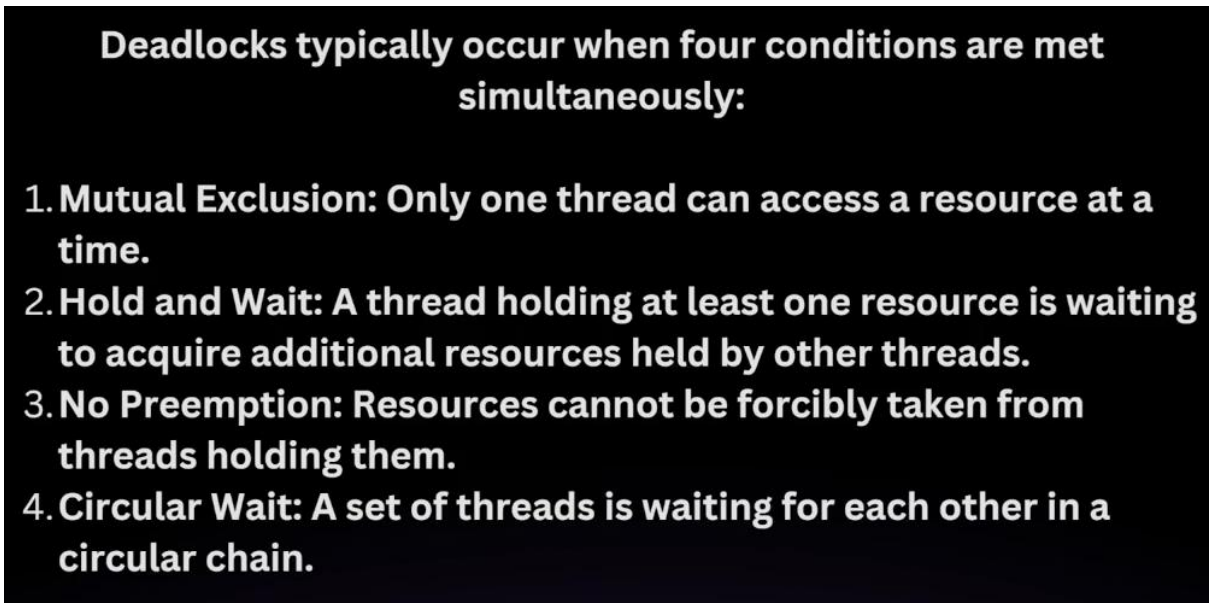
ReadWriteLock – it allow multiple threads to read resources concurrently as long as no thread is writing to it. It ensures exclusive access for write operations.

```java
public class ReadWriteCounter {
    private int count = 0;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    public void increment(){
        writeLock.lock();
        try{
            count++;
        }finally {
            writeLock.unlock();
        }
    }

    public int getCount(){
        readLock.lock(); // multiple threads can acquire this lock when
                         // writeLock has not been acquired by some another thread
        try{
            return count;
        }finally {
            readLock.unlock();
        }
    }
```

```java
public static void main(String[] args) throws InterruptedException {
    ReadWriteCounter counter = new ReadWriteCounter();
    Runnable writeTask = new Runnable() {
        @Override
        public void run() {
            counter.increment();
        }
    };
    Runnable readTask = new Runnable() {
        @Override
        public void run() {
            counter.getCount();
        }
    };

    Thread writerThread = new Thread(writeTask);
    Thread readThread1 = new Thread(readTask);
    Thread readThread2 = new Thread(readTask);

    writerThread.start();
    readThread1.start();
    readThread2.start();

    writerThread.join();
    readThread1.join();
    readThread2.join();

    System.out.println(counter.getCount()); // 1
}
```

Deadlock


Deadlock is a situation in multithreading where two or more threads are blocked forever, waiting for each other to release a resource. This typically occurs when two or more threads have circular dependencies on a set of locks.


Deadlocks typically occur when four conditions are met simultaneously:
1. Mutual Exclusion: Only one thread can access a resource at a time.
2. Hold and Wait: A thread holding at least one resource is waiting to acquire additional resources held by other threads.
3. No Preemption: Resources cannot be forcibly taken from threads holding them.
4. Circular Wait: A set of threads is waiting for each other in a circular chain.

```java
class Pen{
    public synchronized void writeWithPenAndPaper(Paper paper){
        System.out.println(Thread.currentThread().getName() + " is using pen " + this + " and trying to write using paper " + paper);
        paper.finishWriting();
    }
    public synchronized void finishWriting(){
        System.out.println(Thread.currentThread().getName() + " finished using pen " + this);
    }
}
class Paper{
    public synchronized void writeWithPenAndPaper(Pen pen){
        System.out.println(Thread.currentThread().getName() + " is using paper " + this + " and trying to write using pen " + pen);
        pen.finishWriting();
    }
    public synchronized void finishWriting() {
        System.out.println(Thread.currentThread().getName() + " finished using paper " + this);
    }
}

class Task1 implements Runnable{
    private Pen pen;
    private Paper paper;
    public Task1(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }
    @Override
    public void run() {
        pen.writeWithPenAndPaper(paper); // thread1 locks pen and tries to lock paper
    }
}
```

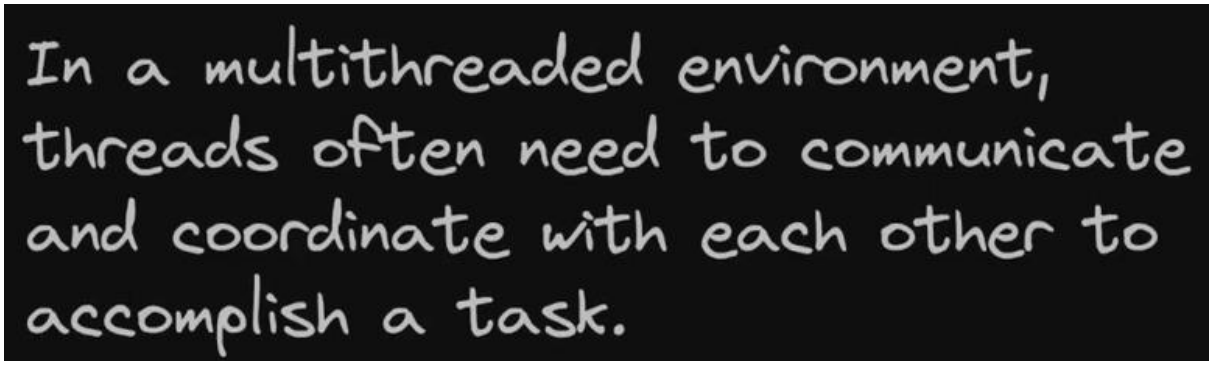
```java
class Task2 implements Runnable{
    private Pen pen;
    private Paper paper;
    public Task2(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }
    @Override
    public void run() {
        paper.writeWithPenAndPaper(pen); // thread2 locks paper and tries to lock pen
    }
}
public class Deadlock {
    public static void main(String[] args) {
        Pen pen = new Pen();
        Paper paper = new Paper();
        Thread t1 = new Thread(new Task1(pen, paper), name: "t1");
        Thread t2 = new Thread(new Task2(pen, paper), name: "t2");
        t1.start(); // it will acquire the lock of pen
        t2.start(); // it will acquire the lock of paper
        // when t1 try to call paper.finishWriting(); then it should first get lock of paper which is  acquired by t2
        // when t2 try to call pen.finishWriting(); then it should first get lock of pen which is acquired by t1
        // deadlock situation is created
    }
}
```

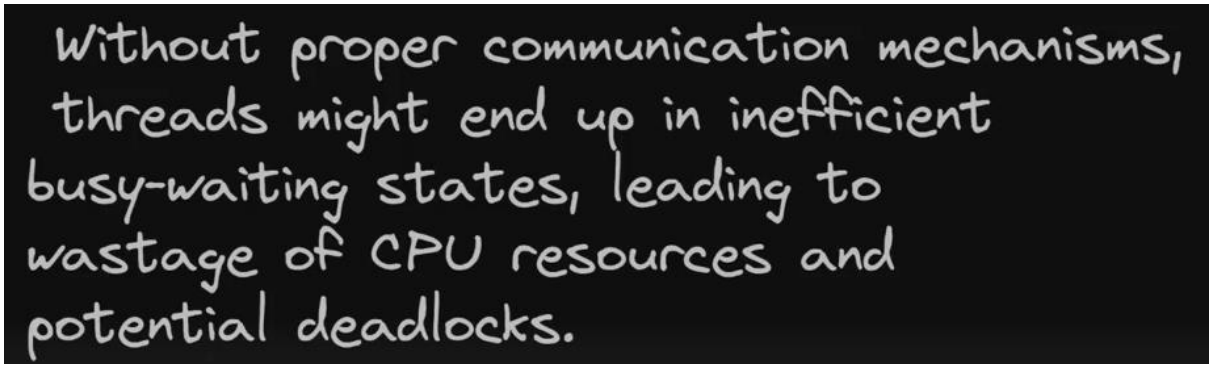
To avoid deadlock ensure that all the threads acquire locks in consistent order.

```java
class Task2 implements Runnable{
    private Pen pen;
    private Paper paper;
    public Task2(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }
    @Override
    public void run() {
        synchronized (pen){  // acquire lock of pen first
            paper.writeWithPenAndPaper(pen); // thread2 locks paper
        }
    }
}
```

This will ensure that t2 will run only when it locks both pen and paper

Thread Communication





For thread communication, we have methods like wait(), notify() and notifyAll(). These methods can only be called within a synchronized context.

Wait() → tells current thread to release the lock and wait until some other thread run notify() or notifyAll()

Notify() → wakes up a single thread that is waiting

NofityAll() → wakes up all threads that are waiting

Ex-

We want Producer to produce data but wait if hasData is true. And Consumer should consume data when hasData is true.

```java
class SharedResource{
    private int data;
    private boolean hasData;
    public synchronized void produce(int value){ // we have acquired lock on object of SharedResource
        while(hasData){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data = value;
        hasData = true;
        System.out.println("Produced : " + value);
        notify();  // it will notify the different thread who is waiting & trying to lock the same object of SharedResource
        // notifyAll(); -------> if for than 1 thread are waiting
    }
    public synchronized int consume(){
        while(!hasData){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        hasData = false;
        System.out.println("Consumed: " + data);
        notify();
        return data;
    }
}
```

```java
class Producer implements Runnable{
    private SharedResource resource;
    public Producer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for(int i=0;i<3;i++) {
            resource.produce(i);
        }
    }
}
class Consumer implements Runnable{
    private SharedResource resource;
    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for(int i=0;i<3;i++) {
            resource.consume();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread producerThread = new Thread(new Producer(resource));
        Thread consumerThread = new Thread(new Consumer(resource));
        producerThread.start();
        consumerThread.start();
    }
```

O/P -

```
Produced : 0
Consumed: 0
Produced : 1
Consumed: 1
Produced : 2
Consumed: 2

Process finished with exit code 0
```

Thread Safety – a code is thread safe if it guarantee no unexpected results, no race condition when multiple threads try to access that piece of code with same object.

Lambda Expression – anonymous function

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
};
Runnable task1 = ()-> System.out.println("Hello");
Thread t1 = new Thread(task1);
Thread t2 = new Thread(()-> System.out.println("Hello"));
```

```
@FunctionalInterface
public interface Runnable {
    When an object implementing interface Runnable is used to create a thread, starting the thread
    causes the object's run method to be called in that separately executing thread.
    The general contract of the method run is that it may take any action whatsoever.
    See Also: Thread.run()
    public abstract void run();
}
```

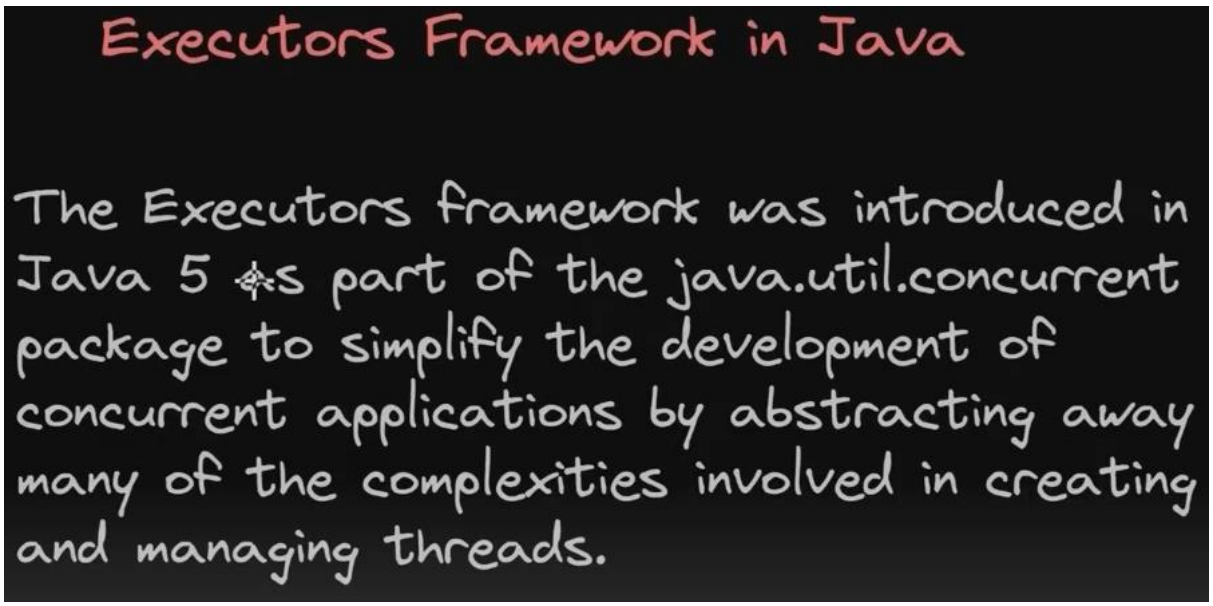
It does not return.

Thread Pool – collection of pre-initialised threads that are ready to perform a task.

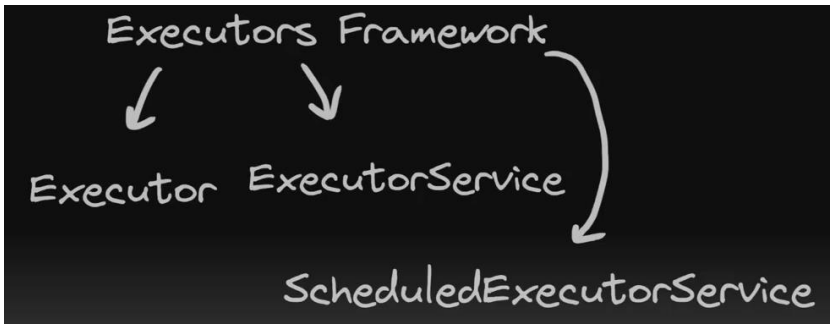
Why ?

1. Resource management – every time creating and destroying threads is expensive.
2. Response time improves
3. Control over thread count – puts a limit o no of threads that can be created.

Executors Framework



Problems prior to executor framework - 

3 interfaces available – 


```
long startTime = System.currentTimeMillis(); // 1 Jan 1970
for(int i=1;i<10;i++){
    System.out.println(factorial(i));
}
System.out.println("Total Time : " + (System.currentTimeMillis() - startTime));
```

O/P- Total Time : 4575 ms

Using multithreading ➔ O/P – 515 ms

```
long startTime = System.currentTimeMillis(); // 1 Jan 1970
Thread[] threads = new Thread[9];
for(int i=1;i<10;i++){
    int finalI = i;
    threads[i-1] = new Thread(()->{
        long result = factorial(finalI);
        System.out.println(result);

    });
    threads[i-1].start();
}
for(Thread thread: threads){
    try{
        thread.join();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
System.out.println("Total Time : " + (System.currentTimeMillis() - startTime));
```

Using Executor Framework

```
long startTime = System.currentTimeMillis(); // 1 Jan 1970
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 3);
for(int i=1;i<10;i++){
    int finalI = i;
    executor.submit(()->{
        long result = factorial(finalI);
        System.out.println(result);
    });
}
executor.shutdown(); // Initiates an orderly shutdown in which previously submitted tasks are executed,
// but no new tasks will be accepted.

try {
    executor.awaitTermination( timeout: 10, TimeUnit.SECONDS);
    // Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs,
    // or the current thread is interrupted, whichever happens first.
    // timeout - we pass maximum time to wait
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
System.out.println("Total Time : " + (System.currentTimeMillis() - startTime));
```

```
try {
    // waits infinitely
    // Returns: true if this executor terminated and false if the timeout elapsed before termination
    while(!executor.awaitTermination( timeout: 1, TimeUnit.MILLISECONDS)){
        System.out.println("Waiting...");
    }
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

When you call submit(), the task is queued and will execute automatically if a thread from the pool is available. Calling shutdown() signals the executor to stop accepting new tasks and to complete the already submitted tasks, it does not block the current thread.

```
public interface ExecutorService extends Executor {
```

```
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 Executor implementation.
    Params: command – the runnable task
    Throws: RejectedExecutionException – if this task cannot be accepted for execution
            NullPointerException – if command is null

    void execute( @NotNull Runnable command);
}
```

Executor does not contain methods like shutdown() so u need to manually do it. That's why its useless. We'll use ExecutorService becoz it provides more features.

Submit() method returns Future<T>

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> 42);
System.out.println(future.get()); // get() waits if necessary for the computation to complete,
                                  // and then retrieves its result.

if(future.isDone()){ // Returns true if this task completed. Completion may be due to
    // normal termination, exception, or cancellation -- in all of these cases, this method will return true.
    System.out.println("Task is done");
}
executorService.shutdown();
```

O/P -

```
42
Task is done
```

Submit() can take Runnable as well as Callable –

```
@NotNull Callable<Integer> task
@NotNull Runnable task, Integer result
@NotNull Runnable task
```

```
@FunctionalInterface
public interface Callable<V> {
    Computes a result, or throws an exception if unable to do so.
    Returns: computed result
    Throws: Exception – if unable to compute a result
    V call() throws Exception;
}
```

Callable returns value.

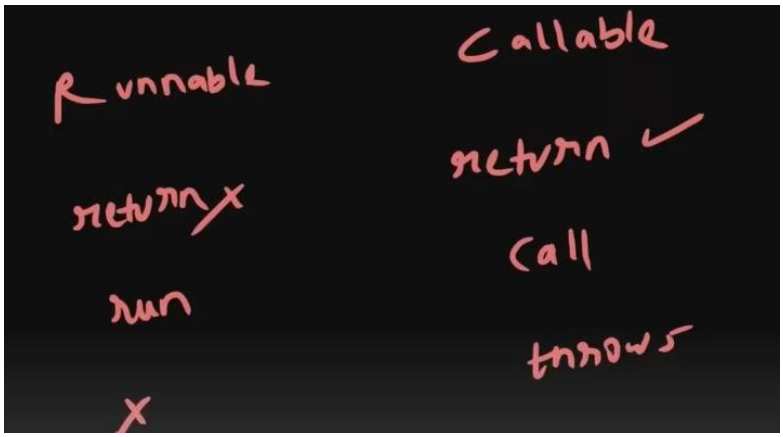
```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Callable<String> callable = ()->"Hello";
Future<String> future = executorService.submit((callable));
System.out.println(future.get()); // Hello
executorService.shutdown();
```

submit() method of Runnable argument is called -

```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<?> future = executorService.submit(()-> System.out.println("Hello"));
System.out.println(future.get()); // null
executorService.shutdown();
```

```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String> future = executorService.submit(()-> System.out.println("Hello"), result: "success");
//Submits a Runnable task for execution and returns a Future representing that task.
// The Future's get method will return the given result upon successful completion.
```

Difference -




```java
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
Future<Integer> future = executorService.submit(() -> 1 + 2);
Integer i = future.get(); // main thread will wait to get its result but execution done by threads of threadpool
System.out.println(i);

executorService.shutdown();
System.out.println(executorService.isShutdown()); // true
Thread.sleep( millis: 10); // give some time for termination
System.out.println(executorService.isTerminated()); // true
```

```java
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
List<Callable<Integer>> list = Arrays.asList(()->{
    System.out.println("Task 1");
    return 1;
}, ()->{
    System.out.println("Task 2");
    return 2;
}, ()->{
    System.out.println("Task 3");
    return 3;
});
List<Future<Integer>> futures = executorService.invokeAll(list);
// Executes the given tasks, returning a list of Futures holding their status and results when all complete
// it blocks the main thread and wait for completion of all tasks
for(Future<Integer> f:futures){
    System.out.println(f.get());
}
executorService.shutdown();
System.out.println("Hello");
```

O/P →

```
Task 2
Task 1
Task 3
1
2
3
Hello
```

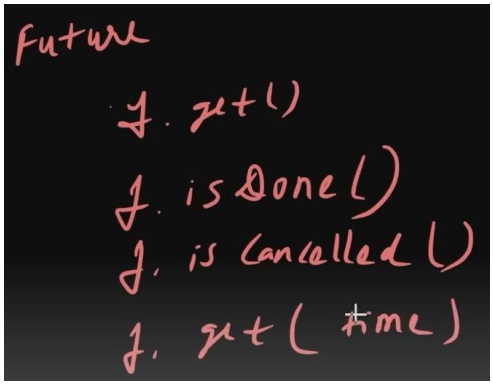
```java
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
List<Callable<Integer>> list = Arrays.asList(()->{
    Thread.sleep( millis: 1000);
    System.out.println("Task 1");
    return 1;
}, ()->{
    Thread.sleep( millis: 1000);
    System.out.println("Task 2");
    return 2;
}, ()->{
    Thread.sleep( millis: 2000);
    System.out.println("Task 3");
    return 3;
});
List<Future<Integer>> futures = executorService.invokeAll(list, timeout: 1, TimeUnit.SECONDS);
```

O/P →

```
Task 1
Task 2
```

When we pass timeout in constructor, then it executes all the task which can be executed within time frame and other tasks are cancelled.

```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
List<Callable<Integer>> list = Arrays.asList(()->1, ()->2, ()->3);
Integer i = executorService.invokeAny(list); //Executes the given tasks, returning the result of one that has
                                // completed successfully (i.e., without throwing an exception), if any do
System.out.println(i); // 1
executorService.shutdown();
```



```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000);
    } catch (InterruptedException e) { }
    return 42;
});
Integer i = null;
try{
    i = future.get( timeout: 1, TimeUnit.SECONDS);
    System.out.println(future.isDone());
    System.out.println(i);
}catch (InterruptedException | ExecutionException | TimeoutException e) {
    System.out.println("Exception occurred: " + e);
}
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED
Exception occurred: java.util.concurrent.TimeoutException
```

Explaination :- It waits for 1 sec, if task get completed then fine otherwise it gives TimeoutException.

```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000);
    } catch (InterruptedException e) { }
    return 42;
});
future.cancel( mayInterruptIfRunning: true);
System.out.println(future.isCancelled());
System.out.println(future.isDone());
executorService.shutdown();
```

O/P →

```
true
true
```

```java
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000);
    } catch (InterruptedException e) { }
    System.out.println("Hello");
    return 42;
});
future.cancel( mayInterruptIfRunning: false); // task will not be interrupted, it will still run
System.out.println(future.isCancelled());
System.out.println(future.isDone());
executorService.shutdown();
```

O/P →

```
true
true
Hello
```

Scheduled Executor Service – used when we want something to run periodically.

```java
public interface ScheduledExecutorService extends ExecutorService {
```

```java
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 1);
scheduler.schedule(
        ()-> System.out.println("Task executed after 5 second delay"),
        delay: 5,
        TimeUnit.SECONDS);
scheduler.shutdown();
```

O/P →

```
Task executed after 5 second delay
```

On running .schedule() task will immediately go inside queue for execution and scheduler will wait for it to finish then shutdown.

But on running .scheduleAtFixedRate() it will not immediately come into queue. If scheduler.shutdown() is called **before the initial delay expires**, the scheduler will terminate immediately, and the task will never run. So we need to handle it properly –

```
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 1);
        scheduler.scheduleAtFixedRate(
                ()-> System.out.println("Task executed after 5 second delay"),
                initialDelay: 0,
                period: 5,
                TimeUnit.SECONDS);
        scheduler.schedule(()->{
            System.out.println("Initiating shutdown...");
            scheduler.shutdown();
        }, delay: 20, TimeUnit.SECONDS);
    }
```

```
Main ×
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-javaager
Task executed after 5 second delay
Task executed after 5 second delay
Task executed after 5 second delay
Task executed after 5 second delay
Task executed after 5 second delay
Initiating shutdown...
```

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 1);
ScheduledFuture<?> scheduledFuture = scheduler.scheduleWithFixedDelay(
        () -> System.out.println("Task executed after 5 second delay"),
        initialDelay: 0,
        delay: 5, // how much time to wait after completing 1 task to start the next task
        TimeUnit.SECONDS);

scheduler.schedule(()->{
    System.out.println("Initiating shutdown...");
    scheduler.shutdown();
}, delay: 20, TimeUnit.SECONDS);
```

Cached Thread Pool – It creates a thread pool where it creates new threads as per need and terminates then after 60 seconds of inactivity. It dynamically adjusts the pool size. It is useful when load is variable and short lived. Do not use it when task are very time consuming.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

CountDownLatch – It allows one or more threads to wait for a set of operations being performed by other threads to complete before proceeding further.

Ex- we want to achieve that our main thread wait for some other dependent services.

Without –

```
class DependentService implements Callable<String>{
    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " service started");
        Thread.sleep( millis: 2000);
        return "ok";
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 3);
        Future<String> future1 = executorService.submit(new DependentService());
        Future<String> future2 = executorService.submit(new DependentService());
        Future<String> future3 = executorService.submit(new DependentService());
        future1.get();
        future2.get();
        future3.get();
        System.out.println("All dependent services finished. Starting main service.....");
        executorService.shutdown();
```

With CountDownLatch -

```
class DependentService implements Callable<String>{
    CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public String call() throws Exception {
        try {
            System.out.println(Thread.currentThread().getName() + " service started");
            Thread.sleep( millis: 2000);
        }finally {
            latch.countDown();
        }
        return "ok";
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
        CountDownLatch latch = new CountDownLatch(numberOfServices);
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        latch.await();

        System.out.println("All dependent services finished. Starting main service.....");
        executorService.shutdown();
```

O/P →

```
pool-1-thread-1 service started
pool-1-thread-2 service started
pool-1-thread-3 service started
All dependent services finished. Starting main service.....
```

Using threads –

```
class DependentService implements Runnable{
    CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run()  {
        try {
            System.out.println(Thread.currentThread().getName() + " service started");
            Thread.sleep( millis: 2000);
        }catch (InterruptedException e) { }
        finally {
            latch.countDown();
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        CountDownLatch latch = new CountDownLatch(numberOfServices);
        for(int i=0;i<numberOfServices;i++){
            new Thread(new DependentService(latch)).start();
        }
        latch.await();
        System.out.println("All dependent services finished. Starting main service.....");
```

O/P →

```
Thread-2 service started
Thread-1 service started
Thread-0 service started
All dependent services finished. Starting main service.....
```

We can also pass timeout in await()

```
class DependentService implements Runnable{
    CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run(){
        try {
            Thread.sleep( millis: 6000);
            System.out.println(Thread.currentThread().getName() + " service started");
        }catch (InterruptedException e) {}
        finally {
            latch.countDown();
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        CountDownLatch latch = new CountDownLatch(numberOfServices);
        for(int i=0;i<3;i++){
            new Thread(new DependentService(latch)).start();
        }
        latch.await( timeout: 5, TimeUnit.SECONDS);  // wait for 5 seconds and start executing main thread

        System.out.println("Main thread executing....");
```

O/P →

```
Main thread executing....
Thread-1 service started
Thread-2 service started
Thread-0 service started
```

```
public static void main(String[] args) throws ExecutionException, InterruptedException

    int numberOfServices = 3;
    ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
    CountDownLatch latch = new CountDownLatch(numberOfServices);
    executorService.submit(new DependentService(latch));
    executorService.submit(new DependentService(latch));
    executorService.submit(new DependentService(latch));
    latch.await( timeout: 5, TimeUnit.SECONDS);
    System.out.println("Main");
    executorService.shutdownNow();
```

.shutdownNow() will terminate the worker threads immediately.

CyclicBarrier – It is reusable

CountDownLatch is not reusable, once the count reaches 0 it cannot be reset.

In cyclic barrier, multiple threads wait at a point until all threads reached at that point. When the last thread reaches the barrier then all threads are released. It does not block main thread. It is used to make sure that certain no of threads reach at a particular point before any of them proceed.

```
class DependentService implements Callable<String>{
    CyclicBarrier barrier;
    public DependentService(CyclicBarrier barrier) {
        this.barrier = barrier;
    }
    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " service started");
        Thread.sleep( millis: 1000);
        System.out.println(Thread.currentThread().getName() + " is waiting at the barrier");

        barrier.await(); // all worker threads wait here.
        return "ok";
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
        CyclicBarrier barrier = new CyclicBarrier(numberOfServices);
        executorService.submit(new DependentService(barrier));
        executorService.submit(new DependentService(barrier));
        executorService.submit(new DependentService(barrier));
        System.out.println("Main executing.....");

        System.out.println(barrier.getParties()); // 3
        barrier.reset(); // reset it
        executorService.shutdown();
    }
```

```
Main executing.....
3
pool-1-thread-2 service started
pool-1-thread-3 service started
pool-1-thread-1 service started
pool-1-thread-2 is waiting at the barrier
pool-1-thread-1 is waiting at the barrier
pool-1-thread-3 is waiting at the barrier
```

Ex- image a big application having multiple subsystem

```java
class Subsystem implements Runnable{
    private String name;
    private long initializationTime;
    private CyclicBarrier barrier;
    public Subsystem(String name, long initializationTime, CyclicBarrier barrier) {
        this.name = name;
        this.initializationTime = initializationTime;
        this.barrier = barrier;
    }
    @Override
    public void run() {
        try{
            System.out.println(name + " initialization started");
            Thread.sleep(initializationTime); // simulate time taken to initialize
            System.out.println(name + " initialization completed");
            barrier.await();
        }catch (Exception e){ }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfSubsystems = 4;
        CyclicBarrier barrier = new CyclicBarrier(numberOfSubsystems, new Runnable() {
            @Override
            public void run() {
                System.out.println("All subsystems are up and runnning. System startup complete.");
            }
        });  // when all the waiting threads reach the barrier, then this barrier action is executed
             // performed by the last thread entering the barrier

        Thread webServerThread = new Thread(new Subsystem( name: "Web Server",  initializationTime: 2000, barrier));
        Thread dbThread = new Thread(new Subsystem( name: "DB",  initializationTime: 4000, barrier));
        Thread cacheThread = new Thread(new Subsystem( name: "Cache",  initializationTime: 3000, barrier));
        Thread messagingServiceThread = new Thread(new Subsystem( name: "Messaging Service",  initializationTime: 3500, barrier));
        webServerThread.start();
        dbThread.start();
        cacheThread.start();
        messagingServiceThread.start();
    }
```

O/P →

```
Messaging Service initialization started
Cache initialization started
DB initialization started
Web Server initialization started
Web Server initialization completed
Cache initialization completed
Messaging Service initialization completed
DB initialization completed
All subsystems are up and runnning. System startup complete.
```

CompletableFuture

Introduced in JAVA 8 to handle asynchronous programming

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});
System.out.println("Main");
```

O/P →

```
Main
```

we don't see sout of "Worker" because CompletableFuture uses daemon thread.

If we want to run the task –

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});
String s = completableFuture.get(); // Waits if necessary for this future to complete,
                                    // and then returns its result.
System.out.println(s);
System.out.println("Main");
```

O/P →

```
Worker
ok
Main
```

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});
String s = completableFuture.getNow( valueIfAbsent: "noo"); // Returns the result value if completed,
                                                            // else returns the given valueIfAbsent.
System.out.println(s);
System.out.println("Main");
```

O/P →

```
noo
Main
```

```java
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});

CompletableFuture<String> f2 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});

CompletableFuture<Void> f = CompletableFuture.allOf(f1, f2); // Returns a new CompletableFuture that is completed
                                                             // when all the given CompletableFutures complete
f.join(); // blocking method that makes the calling thread wait until the CompletableFuture (f) completes.
System.out.println("Main");
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-javaagent:C:\Program Files\JetBrains\
Worker
Worker
Main
```

```java
String f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}).get();
System.out.println("Main");
}
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.ba
Worker
Main
```

```java
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}).thenApply(x->x.toUpperCase());
System.out.println(f1.get());
System.out.println("Main");
}
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAM
Worker
OK
Main
```

```java
        CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep( millis: 5000);
                System.out.println("Worker");
            } catch (InterruptedException e) { }
            return "ok";
        }).orTimeout( timeout: 2, TimeUnit.SECONDS).exceptionally(s -> "Timeout occurred");
        System.out.println(f1.get());
        System.out.println("Main");
    }
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-java
Timeout occurred
Main
```

By default, CompletableFuture tasks often run on daemon threads due to the use of ForkJoinPool.commonPool

You can control the thread type by providing a custom executor service.

The CompletableFuture task itself doesn't dictate whether it's a daemon or user thread.

```java
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 3);
        CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep( millis: 5000);
                System.out.println("Worker");
            } catch (InterruptedException e) { }
            return "ok";
        }, executorService); // now it will use threads from our pool
        System.out.println(f1.get());
        executorService.shutdown();
        System.out.println("Main");
    }
```

Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "
Worker
ok
Main
```

Volatile vs Atomic

```java
class SharedResource{
    // if we want to tell Threads that don't cache it and read/manipulate directly the main memory
    // make the variable volatile
    private volatile boolean flag = false;
    public void setFlagTrue() {
        System.out.println("Writer thread made the flag true");
        this.flag = true;
    }
    public void printIfFlagTrue() {
        while(!flag){
            // flag became true after 1000 ms but readerThread is still stuck in infinite loop
            // because each thread keeps local copy of variables in its cache for performance reasons
            // cached value of flag is false
            // problematic because flag is sharedResource
        }
        System.out.println("Flag is true");
    }
}
public class Main {
    public static void main(String[] args){
        SharedResource sharedObj = new SharedResource();
        Thread writerThread = new Thread(()->{
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            sharedObj.setFlagTrue();
        });
        Thread readerThread = new Thread(()->sharedObj.printIfFlagTrue());
        writerThread.start();
        readerThread.start();
    }
}
```

If we don't make variable volatile then readerThread is stuck in infinite loop. But in complex scenarios it also has disadvantages as there is no thread safety.

AtomicInteger, AtomicLong, AtomicBoolean - does not use locks. It achieves thread safety using **non-blocking, lock-free algorithms** that rely on **low-level hardware support** for atomic operations.

```java
public class VolatileCounter {
    private  int counter = 0;
    public void increment(){
        counter++;
    }
    public int getCounter(){
        return counter;
    }

    public static void main(String[] args) throws InterruptedException {
        VolatileCounter vc = new VolatileCounter();
        Thread t1 = new Thread(()->{
            for(int i=0;i<1000;i++) vc.increment();
        });
        Thread t2 = new Thread(()->{
            for(int i=0;i<1000;i++) vc.increment();
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(vc.getCounter());  // value < 2000
    }
```

```java
public class VolatileCounter {
    // ensure atomicity without using locks
    private AtomicInteger counter = new AtomicInteger( initialValue: 0);
    public void increment(){
        counter.incrementAndGet();
    }
    public int getCounter(){
        return counter.get();
    }

    public static void main(String[] args) throws InterruptedException {
        VolatileCounter vc = new VolatileCounter();
        Thread t1 = new Thread(()->{
            for(int i=0;i<1000;i++) vc.increment();
        });
        Thread t2 = new Thread(()->{
            for(int i=0;i<1000;i++) vc.increment();
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(vc.getCounter());  // 2000
```