**Multi-Threading**

**─**

Contents

[**1. What is Multithreading?** 2](#_Toc197853785)

[**2. Java Thread Model** 4](#_Toc197853786)

[# Thread Class 4](#_Toc197853787)

[# Runnable Interface 5](#_Toc197853788)

[# Using Lambda Expression 5](#_Toc197853789)

[**3. Important Thread methods** 6](#_Toc197853790)

[# Thread Lifecycle Methods 6](#_Toc197853791)

[# Thread Methods 10](#_Toc197853792)

[**4. Synchronization** 14](#_Toc197853793)

[# Race Condition 14](#_Toc197853794)

[# Synchronized Keyword 15](#_Toc197853795)

[# Locks 17](#_Toc197853796)

[# Inter-Thread communication 18](#_Toc197853797)

[**5. Deadlock** 21](#_Toc197853798)

[# Prevention of Deadlock 24](#_Toc197853799)

[**6. Java Executor Framework** 27](#_Toc197853800)

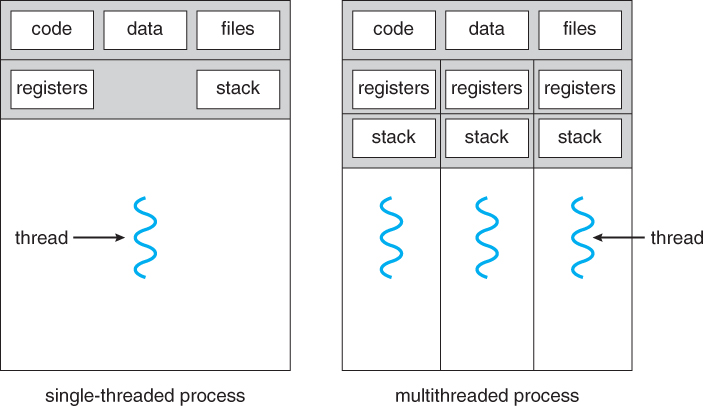
[**5. Interview Questions** 30](#_Toc197853801)

## **1. What is Multithreading?**

Multithreading is the concurrent execution of multiple threads within a single process. Each thread represents a separate path of execution.

**Threads:**

* Lightweight processes.
* Share the same memory and resources of their parent process.
* Each thread operates independently but runs within the context of the main program.



Advantages of Multithreading: -

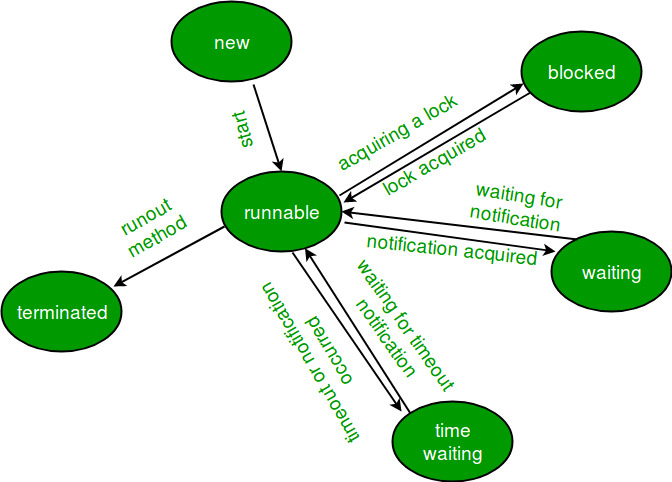
* **Concurrency**: Enables simultaneous execution of multiple tasks.
* **Better CPU Utilization**: Keeps the CPU busy during I/O operations.
* **Responsiveness**: Improves application responsiveness (e.g., GUI-based apps remain interactive while performing background tasks).
* **Scalability**: Allows tasks to scale effectively across available CPU cores.

Use Cases: -

* Web Servers: Handle multiple client requests simultaneously.
* GUI Applications: Keep UI responsive while executing background tasks.
* Gaming: Handle AI, rendering, and user inputs simultaneously.
* Data Processing: Process large data sets in parallel.
* Real-Time Systems: Applications like traffic control or stock market monitoring.

Thread Lifecycle: -

| **State** | **Description** |
| --- | --- |
| **NEW** | A thread is created but not started. |
| **RUNNABLE** | A thread is ready to run but waiting for CPU time. |
| **BLOCKED** | A thread is waiting for a monitor lock. |
| **WAITING** | A thread is waiting indefinitely for another thread to signal. |
| **TIMED\_WAITING** | A thread is waiting for a signal for a specified amount of time. |
| **TERMINATED** | A thread has finished its execution. |



## **2. Java Thread Model**

There are 2 ways to create threads in java

1. Thread class: By extending Thread class

2. Runnable Interface: By implementing runnable interface and passing it to Thread constructor.

### # Thread Class

* Create a subclass of the Thread class.
* Override the run() method to define the thread's task.
* Use start() to initiate the thread.

class MyThread extends Thread {  
 public void run() {  
 System.*out*.println("Thread is running: " + Thread.*currentThread*().getName());  
 }  
}  
  
public class LearnThread {  
 public static void main(String[] args) {  
 MyThread thread1 = new MyThread();  
 MyThread thread2 = new MyThread();  
  
 thread1.start(); //Thread is running: Thread-0  
 thread2.start(); //Thread is running: Thread-1  
 }  
}

### # Runnable Interface

Issue with Thread class is that Java don’t allow multiple inheritance, so if we are extending Thread class, we can’t extend any other class. So in place we can implement Runnable interface.

* Implement the Runnable interface.
* Pass an instance of the Runnable implementation to the Thread constructor.

class MyRunnable implements Runnable {  
 public void run() {  
 System.*out*.println("Runnable thread is running: " + Thread.*currentThread*().getName());  
 }  
}  
  
public class LearnThread {  
 public static void main(String[] args) {  
 MyRunnable runnable = new MyRunnable();  
 //Pass that to Thread Constructor  
 Thread thread = new Thread(runnable);  
 thread.start(); //Runnable thread is running: Thread-0  
 }   
}

### # Using Lambda Expression

public class LambdaThreadExample {  
 public static void main(String[] args) {  
 Thread thread = new Thread(() -> {  
 System.*out*.println("Thread running using lambda: " + Thread.*currentThread*().getName());  
 });  
 thread.start();  
 }  
}

Threading Example:

* Without threading, all code by default runs on main thread.
* If all threads have same priority, threads will run in random order

class MyThread extends Thread{  
 //thread.start runs run function ( we must override this in our thread class)  
 public void run(){  
 System.*out*.println("Hi from " + Thread.*currentThread*().getName());  
 for(int i=0;i<3;i++){  
 System.*out*.println(i + "=>" + Thread.*currentThread*().getName());  
 }  
 }  
}  
  
class LearnThread{  
 public static void main(String[] args) {  
 MyThread m1 = new MyThread();  
 MyThread m2 = new MyThread();  
  
 m1.start();  
 m2.start();  
  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 }  
}  
//Output   
//Hi from Thread-0  
//Hello from main  
//Hi from Thread-1  
//0=>Thread-1  
//1=>Thread-1  
//0=>Thread-0  
//2=>Thread-1  
//1=>Thread-0  
//2=>Thread-0

## **3. Important Thread methods**

### # Thread Lifecycle Methods

| **Method** | **Description** |
| --- | --- |
| start() | Moves a thread from the NEW state to the RUNNABLE state. |
| run() | Contains the logic for the thread's execution. |
| sleep(milliseconds) | Causes the thread to pause execution for a specified time. |
| join() | Causes the current thread to wait until another thread completes its execution. |
| wait() | Causes a thread to wait until another thread notifies it. |
| notify() | Wakes up a single thread that is waiting on the object's monitor. |
| notifyAll() | Wakes up all threads waiting on the object's monitor. |

**1.** In above code, if in place of m1.start(), if we did m1.run() , it will keep new thread m1 in NEW state and will execute the code in main thread. Thus multithreading won’t work that time and code will run sequentially.

class MyThread extends Thread{  
 public void run(){  
 System.*out*.println("Hi from " + Thread.*currentThread*().getName());  
 for(int i=0;i<3;i++){  
 System.*out*.println(i + "=>" + Thread.*currentThread*().getName());  
 }  
 }  
}  
  
class LearnThread{  
 public static void main(String[] args) {  
 MyThread m1 = new MyThread();  
 MyThread m2 = new MyThread();  
  
 m1.run();  
 m2.run();  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 }  
}  
//Output  
//Hi from main  
//0=>main  
//1=>main  
//2=>main  
//Hi from main  
//0=>main  
//1=>main  
//2=>main  
//Hello from main

**2.** If we want main thread to wait until task of thread1 and thread2 are completed, we can use join() method.

The join() method in Java is used to ensure that one thread waits for the completion of another thread before proceeding further. It is a way to synchronize threads so that the main thread or any other thread does not continue execution until the specified thread has finished its task.

* join(): Makes the current thread wait indefinitely until the specified thread finishes execution.
* join(long millis): Makes the current thread wait for a maximum of millis milliseconds for the specified thread to finish.

class Worker extends Thread {  
 private String name;  
  
 public Worker(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void run() {  
 try {  
 System.*out*.println(name + " started");  
 Thread.*sleep*(2000); // Simulate work  
 System.*out*.println(name + " completed");  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
   
 }  
}  
  
public class JoinExample {  
 private static Worker *thread2*;  
  
 public static void main(String[] args) throws InterruptedException {  
 Worker thread1 = new Worker("Thread 1");  
 Worker thread2 = new Worker("Thread 2");  
  
 thread1.start();  
 thread2.start();  
   
 thread1.join(); // Main thread waits for thread1 to complete  
 System.*out*.println("Thread 1 has finished, now waiting for Thread 2.");  
 thread2.join(); // Main thread waits for thread2 to complete  
  
 System.*out*.println("Both threads have completed. Main thread proceeds.");  
 }  
}  
//Thread 1 started  
//Thread 2 started  
//Thread 2 completed  
//Thread 1 completed  
//Thread 1 has finished, now waiting for Thread 2.  
//Both threads have completed. Main thread proceeds.

Real world example of File download and processing. We want file processing code to run only when download is completed.

class FileDownloader extends Thread {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("File downloading...");  
 Thread.*sleep*(3000); // Simulate file download time  
 System.*out*.println("File downloaded.");  
 } catch (InterruptedException e) {  
 System.*out*.println(e);  
 }  
 }  
}  
  
class FileProcessor extends Thread {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("Processing file...");  
 Thread.*sleep*(2000); // Simulate processing time  
 System.*out*.println("File processed.");  
 } catch (InterruptedException e) {  
 System.*out*.println(e);  
 }  
 }  
}  
  
public class FileDownloadAndProcess {

public static void main(String[] args) throws InterruptedException {  
 FileDownloader downloader = new FileDownloader();  
 FileProcessor processor = new FileProcessor();  
  
 downloader.start();

// Wait for file download to finish  
 downloader.join();   
  
 // Start processing after download is complete

processor.start();   
 }  
}

//File downloading...  
//File downloaded.  
//Processing file...  
//File processed.

**3.** Errors in one thread don’t stop execution of other threads.

Example in below code, for thread 1 when i=1, we throw Exception, but it don’t stop execution of other threads.

class ThreadEx extends Thread{  
  
 @Override  
 public void run(){  
 for (int i=0; i<3 ; i++){  
 System.*out*.println(i + "=>" + Thread.*currentThread*().getName());  
  
 if(Thread.*currentThread*().getName().equals("Thread-1") && i==0){  
 throw new RuntimeException("Stopped");  
 }  
 }  
 }  
}  
  
public class ThreadException {  
  
 public static void main(String[] args) {  
 ThreadEx t1 = new ThreadEx();  
 ThreadEx t2 = new ThreadEx();  
 ThreadEx t3 = new ThreadEx();  
  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}  
//0=>Thread-1  
//0=>Thread-0  
//0=>Thread-2  
//1=>Thread-0  
//1=>Thread-2  
//2=>Thread-0  
//2=>Thread-2  
//Exception in thread "Thread-1" java.lang.RuntimeException:

### # Thread Methods

| **Method** | **Description** |
| --- | --- |
| setName(String name) | Sets the name of the thread. |
| getName() | Returns the thread's name. |
| getId() | Returns the thread's unique ID. |
| setPriority(int priority) | Sets the thread's priority. |
| getPriority() | Returns the thread's priority. |
| isAlive() | Checks if the thread is still active. |
| interrupt() | Interrupts a thread in sleep or wait state. |
| isInterrupted() | Checks if the thread has been interrupted. |

Thread Name:

By default thread are named as main, Thread-0 , Thread-1… We can set its name via setName() or in Thread Constructor too.

class MyThread implements Runnable{  
  
 @Override  
 public void run() {  
 System.*out*.println("Thread Name is " + Thread.*currentThread*().getName());  
 }  
}  
  
class LearnThread{  
 public static void main(String[] args) {  
 //set name using setName method  
 Thread m1 = new Thread(new MyThread());  
 m1.setName("First Thread");  
  
 //set name via Thread Constructor  
 MyThread my = new MyThread();  
 Thread m2 = new Thread(my,"Second Thread");  
  
 m1.start();  
 m2.start();  
  
 System.*out*.println("Main Thread is " + Thread.*currentThread*().getName());  
 }  
}  
//Main Thread is main  
//Thread Name is Second Thread  
//Thread Name is First Thread

Thread Priority:

We have thread scheduler, which runs thread based on priority. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

* The default priority is set to 5 as excepted( Thread.NORM\_PRIORITY)
* Minimum priority is set to 1 (Thread.MIN\_PRIORITY)
* Maximum priority is set to 10. (Thread.MAX\_PRIORITY)

**How Thread Priority Works:**

* Concept: Higher-priority threads are more likely to be executed before lower-priority threads when there are CPU resource constraints.
* Thread Scheduler: The thread priority is only a suggestion to the thread scheduler, which may ignore it depending on the operating system's implementation.
* Not Guaranteed: Thread priority is just a suggestion to the thread scheduler, which may or may not respect it.
* Platform dependent: Effect of thread priority depends on the operating system and JVM implementation.

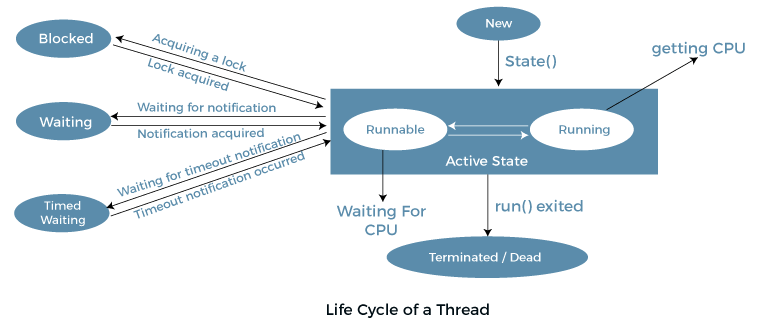
class MyThread implements Runnable{  
  
 @Override  
 public void run() {  
 System.*out*.println("Thread " + Thread.*currentThread*().getName() + " Priority " + Thread.*currentThread*().getPriority());

}  
}  
  
class LearnThread{  
 public static void main(String[] args) {  
 Thread m1 = new Thread(new MyThread());  
 Thread m2 = new Thread(new MyThread());  
 Thread m3 = new Thread(new MyThread());  
  
 m1.setPriority(Thread.*MIN\_PRIORITY*);  
 m2.setPriority(10); //Max\_Priority  
  
 m1.start();  
 m2.start();  
 m3.start();  
  
 }  
}  
//Note:- Output may change, but this will be most likely to occur  
//Thread Thread-1 Priority 10  
//Thread Thread-2 Priority 5  
//Thread Thread-0 Priority 1

Thread States:

class ThreadStatesDemo implements Runnable {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() + " is RUNNING");  
  
 // TIMED\_WAITING: Thread sleeps for 2 seconds  
 Thread.*sleep*(1000);

} catch (InterruptedException e) {  
 System.*out*.println(Thread.*currentThread*().getName() + " was INTERRUPTED");  
 }  
 }  
}  
  
public class ThreadStateExample {  
 public static void main(String[] args) throws InterruptedException {  
 ThreadStatesDemo task = new ThreadStatesDemo();  
 Thread thread = new Thread(task, "DemoThread");  
  
 // NEW: Thread is created but not yet started  
 System.*out*.println(thread.getName() + " state: " + thread.getState());  
  
 thread.start(); // Thread is moved to RUNNABLE state  
  
 // RUNNABLE: Thread is ready to run or running  
 System.*out*.println(thread.getName() + " state after start(): " + thread.getState());  
  
 // Ensure thread starts running  
 Thread.*sleep*(100);  
 System.*out*.println(thread.getName() + " state during sleep(): " + thread.getState());  
  
 // Wait for the thread to complete  
 thread.join();  
  
 // TERMINATED: Thread has finished execution  
 System.*out*.println(thread.getName() + " state after completion: " + thread.getState());  
 }  
}  
//DemoThread state: NEW  
//DemoThread state after start(): RUNNABLE  
//DemoThread is RUNNING  
//DemoThread state during sleep(): TIMED\_WAITING  
//DemoThread state after completion: TERMINATED



NEW:

* The thread is created but not yet started using start().
* thread.getState() returns NEW.

RUNNABLE:

* After calling start(), the thread is in the RUNNABLE state.
* It is either running or ready to run, depending on the CPU scheduling.

TIMED\_WAITING:

* When Thread.sleep(2000) is called, the thread moves to the TIMED\_WAITING state for 2 seconds.
* It will transition back to RUNNABLE after the sleep time is over.

WAITING:

* Inside the run() method, the thread calls wait() inside a synchronized block.
* It moves to the WAITING state, waiting to be notified by another thread.

BLOCKED (optional in this example):

* If another thread is holding the lock that this thread is trying to acquire, it moves to the BLOCKED state.

TERMINATED:

* After completing its task, the thread enters the TERMINATED state.
* At this point, it can no longer be restarted.

### # Yield

yield() is a static method that allows the currently executing thread to pause and let other threads of **equal or higher priority** execute.

* **Pauses the current thread voluntarily**, giving a hint to the thread scheduler that **it can switch to another runnable thread**.
* It's like the thread saying, **“I’m willing to let others run if they are ready.”**

public class YieldDemo {  
 public static void main(String[] args) {  
  
 Runnable printer = ()->{  
 for(int i=1;i<=3;i++){  
 System.*out*.println(Thread.*currentThread*().getName()+ " printing " +i);  
  
 Thread.*yield*(); //allow other thread to run  
 }  
 };  
  
 Thread t1 = new Thread(printer,"Thread-A");  
 Thread t2 = new Thread(printer, "Thread-B");  
  
 t1.start();  
 t2.start();  
 }  
}

output:

Thread-B printing 1  
Thread-A printing 1  
Thread-B printing 2  
Thread-B printing 3  
Thread-A printing 2  
Thread-A printing 3

Explanation:

* Both threads are executing the same task.
* After printing a page, each thread **yields**.
* This allows the **other thread to run**, encouraging more fair execution.
* Note: The actual switch depends on **CPU availability** and **scheduler's decision**.

## **4. Synchronization**

Synchronization is a technique to control access to shared resources by multiple threads to avoid race conditions. It ensures that only one thread can access a critical section (shared resource) at a time.

Java provides the synchronized keyword for this.

### # Race Condition

A race condition occurs when two or more threads access shared mutable data and try to change it at the same time, leading to unpredictable results.

**Code example:** Here we have 2 thread accessing the same counter

class Counter {  
 int count = 0;  
  
 public void increment() {  
 count++; // not atomic: read → modify → write  
 }  
}  
  
public class CounterTest {  
 public static void main(String[] args) throws InterruptedException {  
 Counter counter = new Counter();  
  
 Thread t1 = new Thread(() -> {  
 for (int i = 0; i < 1500; i++) {  
 //incrementing value inside this run method  
 counter.increment();  
 }  
 });  
  
 Thread t2 = new Thread(() -> {  
 for (int i = 0; i < 1500; i++) {  
 counter.increment();  
 }  
 });  
  
 t1.start();  
 t2.start();  
  
 t1.join();  
 t2.join();  
  
 System.*out*.println("Final count (unsynchronized): " + counter.count);  
 }  
}

Here as output sometimes we get 2910, 2701, 2828 etc. Rarely we can get 3000. Though if both threads increase value by 1500, total should become 3000. Issue is race condition. count++ is not atomic, it expands to read count, add 1, write back. Threads may read the same value and overwrite each other.

### # Synchronized Keyword

* **Ensures mutual exclusion**: Only one thread at a time can execute the synchronized block/method.
* **Prevents race conditions**.

Java Synchronization Types

* Synchronized Instance Methods – Locks on the current object (this).
* Synchronized Static Methods – Locks on the class object (ClassName.class).
* Synchronized Blocks – Fine-grained control, you specify which object to lock.

Can fix above code by: Now will get 3000

**1. Synchronized method:**

class Counter {  
 int count = 0;  
  
 public synchronized void increment() {  
 count++;   
 }  
}

**2. Synchronized block**

class Counter {  
 int count = 0;  
  
 public void increment() {  
 synchronized(this) {  
 count++;  
 }  
 }  
}

You should use **synchronized blocks** when:

* Only a part of the method is critical.
* You want to synchronize on a specific object.

**How this works?**

Intrinsic lock (or monitor lock) is the built-in lock that every Java object has, used implicitly by the synchronized keyword to ensure exclusive access to synchronized code blocks.

Every object in Java has an **intrinsic lock (or monitor)**. When a thread enters a synchronized block or method, it:

1. Tries to acquire the intrinsic lock.
2. If the lock is held by another thread, it waits.
3. Once it gets the lock, it enters and executes the block.
4. After execution, it releases the lock.

Java's synchronized keyword ensures:

* **Mutual exclusion** – only one thread at a time.
* **Consistency** – no race condition.
* **Predictability** – same logic gives same results every time.

**Eg2: Movie ticket booking**

class MovieTheater {  
 int availableSeats = 1;  
  
 public synchronized void bookSeat(String name) {  
 if (availableSeats > 0) {  
 System.*out*.println(name + " found seat available...");  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {}  
 availableSeats--;  
 System.*out*.println(name + " successfully booked seat!");  
 } else {  
 System.*out*.println(name + " found no seat available.");  
 }  
 }  
}  
  
public class TicketBooking {  
 public static void main(String[] args) {  
 MovieTheater theater = new MovieTheater();  
 Runnable r = () -> theater.bookSeat(Thread.*currentThread*().getName());  
  
 Thread t1 = new Thread(r, "Alice");  
 Thread t2 = new Thread(r, "Bob");  
  
 t1.start();  
 t2.start();  
 }  
}

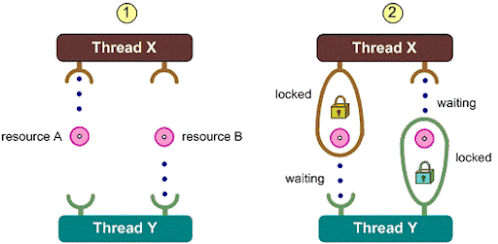
Without synchronization, both can acquire seat, even though there is only one seat. With synchronization, output is always:

//Alice found seat available...  
//Alice successfully booked seat!  
//Bob found no seat available.

### # Locks

In **multithreading**, **locks** are tools used to **prevent multiple threads from accessing a shared resource at the same time**, which could lead to **race conditions** (unpredictable behaviour or corrupted data).

Think of a **Lock** as a **gatekeeper** or **security guard** who ensures **only one person or a controlled group enters a room** (shared resource) at a time, depending on the access type.



Synchronization too by default use intrinsic lock.

Lock is an **interface** in the java.util.concurrent.locks package that provides **more powerful and flexible locking** than synchronized. Lock methods are

| **Method** | **Definition** | **Theory** |
| --- | --- | --- |
| lock() | Acquires the lock, blocking until it's available. | Ensures exclusive access by making the thread wait until the lock is released. |
| unlock() | Releases the lock held by the current thread. | Required to manually release the lock so other threads can acquire it. |
| tryLock() | Attempts to acquire the lock without waiting. | Returns immediately with success or failure, avoiding thread blocking. |
| tryLock(timeout, unit) | Tries to acquire the lock within the specified time limit. | Waits up to the timeout duration before failing if the lock isn't available. |
| lockInterruptibly() | Acquires the lock unless the thread is interrupted. | Allows waiting threads to be interrupted, useful for responsive multithreaded apps. |
| newCondition() | Returns a Condition instance for advanced thread control. | Supports fine-grained wait/notify mechanisms similar to Object.wait() and notify(). |

Same movie ticket booking code with Lock:

class MovieTheater {  
 private int seats = 1;  
 private Lock lock = new ReentrantLock();  
  
 public void bookSeat(String name) {  
 lock.lock();  
 try {  
 if (seats > 0) {  
 System.*out*.println(name + " found a seat...");  
 Thread.*sleep*(100);  
 seats--;  
 System.*out*.println(name + " booked a seat!");  
 } else {  
 System.*out*.println(name + " found no seats.");  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock(); // always unlock!  
 }  
 }  
}

* **private Lock lock = new ReentrantLock() :** A ReentrantLock is created to manage access to the critical section (seat booking logic). Ensures that only one thread can execute that logic at a time**.** Supports fairness policy (FIFO lock granting).
* **lock.lock():** Acquires the lock. If another thread already holds the lock, this thread blocks and waits.

### # Inter-Thread communication

**Inter-thread communication** allows synchronized threads to cooperate rather than compete. It enables threads to **wait for a condition to be met** by another thread before proceeding.

Java provides this mechanism using:

* wait()
* notify()
* notifyAll()

These methods are used with **intrinsic locks** (inside a synchronized block) and are defined in the Object class.

**Notes:**

* These methods must be called from inside a synchronized block.
* The object used in synchronized(obj) must be the same object whose monitor is used in wait() / notify().

**Important methods:**

| **Concept** | **Description** |
| --- | --- |
| wait() | Tells the current thread to **release the lock** and **wait** until another thread invokes notify() or notifyAll() on the same object. |
| notify() | Wakes up **one** thread that is waiting on the object’s monitor (if any). |
| notifyAll() | Wakes up **all** threads waiting on that object’s monitor. |

**Why do we need it?**

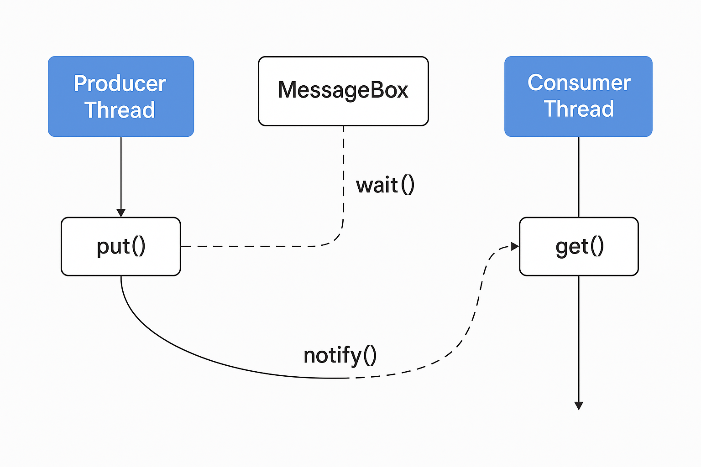
Let’s say:

* One thread **produces data** (Producer).
* Another thread **consumes it** (Consumer).

They must **coordinate**:

* The consumer should **wait** until the producer has produced data.
* The producer should **notify** the consumer once the data is ready.

Without inter-thread communication, you'd need awkward polling (wasting CPU), or face **race conditions**.



class MessageBox {  
 private String message;  
 private boolean hasMessage = false;  
  
 public synchronized void put(String msg) {  
 while (hasMessage) {  
 try {  
 wait(); // wait for consumer to consume  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 this.message = msg;  
 hasMessage = true;  
 System.*out*.println("Produced: " + msg);  
 notify(); // notify waiting consumer  
 }  
  
 public synchronized void get() {  
 while (!hasMessage) {  
 try {  
 wait(); // wait for producer to produce  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("Consumed: " + message);  
 hasMessage = false;  
 notify(); // notify producer  
 }  
}

class Producer extends Thread {  
 private MessageBox box;  
 public Producer(MessageBox box) {  
 this.box = box;  
 }  
 public void run() {  
 box.put("Hello from Producer!");  
 }  
}  
  
class Consumer extends Thread {  
 private MessageBox box;  
 public Consumer(MessageBox box) {  
 this.box = box;  
 }  
 public void run() {  
 box.get();  
 }  
}

public class ThreadCommunication {  
 public static void main(String[] args) {  
 MessageBox box = new MessageBox();  
 new Consumer(box).start();  
 new Producer(box).start();  
 }  
}

//output: Even though consumer is called first, it waits for producer to put message and then it receives it  
  
//Produced: Hello from Producer!  
//Consumed: Hello from Producer!

**Flow:**

* Consumer starts and sees hasMessage == false, so it waits.
* Producer puts the message, sets hasMessage = true, then notify() is called.
* Consumer wakes up, reads the message, sets hasMessage = false, and notify() is called again (so the producer can send another message if needed).

Both notify and wait should be called on same lock, meaning there should be some shared resource in between producer and consumer.

**Eg2:** Print numbers 1 to 10, using 2 different threads

class PrintNum{  
 private boolean isOdd = true;  
  
 public synchronized void printOdd(int number) throws InterruptedException {  
 while(!isOdd){  
 wait(); //wait till it's odd turn  
 }  
 System.*out*.println("Odd: "+number);  
 isOdd=false;  
 notify(); //notify even thread  
 }  
  
 public synchronized void printEven(int number) throws InterruptedException{  
 while(isOdd){  
 wait(); //wait to get even turn  
 }  
 System.*out*.println("Even: " + number);  
 isOdd=true;  
 notify(); //notify odd thread  
 }  
}

public class AlternatePrinter {  
 public static void main(String[] args) {  
 PrintNum printer = new PrintNum();  
  
 Thread oddThread = new Thread(()->{  
 for(int i=1;i<=10;i+=2){  
 try{  
 printer.printOdd(i);  
 }catch (InterruptedException e){  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 });  
  
 Thread evenThread = new Thread(() -> {  
 for (int i = 2; i <= 10; i += 2) {  
 try {  
 printer.printEven(i);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 oddThread.start();  
 evenThread.start();  
 }  
}

### # Atomic class

Atomic classes are part of the java.util.concurrent.atomic package and provide **lock-free, thread-safe operations** using **low-level atomic CPU instructions** like **compare-and-swap (CAS)**.

| **Atomic Type** | **Used For** | **Example Operation** |
| --- | --- | --- |
| AtomicInteger | Atomic operations on int | get(), incrementAndGet() |
| AtomicLong | Atomic operations on long | addAndGet(5) |
| AtomicBoolean | Atomic boolean flag | compareAndSet(false, true) |
| AtomicReference<T> | Atomic operations on any object | getAndSet(newObj) |

import java.util.concurrent.atomic.AtomicInteger;  
  
class Counter{ //shared resource  
 AtomicInteger counter = new AtomicInteger(0);  
  
 public void increment(){  
 counter.getAndIncrement(); // counter = counter + 1;(as single transactions)  
 }  
}  
  
  
class MyThread2 extends Thread{  
 Counter counter;  
  
 public MyThread2(Counter counter){  
 this.counter = counter;  
 }  
  
 @Override  
 public void run(){  
 for(int i=0;i<1000;i++) {  
 counter.increment();  
 }  
 }  
}  
  
  
public class ThreadBasics {  
 public static void main(String[] args) throws InterruptedException {  
 Counter abc = new Counter();  
  
 Thread t0 = new MyThread2(abc);  
 MyThread2 t1 = new MyThread2(abc);  
   
 t0.start();  
 t1.start();  
  
 t0.join();  
 t1.join();  
 System.*out*.println(abc.counter);  
 }  
}

## **5. Deadlock**

A deadlock is a situation where two or more threads are blocked forever, each waiting for the other to release a resource.

In simpler terms, it's like a traffic jam of threads — no one can move forward because each is waiting for something held by another.

**Real world analogies:**

**1. Chopstick Problem (Dining Philosophers)**

* Five philosophers are sitting at a round table, each needing **two chopsticks** to eat.
* Each picks up the chopstick to their right and waits for the left one.
* Now all are holding one chopstick and **waiting forever** for the other → **Deadlock**.

**2. Printer and Scanner**

* Thread A holds the **printer**, needs the **scanner**.
* Thread B holds the **scanner**, needs the **printer**.
* Both are waiting forever → **Deadlock**.

**Issues with deadlocks**

| **Impact** | **Explanation** |
| --- | --- |
| **App Freezes** | Threads stop progressing indefinitely. |
| **Resource Waste** | CPU, memory, and locks are stuck. |
| **Difficult Debugging** | Deadlocks are hard to detect and reproduce. |
| **User Impact** | Poor UX, app crashes, unresponsiveness. |

**Coffman’s Conditions** (Condition for deadlock)

A deadlock can occur **only if all four of these conditions are true** at the same time:

| **Condition** | **Explanation** |
| --- | --- |
| **Mutual Exclusion** | At least one resource is held in a non-shareable mode. |
| **Hold and Wait** | A thread is holding at least one resource and waiting for another. |
| **No Preemption** | Resources cannot be forcibly taken away; they must be released voluntarily. |
| **Circular Wait** | A circular chain of threads exists where each thread is waiting for a resource held by the next. |

If **all four** are true, deadlock is possible.

**What is a Monitor Lock?**

In Java, every object has an internal lock called a **monitor,** hence any object can act as a lock. When a thread enters a synchronized block, it:

* Acquires the monitor for the object.
* Releases it when it exits the block (normally or via return, break, or even exception).

**Java Code Example:**

Imagine:

* Thread 1 acquires **Lock A**, then waits for **Lock B**.
* Thread 2 acquires **Lock B**, then waits for **Lock A**.
* Now both threads are **stuck**, each waiting for the other → Deadlock.

public class DeadlockExample {  
  
 // Define two locks (resources)  
 static final Object *LOCK\_A* = new Object();  
 static final Object *LOCK\_B* = new Object();  
  
 public static void main(String[] args) {  
 // Thread 1 tries to acquire LOCK\_A then LOCK\_B  
 Thread thread1 = new Thread(() -> {  
 synchronized (*LOCK\_A*) {  
 System.*out*.println("Thread 1: Acquired LOCK\_A");  
  
 // Sleep to simulate work and increase chance of deadlock  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
  
 System.*out*.println("Thread 1: Waiting for LOCK\_B...");  
 synchronized (*LOCK\_B*) {  
 System.*out*.println("Thread 1: Acquired LOCK\_B");  
 }  
 }  
 });  
  
 // Thread 2 tries to acquire LOCK\_B then LOCK\_A (reverse order!)  
 Thread thread2 = new Thread(() -> {  
 synchronized (*LOCK\_B*) {  
 System.*out*.println("Thread 2: Acquired LOCK\_B");  
  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
  
 System.*out*.println("Thread 2: Waiting for LOCK\_A...");  
 synchronized (*LOCK\_A*) {  
 System.*out*.println("Thread 2: Acquired LOCK\_A");  
 }  
 }  
 });  
  
 // Start both threads  
 thread1.start();  
 thread2.start();  
 }  
}

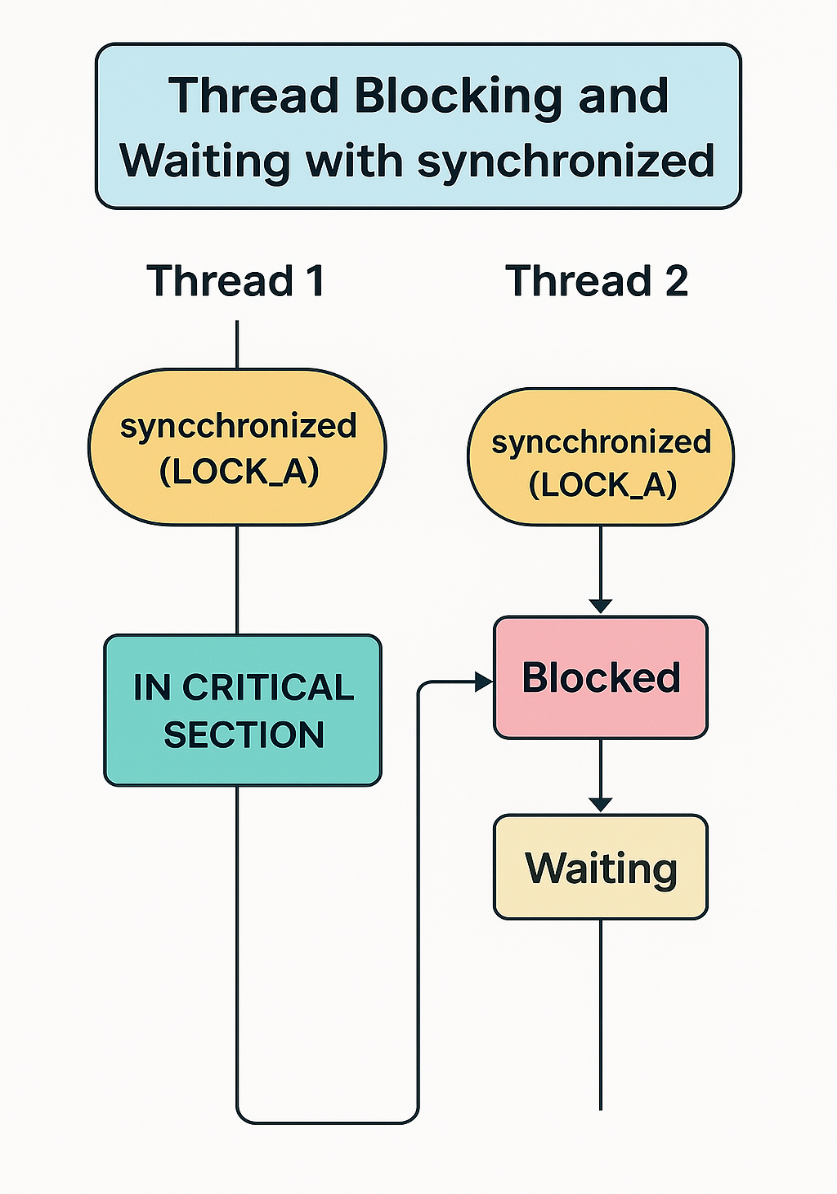
output:

//Thread 2: Acquired LOCK\_B  
//Thread 1: Acquired LOCK\_A  
//Thread 1: Waiting for LOCK\_B...  
//Thread 2: Waiting for LOCK\_A...

And then the program **freezes**, because both threads are stuck forever — **deadlock!**

**Breakdown of What Happens**

* LOCK\_A is an object (any object in Java can act as a lock).
* synchronized (LOCK\_A) means:
  + This code block is **mutually exclusive** — **only one thread** can execute it **at a time** for the same lock.
  + If Thread-1 acquires the lock, and Thread-2 tries to enter, Thread-2 must **wait** until Thread-1 releases the lock.
* Once inside, the thread is guaranteed **exclusive access** to the block.



### # Prevention of Deadlock

To prevent deadlock in Java, you need to follow certain design principles and use best practices. Deadlocks occur when multiple threads hold resources and wait indefinitely for each other to release them.

To **prevent deadlocks**, your strategy should focus on:

* Good design upfront (especially resource ordering).
* Using non-blocking locking mechanisms (tryLock).
* Monitoring and detecting risks at runtime.

Example: Preventing deadlock via tryLock

import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
import java.util.concurrent.TimeUnit;  
  
public class DeadlockFreeDemo {  
 private static final Lock *LOCK\_A* = new ReentrantLock();  
 private static final Lock *LOCK\_B* = new ReentrantLock();  
  
 public static void main(String[] args) {  
 Thread t1 = new Thread(() -> *acquireLocks*("Thread-1", *LOCK\_A*, *LOCK\_B*));  
 Thread t2 = new Thread(() -> *acquireLocks*("Thread-2", *LOCK\_B*, *LOCK\_A*));  
  
 t1.start();  
 t2.start();  
 }  
  
 public static void acquireLocks(String name, Lock firstLock, Lock secondLock) {  
 while (true) {  
 try {  
 // Try to acquire first lock  
 boolean gotFirst = firstLock.tryLock(100, TimeUnit.MILLISECONDS);  
 if (gotFirst) {  
 // Try to acquire second lock  
 boolean gotSecond = secondLock.tryLock(100, TimeUnit.MILLISECONDS);  
 if (gotSecond) {  
 try {  
 System.*out*.println(name + " acquired both locks.");  
 // Perform critical section  
 break;  
 } finally {  
 secondLock.unlock();  
 firstLock.unlock();  
 System.*out*.println(name + " released both locks.");  
 }  
 } else {  
 // Could not acquire second lock  
 firstLock.unlock();  
 System.*out*.println(name + " released first lock, will retry.");  
 }  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 // Optional: small delay before retry  
 try {  
 Thread.*sleep*(50);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

|  | **What Happens** |
| --- | --- |

|  |  |
| --- | --- |
| 1 | Each thread tries to acquire its first lock using tryLock(timeout) |

|  |  |
| --- | --- |
| 2 | If it fails to acquire **both locks**, it releases the acquired lock and **retries** |

|  |  |
| --- | --- |
| 3 | No thread waits indefinitely — so deadlock is **avoided** |

|  |  |
| --- | --- |
| 4 | If both locks are acquired, thread enters critical section |

|  |  |
| --- | --- |
| 5 | Locks are always released in finally block |

**Some other ways:**

| **Step** | **One-Liner** |
| --- | --- |
| 1. Resource Ordering | Acquire locks in a consistent global order to avoid circular wait. |
| 2. Lock Timeout | Use timeouts while trying to acquire a lock to break potential deadlocks. |
| 3. Try-Lock Mechanism | Use tryLock() instead of blocking lock() to avoid waiting indefinitely. |
| 4. Avoid Nested Locks | Minimize or avoid acquiring multiple locks together. |
| 5. Deadlock Detection & Recovery | Detect cycles in resource allocation and break the deadlock using intervention. |
| 6. Lock Level Design | Use coarse or fine-grained locks appropriately to avoid conflicts. |
| 7. Thread Design | Minimize thread-to-thread dependencies and shared mutable state. |

## **6. Java Executor Framework**

The **Executor Framework** is a part of java.util.concurrent package that provides a **simple, flexible, and powerful way to manage threads** in Java.

Instead of manually creating and starting threads using new Thread(), the Executor Framework allows you to submit tasks for execution and let it manage the thread pool (set of reusable threads) for you.

**Why do we need Executor Framework?**

| **Problem with Thread class** | **How Executor solves it** |
| --- | --- |
| Manual creation and starting of each thread | Executors reuse a pool of threads |
| No proper thread management | Executors provide controlled lifecycle |
| Hard to scale for large apps | Executors optimize performance with different strategies |
| No result from threads | Executors allow us to get results back (via Future) |

**Core Components**

1. **Executor**: Basic interface with method execute(Runnable)

2. **ExecutorService:** Extends Executor, adds methods like submit(), shutdown()

3. **Executors:** Utility class to create different thread pools

4. **Future:** Represents the result of an asynchronous computation

5. **Callable:** Like Runnable, but returns a result and can throw exceptions. Runnable has run() method which can’t return a value. Both are functional interfaces.

**Types of Thread Pools (via Executors)**

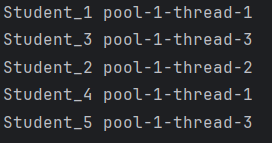
| **Method** | **Description** |
| --- | --- |
| newFixedThreadPool(n) | Fixed number of threads in the pool |
| newCachedThreadPool() | Creates new threads as needed and reuses existing |
| newSingleThreadExecutor() | Single thread executes tasks sequentially |
| newScheduledThreadPool(n) | Used for delayed or periodic task execution |

**Code 1:** Program to print names of 5 students using ExecutorService.

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class ExecutorDemo {  
 public static void main(String[] args) {  
  
 //Step 1: Create a thread pool with 3 threads  
 ExecutorService executor = Executors.*newFixedThreadPool*(3);  
  
 //Step2: Create 5 student tasks and then submit all  
 for(int i=1;i<=5;i++){  
 String studentName = "Student\_"+i;

Runnable task = ()-> System.*out*.println(studentName + " " + Thread.*currentThread*().getName() );

executor.execute(task); //submit task to thread  
 }  
 //Step 3: Shutdown the executor  
 // Tells executor to stop accepting new tasks and gracefully shutdown  
 executor.shutdown();  
 }  
}

Output: 

**Code 2:** Using submit() and Callable to get result

import java.util.concurrent.\*;  
  
public class CallableDemo {  
 public static void main(String[] args) throws Exception {  
 // Step 1: Create a single-thread executor  
 ExecutorService executor = Executors.*newSingleThreadExecutor*();  
  
 // Step 2: Submit a Callable task which returns a result  
 Callable<String> task = () -> {  
 Thread.*sleep*(1000); // Simulate delay  
 return "Hello from Callable!";  
 };  
  
 //A Future in Java represents the result of an asynchronous computation.  
 Future<String> future = executor.submit(task);  
  
 // Step 3: Use Future to get the result  
 System.*out*.println("Waiting for result...");  
 //Future provides methods like get() to retrieve the result once the task is complete, blocking if necessary.  
 String result = future.get(); // This will block until result is ready  
 System.*out*.println("Result: " + result);  
  
 executor.shutdown(); // Always shut down executor  
 }  
}

output:

Waiting for result...  
Result: Hello from Callable!

**Code 3:** Using invokeAll() to Run Multiple Callables in Parallel

import java.util.concurrent.\*;  
import java.util.\*;  
  
public class InvokeAllExample {  
 public static void main(String[] args) throws Exception {  
 ExecutorService executor = Executors.*newFixedThreadPool*(3);  
  
 List<Callable<String>> tasks = Arrays.asList(  
 () -> "Task 1 done",  
 () -> "Task 2 done",  
 () -> "Task 3 done"  
 );  
  
 List<Future<String>> results = executor.invokeAll(tasks); // Executes all tasks and waits  
  
 for (Future<String> future : results) {  
 System.*out*.println(future.get());  
 }  
  
 executor.shutdown();  
 }  
}

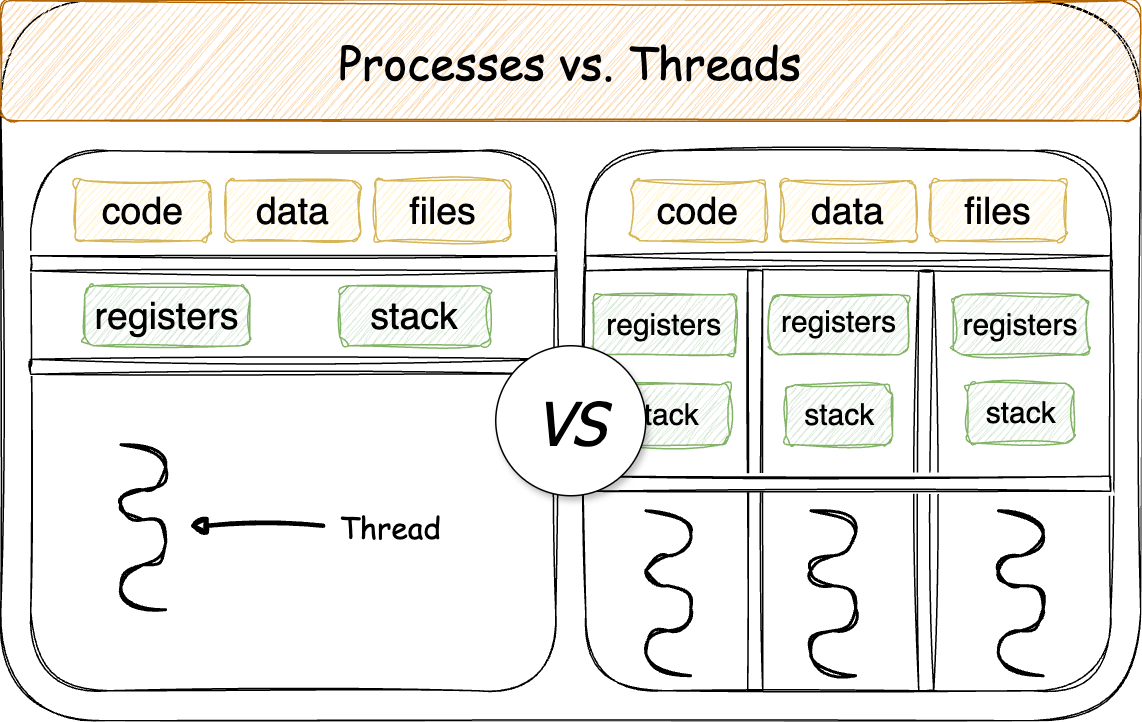
* invokeAll() executes a collection of Callable tasks
* It **waits for all tasks** to finish and returns a list of Future objects

## **7. Interview Questions**

1. **What are differences between threads and processes?**

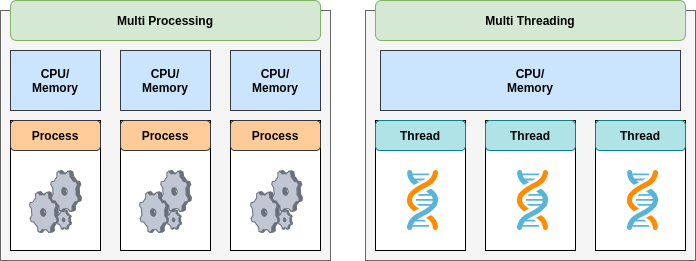
**Process:** - A process is an independent program in execution. It has its own memory space and resources, making it a heavyweight entity.

**Thread:** - A thread is the smallest unit of execution within a process. It shares the process’s memory and resources, making it a lightweight entity.



| **Aspect** | **Process** | **Thread** |
| --- | --- | --- |
| **Execution** | A process has its own separate execution context. | Threads within a process execute in the same context. |
| **Memory** | Each process has its own memory space (code, data, and stack). | Threads share the memory space of the parent process. |
| **Communication** | Communication between processes is complex (e.g., IPC). | Threads communicate easily through shared memory. |
| **Creation** | Creating a process is time-consuming and resource-intensive. | Creating threads is faster and less resource-intensive. |
| **Independence** | Processes are independent and do not affect each other. | Threads are interdependent and can affect one another. |

* Processes are managed by the operating system with mechanisms like context switching and process scheduling.
* Threads are managed either by the operating system or by the process itself (user-level threads).



1. **What are disadvantages of using multithreading?**

| **Disadvantage** | **Description** |
| --- | --- |
| Complexity | Hard to design, debug, and test multithreaded programs. |
| Race Conditions | Unpredictable results when threads access shared resources simultaneously. |
| Deadlocks | Threads waiting indefinitely for resources held by each other. |
| Context Switching Overhead | CPU time wasted in switching between threads. |
| Resource Consumption | Threads use stack memory and other system resources. |
| Non-Determinism | Unpredictable thread execution order. |
| Shared Data Challenges | Synchronization issues when accessing shared data. |
| Scalability | Too many threads can degrade performance. |
| Priority Inversion | Low-priority threads holding resources needed by high-priority threads. |
| Maintenance Difficulty | Hard to understand and maintain multithreaded code. |

1. **What are different types of threads?**

Threads in Java can be categorized into two main types based on their role and lifecycle:

* **User Threads**: - they are normal threads that perform application level tasks, are independent and keep the application running until their execution completes.
* **Daemon Threads:** - they are background threads that provide supporting services like garbage collection, monitoring and background logging. A daemon thread will stop running as soon as all user threads terminate, even if the daemon thread's task is incomplete.

| **Aspect** | **User Thread** | **Daemon Thread** |
| --- | --- | --- |
| **Role** | Performs main tasks of the application. | Performs background tasks (e.g., garbage collection). |
| **Keeps JVM Alive** | Yes, JVM runs until all user threads finish. | No, JVM exits if only daemon threads remain. |
| **Termination** | Runs until task completion. | Terminates when all user threads finish. |
| **Default Type** | Threads are user threads by default. | Must be explicitly marked as daemon. |
| **Use Cases** | Core application tasks like computations. | Background tasks like logging, monitoring. |

1. **What is difference between multithreading and multitasking?**

Multithreading refers to running multiple threads within a single process, while multitasking refers to running multiple independent processes concurrently. Threads share the same memory space, whereas processes have separate memory spaces.

1. **What is difference sleep and wait?**

| **Aspect** | **Thread.sleep()** | **Object.wait()** |
| --- | --- | --- |
| **Purpose** | Pauses the current thread for a specified duration. | Causes the current thread to wait until it is notified or interrupted. |
| **Belongs To** | **Thread class**. | **Object class** (works with thread synchronization). |
| **Thread Lock** | Does **not release the lock** if the thread holds one. | **Releases the lock** on the synchronized object it is waiting on. |
| **Usage Context** | Used to pause execution for a specific time (e.g., delays). | Used for thread communication (e.g., producer-consumer scenarios). |
| **Notification** | Does not require notification; resumes automatically after the duration. | Must be explicitly notified using notify() or notifyAll(). |
| **Interrupt Handling** | Can throw InterruptedException. | Can throw InterruptedException. |

class WaitNotifyExample {  
 public static void main(String[] args) {  
 final Object lock = new Object();  
  
 Thread thread1 = new Thread(() -> {  
 synchronized (lock) {  
 try {  
 System.*out*.println("Thread is waiting...");  
 lock.wait(); // Wait until notified  
 System.*out*.println("Thread resumed!");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread thread2 = new Thread(() -> {  
 synchronized (lock) {  
 System.*out*.println("Thread is notifying...");  
 System.*out*.println("Thread1 state is " + thread1.getState());  
 lock.notify(); // Notify the waiting thread  
 }  
 });  
  
 thread1.start();  
 try { Thread.*sleep*(1000); } catch (InterruptedException e) { e.printStackTrace(); }  
 thread2.start();  
 }  
}

//Thread is waiting...  
//Thread is notifying...  
//Thread1 state is WAITING  
//Thread resumed!

public class SleepExample {  
 public static void main(String[] args) {  
 System.*out*.println("Thread is going to sleep...");  
 try {  
 Thread.*sleep*(2000); // Sleep for 2 seconds  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("Thread woke up!");  
 }  
}  
//Thread is going to sleep...  
//Thread woke up!

1. **What is thread safety?**

A thread-safe class or method guarantees correct behavior when accessed by multiple threads simultaneously.

// Not thread-safe  
int count = 0;  
public void increment() {  
 count++;  
}

Use synchronization or AtomicInteger for thread safety.

1. **What is context switching in thread?**

Context switching is the process where the CPU switches from one thread to another, saving and restoring its state. It’s expensive, so excessive context switching can degrade performance.

1. **What will happen if start() is called twice on same thread?**

It throws a **java.lang.IllegalThreadStateException**. A thread can be started only once in it’s lifecycle.

Thread t = new Thread(() -> {});  
t.start();  
t.start(); // IllegalThreadStateException

**What is output for this?**

package com.practice.LearnSpring;  
  
class MyThread extends Thread{  
 //thread.start runs run function ( we must override this in our thread class)  
 public void run(){  
 System.*out*.println("Hi from " + Thread.*currentThread*().getName());  
 for(int i=0;i<3;i++){  
 System.*out*.println(i + "=>" + Thread.*currentThread*().getName());  
 }  
 }  
}  
  
class LearnThread{  
 public static void main(String[] args) throws InterruptedException {  
 MyThread m1 = new MyThread();  
 MyThread m2 = new MyThread();  
  
 m1.start();  
 m1.join();  
  
 m2.start();  
 m2.join();  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 }  
}

Ans: Join stops the execution of thread from which other thread is called. So, here m1 is called from main thread, and on line m1.join(), execution of main thread is paused till m1 executes. So m2 is not started till m1 executes. Later same happens with m2. So first m1 is executed then m2 then last line.

Output:

Hi from Thread-0

0=>Thread-0

1=>Thread-0

2=>Thread-0

Hi from Thread-1

0=>Thread-1

1=>Thread-1

2=>Thread-1

Hello from main

**Q. What is output for this?**

class MyThread extends Thread{  
 //thread.start runs run function ( we must override this in our thread class)  
 public void run(){  
 System.*out*.println("Hi from " + Thread.*currentThread*().getName());  
 for(int i=0;i<3;i++){  
 System.*out*.println(i + "=>" + Thread.*currentThread*().getName());  
 }  
 }  
}  
  
class LearnThread{  
 public static void main(String[] args) throws InterruptedException {  
 MyThread m1 = new MyThread();  
 MyThread m2 = new MyThread();  
  
 m1.start();  
 m2.start();  
 m1.join();  
 m2.join();  
  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 }  
}

join is written after m1 and m2 started. So output from m1 and m2 combines and at end main thread runs

Hi from Thread-0

Hi from Thread-1

0=>Thread-1

1=>Thread-1

2=>Thread-1

0=>Thread-0

1=>Thread-0

2=>Thread-0

Hello from main