**Multi-Threading**

**─**

Contents

[**1. What is Multithreading?** 2](#_Toc183876390)

[**2. Java Thread Model** 3](#_Toc183876391)

[# Thread Class 3](#_Toc183876392)

[# Runnable Interface 3](#_Toc183876393)

[# Using Lambda Expression 4](#_Toc183876394)

[**3. Important Thread methods** 5](#_Toc183876395)

[# Thread Lifecycle Methods 5](#_Toc183876396)

[# Thread Methods 9](#_Toc183876397)

[**4. Interview Questions** 13](#_Toc183876398)

## **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.

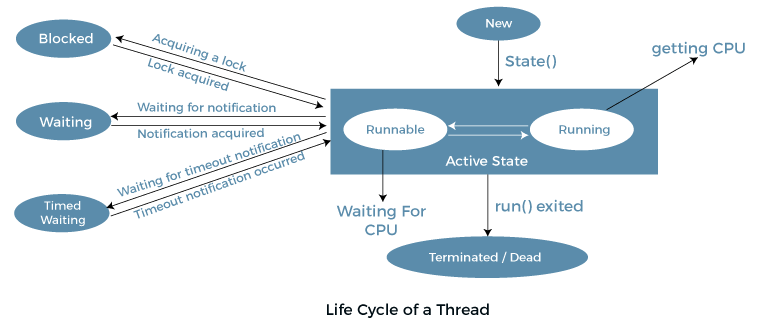
class MyThread implements Runnable{  
  
 @Override  
 public void run() {  
 System.*out*.println("Thread " + Thread.*currentThread*().getName() + " Priority " + Thread.*currentThread*().getPriority());

}  
}  
  
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.

## **4. 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:

* 1. **User Threads**: - they are normal threads that perform application level tasks, are independent and keep the application running until their execution completes.
  2. **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 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!