Java Multithreading
For Java Developers

(Topic 1) Multitasking

Multitasking allows several activities to occur concurrently (at the same time) on the computer.

There are 2 types of multitasking that exists in computer terms :-

(a) Process-based multitasking
(b) Thread- based multitasking

(a) Process - based multitasking

Running multiple programs (processes) at the same time on the computer.

(b) Thread - based Multitasking

Performing multiple functions within a program on the computer.

Example : MS word that is printing and formatting text at the same time.

Topic 2) Threads V/S Process

VVVI
*****

	Thread	Process
(1)	Smallest part of the program that can run at the same time as others.	A process is an instance of a program that is being executed.
(2)	Two threads share the same address space.	A process do have a different memory/address space.
(3)	Context switching between the threads less expensive.	Context switching between the threads is expensive.
(4)	Easier and faster communication as threads share data within the same process.	Slower communication due to inter-Process communication (IPC) mechanisms.

Topic 3) Why do we need multithreading ?

(a) In a single-threaded environment, only one task at a time can be performed.

(b) CPU cycles are wasted for example when waiting for user input.

(c) Multitasking allows idle CPU time to be put to good use.

Topic 4) Threads

A thread is an independent sequential path of execution within a program.

(Or)

A thread is the smallest unit of execution within a process.

(b) Many threads can run concurrently (at the same time) within a program.

(c) At runtime, threads in a program exist in a common memory space and can share both data and code (i.e. Threads are lightweight compared to processes).

(d) They also share the process running the program.

3 Important concepts related to multithreading in Java

(1) Creating a Threads in Java / Implementing multithreading

(2) Accessing common data and code through Synchronization.

(3) Transitioning between thread states.

Topic 5) **Main Thread**

(a) When a java program starts, one thread begins running immediately known as Main thread. (Automatically created)

(b) This thread is responsible for executing the main method of a program.

(c) If no other user threads are created, the program terminates when the main() method finishes executing.

(d) All other threads, called child threads are created from main thread itself.

(e) The main() method completes it's execution but the program will still continue running if the other threads not finished running their tasks.

(f) Here comes to more concept User threads and daemon threads.

Package :→ MainThread
File :→ TestMainThread.java

	Feature	User Thread
①	Definition	A normal thread that performs application level tasks.
②	Lifespan	Runs until it completes it's tasks (or) is explicitly stopped.
③	Priority	Higher priority, executes essential program logic.
④	Examples	Main thread worker threads handling requests

	Feature	User Thread	Daemon Thread
①	Definition	A normal thread that performs application level tasks.	A background thread that provides services to user threads.
②	Lifespan	Runs until it completes it's tasks (or) is explicitly stopped.	Runs in the background and terminates when all user threads finish.
③	Priority	Higher priority, executes essential program logic.	Lower priority, used for background tasks.
④	Examples	Main thread worker threads handling requests	Garbage Collector (GC) finalizer thread

Topic 6) **Thread creation in Java**

- A thread in java is represented by an object of the Thread class.

- Two ways to create thread in Java
  - (a) Extend the thread class
  - (b) Implement the runnable Interface

Package → ThreadCreation.ExtendTheThread
File → Thread1
File2 → TestThreadCreation

(a) **Extend the thread class**

(1) start() → It informs JVM (Java Virtual machine) that user is ready to start the thread whenever JVM feels free it will start the execution.

(2) Here two threads will be running
- (2.1) main thread also known as user thread runs automatically as code execution begins.
- ~~(2.2) Child of Custom~~
- (2.2) Child for main thread will be custom thread (i.e. --- thread1) As thread1 created from main thread.
- (2.3) No order maintained for executing --- depends on JVM in which order thread getting executed.

(2.4) Main thread is a user thread.

VVI (2.5) NOTE :-

If JVM finds no user user threads running then JVM might not allow daemon thread (`thread1.setDaemon(true);`) to get executed.

↳ But if the JVM finds any user thread apart from main thread running it will continue running until all the user threads finishes it's job (or) gets executed.

`Thread.currentThread()`

↳ O/P → Thread[thread1, 5, main]

- name of the thread
- Priority of the thread (default is 5)
- The thread group to which this thread belongs (Parent of the thread)

`Thread.currentThread().getname()`

↳ O/P = thread1

(b) Implement the runnable interface

→ Runnable Interface is a functional interface that has one abstract methods that is run().

→ If any class that implements Runnable has to implement the abstract method run().

Question for Interview

There are two ways of ~~implementing~~ creating threads in Java. Which one to prefer?

↳ (a) Extend the thread class
↳ (b) Implement the runnable Interface

Ans:- We should implement runnable interface for creating threads because there is no constraints(limitations) as we can ~~create~~ do multiple implementations for a particular class but we can't extend multiple classes that's why extending a thread add some constraints to our design.

Package → ThreadCreation.RunnableInterface
File → Testing.java
File2 → Thread2.java

~~Package → .R~~

Package → ThreadCreation.RunnableInterfaceUsinglambdaExpression
File → Testing.java
File2 → Thread2.java

Topic) 7.7) Thread safety

Definition

Thread safety means that a program (or) code segment can be safely accessed by multiple threads without leading to race conditions, data corruption (or) unexpected behavior.

Ex → StringBuffer

Why is Thread safety Important?

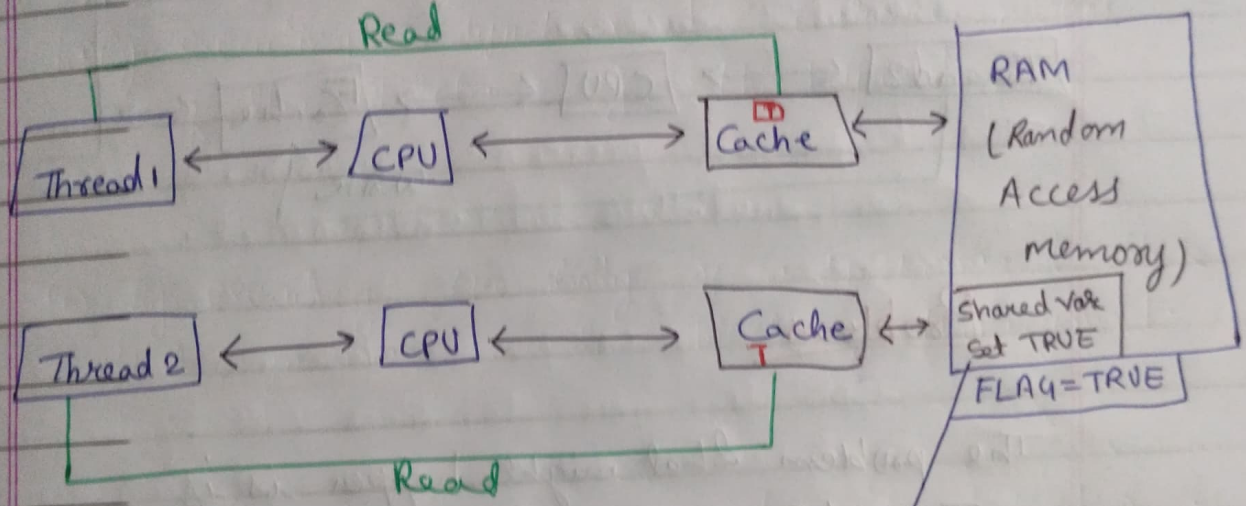
In a multithreaded environment, multiple threads can read and ~~write~~ modify shared data simultaneously.

Topic → The Volatile Keyword

WVI
xxxxx

Internal Implementation of Volatile Keyword

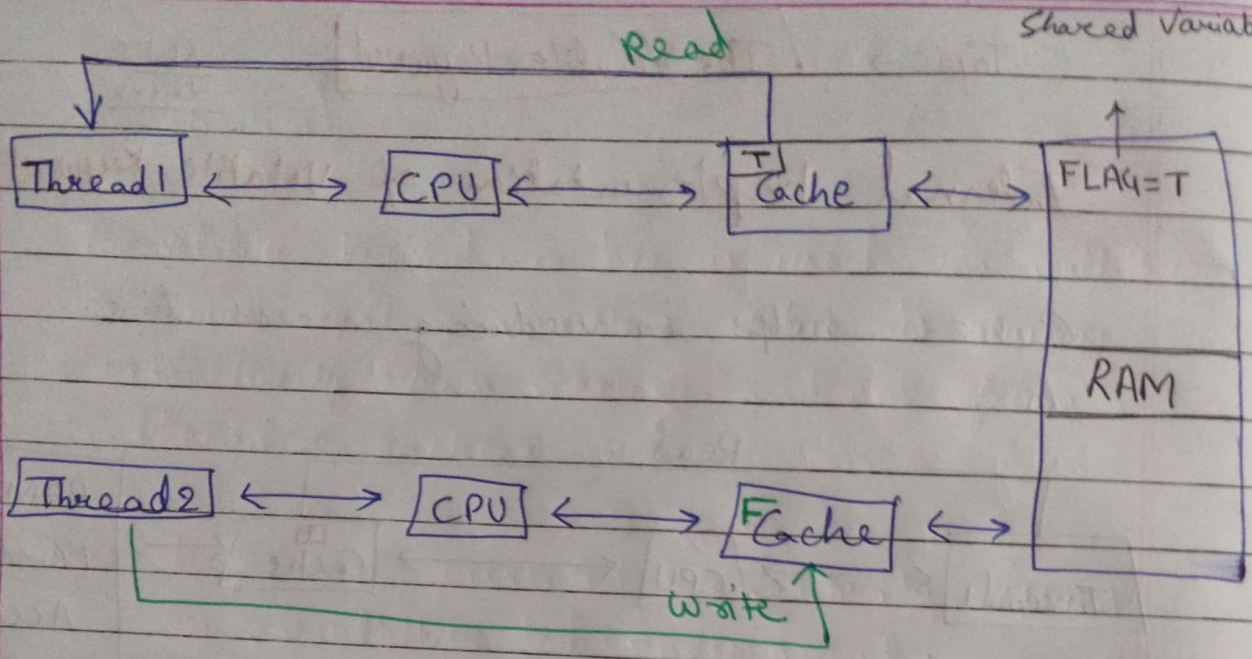
Cache → helps in reducing access time.



Consider a shared variable TRUE that is to be accessed by the different thread (t1 & t2).

→ Now both the threads directly don't interact (read) the variables from the RAM.

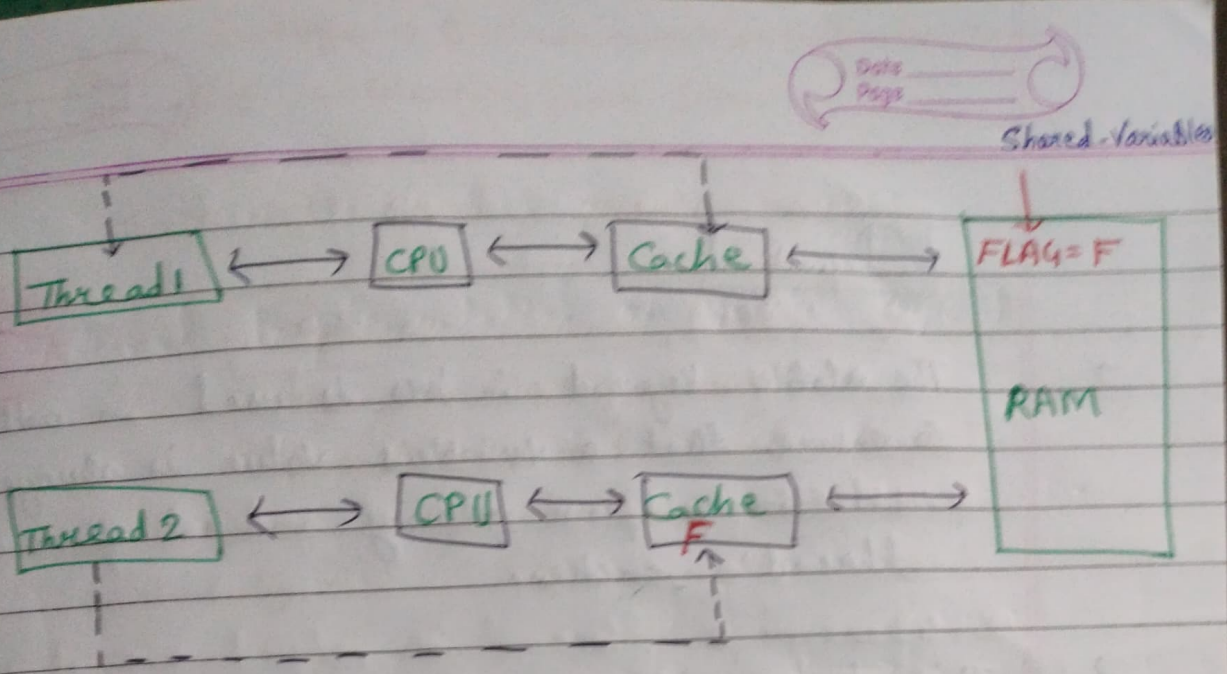
→ They uses Cache to read the value of the FLAG=TRUE ~~from the~~ locally.



Problems

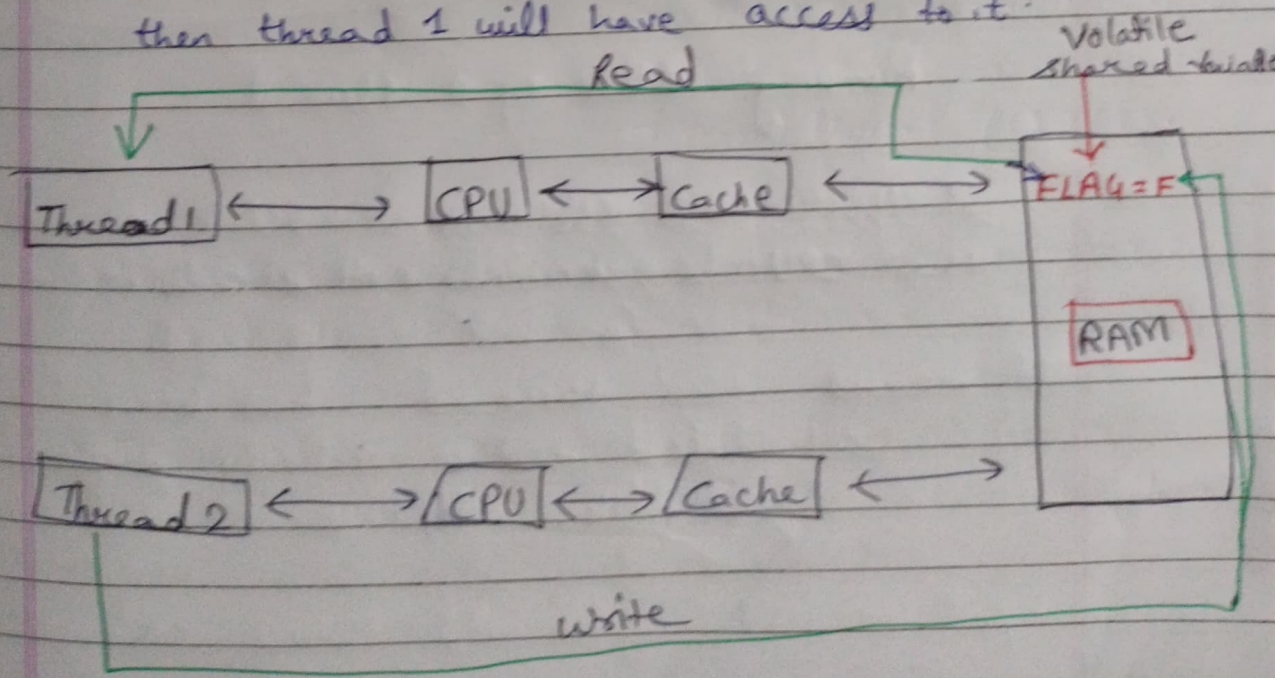
The problem that might happen is if Thread 2 changes the value of FLAG=FALSE

- It won't directly update into the RAM.
- First it update in it's local cache.
  (Still seen FLAG=True for Main memory & locally cache for Thread 1)
- After certain time/ Slowly inside the Main memory RAM FLAG=False gets updated.
- But thread 1 still don't have any visibility that FLAG updated to false.



- In order to get rid off this problem Volatile keyword came into the picture.

- Declaring volatile shared variable FLag = T, now these threads (t_1 & t_2) directly reads from the Main memory i.e. RAM as a result thread 2 changes the flag to FALSE then thread 1 will have access to it.



Date
Page

Definition

→ The volatile keyword in Java is used in multithreading to ensure that a variable's value is always read directly from memory and not from a thread's cache.

→ This helps in preventing visibility issues in multithreaded environment.

Package → ProducerConsumerPatternProblems

File :- BlockingQueue.java

Date
Page

Topic) VVVI (xxxxx) Interview questions

Producer Consumer Pattern Problem

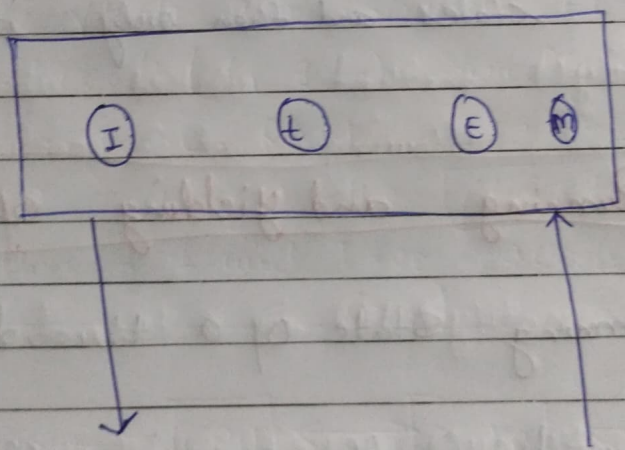
(designing a Blocking queue)

(Introduction to wait() and notify())

→ Let's consider a queue having two threads t_1 and t_2.

→ t_1 is trying to push items to the queue (enqueue).

→ t_2 is trying to omit items from the queue (dequeue).



1(or) more Consumer

1(or) more producers

→ Queue follow First In - First Out (FIFO) rules.

→ We can only push items to the queue is space available.

→ We can only remove items from the queue if items available in the queue.

For add();

NOTE :-

→ Whenever a thread is notified when it is awakened
→ It does not directly just into the runnable state.
→ It has to fight for the lock that means threads goes to block full of acquistion states and then only it starts executing.

Running and yielding of a thread

Running State of a thread

→ A thread in the running state when the CPU is executing Pt's code.
→ It means the thread has been scheduled and is actively performing it's task.

Yielding of a thread

The yield() in Java pauses the currently running thread and gives a chance for other threads of the same (or) higher priority to execute.

Sleeping a Thread (sleep() method)

The sleep() method pauses the thread for a specified time, making it temporarily inactive.

Waking up a Thread

A sleeping thread can wake up in two ways

(1) Automatically: after the sleep time ends

(2) Manually: if another thread calls interrupt(), causing an InterruptedException

Waiting and Notifying

Waiting (wait()) in a Thread

→ The wait() makes a thread pause until another thread notifies it.

→ The thread releases the lock and waits until it is notified.

Notifying (notify()) in a Thread

The notify() method wakes up one waiting thread, allowing it to continue execution.

notifyAll() in a thread

The notifyAll() wakes up all waiting threads but only one will proceed at a time (as per lock availability).

Important methods() for wait and notify

- final void wait (long timeout) throws InterruptedException.
- final void wait (long timeout, int nanos) throws InterruptedException.

- final void wait() throws InterruptedException
- final void notify()
- final void notifyAll()

Thread Timed Out

A thread timeout occurs when a thread takes too long to complete a task, leading to termination (or) interruption.

This happens due to :—

- waiting too long for a resource
- Deadlock situations
- Slow response from an external system

Thread interruption in Java

→ Thread interruption in Java is a mechanism that allows one thread to signal another thread to stop (or) change it's behavior.

→ How does a Thread get interrupted?

thread.interrupt(); (sets the interrupted flag of the thread to true

Thread timed out V/S Thread interruption in

In Thread interruption we come to know a particular thread is interrupted by some other thread.

In Thread Timed out we don't know thread got timed out (or) notified bcz no exception been thrown.

Date
Page

Topic → ~~States of a Thread~~ Thread priority

~~(1) Thread is created but not started.~~

Thread priority in java is a way to tell the CPU which thread is more important.

→ A thread with higher priority has a better chance of getting CPU time before a lower-priority thread, but it is not guaranteed.

(1-lowest)
(10-highest)

→ It does not guarantee Order of execution

Package → Threadpriority
File → TestingThreadPriority.java

Thread Scheduler → is a part of JVM (Java virtual machine) that decides which thread should run next from the pool of waiting threads.

Topic → Thread Joining

The join method is used to make the calling thread wait until the thread on which join has been called completes it's execution.

(or)

The join() method in java makes one thread wait until another thread finishes execution before continuing.

Package : ThreadJoining
File :- Thread1.java
File2 :- Thread2.java
File3 :- Thread3.java
File4 :- TestingThreadJoining.java

Deadlock in multithreading

A deadlock occurs in multithreading when two or more threads are waiting for each other to release resources, but none of them can proceed.

This results in permanent freeze of execution.

Create a deadlock in Java?

Interview Questions

Package → Deadlock
File → TestingDeadlock.java

Topic) Deadlock

- (1) What is deadlock?
- (2) Realtime Example of deadlock
- (3) Java code to understand deadlock

(1) Deadlock

(a) We have two objects and two resources.
(b) First object locked first resource and second object locked second resource.
(c) Now first object is trying to access second resource which is acquired by second object and second object is trying to access first resource which is acquired by first object.

(2) Real time example of deadlock

(a)

Objects	Resources
Desktop	Printer (Locked)
Laptop	Scanner (Locked)

Siddharth - key
Kumar - key

Kumar (lock)
Siddharth (lock)

Conditions for deadlock

1. Mutual Exclusion: only one thread can access a particular resource at a time.
2. Hold & wait: A thread holds one lock and waits for another.

Questions for multithreading

(Q1) What is countDownLatch?

→ CountDownLatch is a tool in java that makes one (or) more threads wait until some tasks are completed.

→ Eg:- Real Life example

Door locked opens only when a member arrived, until then others must wait.

→ Uses:-

↳ Useful for multi-threading
↳ Ensures all the steps are completed before moving forward.

(Q2) Print even odd numbers by using two threads. Can we use Volatile instead of synchronized method.

Package: Interviews
File: EvenOddBinderTesting.java

No, Volatile is not enough because it only ensures visibility but does not prevent race conditions (or) coordinate thread execution.
we need synchronized, wait() and notify() for proper

even-odd alternation.

(Q3) Future and CompletableFuture?

	Future	CompletableFuture
(1)	Introduced in Java 5	(1) Introduced in Java 8
(2)	Future.get() is blocking: It waits for the result, which can cause performance issues.	(2) CompletableFuture is Non-Blocking: methods like getNow(), join() and Callback-based thenApply() prevent blocking

(Q4) multithreading in Java 7 V/S Java 8

→ Java 7 relies on manual multithreading
→ Java 8 introduces Lambda, CompletableFuture, and parallel streams for better asynchronization performance.

(Q5) ~~What is reenter~~

(Q5) What is reenterentLok?

→ Rentrantlock is like a manual lock that is used instead of synchronized.

→ It allows a thread to lock multiple times without getting stuck.

→ Example

→ If a person (thread) enters, they lock the door (lock()).

→ If the same person needs to use it again (reentrent) they can still enter bcz they have the key.

(Q6) When we'll use callable and runnable interface?

runnable → When we just need to perform a task without returning any data/value.

Eg:- → creating a threads

callable → use callable when the task needs to return a value.

→ It can also throw checked exceptions.

Q7) multithreading → in java means running multiple tasks (threads) at the same time to make a program faster and more efficient.

Synchronization →

Threads share the same memory space i.e. they can also share objects.

Q8) Types of Thread Pool

	Thread Pool	Behavior
①	Fixed Thread Pool	Fixed number of threads
②	Cached Thread Pool	Creates new threads as needed
③	Single Thread Pool	one thread, sequential execution
④	Scheduled thread pool	Run tasks after a delay (or) repeatedly
⑤	Work Stealing Pool	Uses multiple queues for Parallelism

Q7) multithreading → in java means running multiple tasks (threads) at the same time to make a program faster and more efficient.

Synchronization →
Threads share the same memory space i.e. they can also share objects.

Q8) Types of Thread Pool

	Thread Pool	Behavior	Best for
①	Fixed Thread Pool	Fixed number of threads	Constant workload
②	Cached Thread Pool	Creates new threads as needed	Short-lived tasks
③	Single Thread Pool	one thread, sequential execution	Tasks that must run in order
④	Scheduled thread pool	Run tasks after a delay/or repeatedly	Scheduled tasks (cron jobs, reminders)
⑤	Work Stealing Pool	Used multiple queues for Parallelism	Large parallel computations

Pls look code from chat GPT

Q9) Give example where you will use Volatile and Atomic variables.

volatile: When a flag is shared b/w threads and needs to be read immediately.
(eg → refer implementation part).

```
private static volatile boolean running = true;
```

Atomic Integer:
When multiple threads modify a shared variable without locks.
→ Avoids race condition without explicit synchronization

Date
Page

(Q10) How Thread works on Arraylist?

→ Arraylist is not thread safe.

→ multiple threads modifying Arraylist may lead to data Inconsistency (or) ConcurrentModificationExcep

→ use Collections.synchronizedList() for basic synchro nization

→ Use CopyOnWriteArraylist for read-heavy operations with minimal writes.