**1. What is thread in Java?**

A thread in Java is a lightweight process. It is a separate path of execution that allows multiple tasks to run concurrently within a program. Java supports threads through the ***Thread*** class and the ***Runnable*** interface. Multithreading allows for better CPU utilization, enabling parallel execution of tasks.

# What is multithreading?

It is a process of executing multiple threads at the same time. Multithreading is used for multitasking purposes. It acquires less memory and provides fast and efficient performance. Multithreading’s main advantages are as follows:

Here threads share the same address space.

* lightweight thread.
* The low cost of communication between the processes.

**2. What is the difference between Thread and Runnable?**

* ***Thread*** is a class that represents a thread of execution, while ***Runnable*** is a functional interface that can be implemented by any class whose instances are intended to be executed by a thread.
* When extending ***Thread***, you cannot extend any other class, but implementing ***Runnable*** allows the class to still extend another class. ***Runnable*** is preferred in most cases because it promotes flexibility and separation of thread behavior from task execution.

**3. What are the key differences between start() and run() methods in Java threads?**

* ***start()*** : Starts the execution of a new thread, invoking the run() method internally on a separate call stack.
* ***run()*** : If called directly, runs in the current thread, not creating a new thread. It’s like a normal method call and does not create concurrency.

**4. Explain the thread lifecycle in Java**

A thread can be in one of the following states:

* **NEW**: A thread is created but not started.
* **RUNNABLE**: After calling start(), the thread is ready to run and is waiting for CPU time.
* **BLOCKED**: The thread is waiting for a lock to enter a synchronized block.
* **WAITING**: The thread is waiting indefinitely for another thread to perform a specific action.
* **TIMED\_WAITING**: The thread is waiting for a specified period (e.g., using sleep() or join(timeout) ).
* **TERMINATED**: The thread has finished execution.

**5. What is thread synchronization and why is it needed?**

Thread synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, two or more threads may corrupt the state of shared resources (like variables or data structures). Java provides synchronized blocks and methods to ensure that only one thread accesses the resource at a time, preventing race conditions.

**6. What is a synchronized block in Java?**

A ***synchronized*** block is used to lock an object for any shared resource. When a thread enters a synchronized block, it acquires a lock on the object. No other thread can access synchronized code on that object until the lock is released.

Example:

synchronized (this) {  
 // synchronized code  
}

**7. What is the difference between synchronized method and synchronized block?**

* A **synchronized method** locks the entire method, meaning only one thread can access the entire method at a time.
* A **synchronized block** is more granular, allowing you to synchronize only specific parts of a method. It is more efficient as it limits the scope of synchronization, reducing the chance of performance bottlenecks.

**8. What are deadlock, livelock, and starvation in multithreading?**

* **Deadlock**: Two or more threads are blocked forever, each waiting for the other to release a resource.
* **Livelock**: Threads continuously change their state in response to each other but cannot make progress.
* **Starvation**: A thread is unable to access a resource for an extended period because other threads are continuously occupying it.

**9. What are wait(), notify(), and notifyAll() methods?**

* **wait()**: Causes the current thread to wait until another thread calls notify() or notifyAll() on the same object.
* **notify()**: Wakes up one waiting thread.
* **notifyAll()**: Wakes up all waiting threads. These methods are used in conjunction with synchronized blocks to implement inter-thread communication.

**10. How does Java handle thread priority?**

Java allows you to set a priority for each thread (ranging from ***Thread.MIN\_PRIORITY*** to ***Thread.MAX\_PRIORITY***), which suggests to the thread scheduler the importance of a thread. However, thread priority behavior is system-dependent, and it doesn't guarantee that a higher-priority thread will always run before lower-priority threads.

**11. What is a ThreadLocal in Java?**

***ThreadLocal*** provides thread-local variables, which means each thread accessing such a variable has its own, independently initialized copy of the variable. It helps avoid issues in multithreaded environments where each thread requires its own instance of an object.

**12. What is the volatile keyword in Java?**

***volatile*** keyword ensures that a variable is read from and written to the main memory, rather than being cached by a thread. It guarantees visibility of changes to variables across threads, preventing visibility issues in a multithreaded environment.

**13. Explain Executor Framework in Java.**

The Executor framework provides a high-level API for managing threads in a multithreaded environment. It decouples task submission from thread creation, improving application performance. ***Executors*** offer services like thread pools (***ExecutorService***), which manage worker threads to execute tasks asynchronously.

**14. How would you create a thread in Java using the Runnable interface?**

class MyRunnable implements Runnable {  
 public void run() {  
 System.out.println("Thread is running.");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Thread thread = new Thread(new MyRunnable());  
 thread.start(); // Creates a new thread and starts it  
 }  
}

**15. Implement producer-consumer problem using wait/notify.**

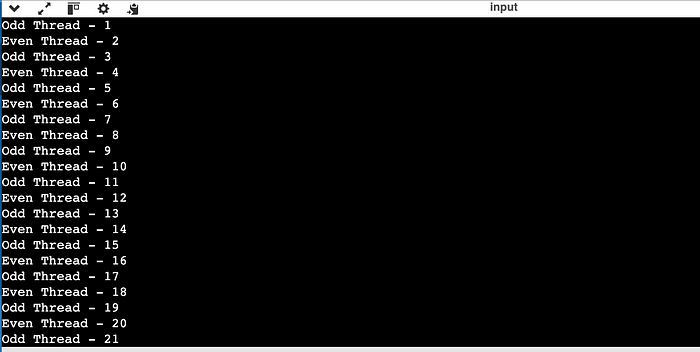
class SharedResource {  
 private int value;  
 private boolean available = false;  
  
 public synchronized void produce(int val) throws InterruptedException {  
 while (available) {  
 wait(); // Wait until consumed  
 }  
 this.value = val;  
 available = true;  
 System.out.println("Produced: " + val);  
 notify();  
 }  
  
 public synchronized void consume() throws InterruptedException {  
 while (!available) {  
 wait(); // Wait until produced  
 }  
 System.out.println("Consumed: " + value);  
 available = false;  
 notify();  
 }  
}  
  
class Producer implements Runnable {  
 private final SharedResource resource;  
  
 public Producer(SharedResource resource) {  
 this.resource = resource;  
 }  
  
 public void run() {  
 try {  
 for (int i = 0; i < 5; i++) {  
 resource.produce(i);  
 Thread.sleep(100);  
 }  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 }  
 }  
}  
  
class Consumer implements Runnable {  
 private final SharedResource resource;  
  
 public Consumer(SharedResource resource) {  
 this.resource = resource;  
 }  
  
 public void run() {  
 try {  
 for (int i = 0; i < 5; i++) {  
 resource.consume();  
 Thread.sleep(100);  
 }  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 }  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 SharedResource resource = new SharedResource();  
 Thread producer = new Thread(new Producer(resource));  
 Thread consumer = new Thread(new Consumer(resource));  
   
 producer.start();  
 consumer.start();  
 }  
}

**15. Write a program to print numbers from 1 to 100 using two threads with one thread printing odd numbers and one thread printing even numbers**

**(** **Most Important Question)\*\*\***

class PrintNumbers {  
 private int number = 1;  
 private final int MAX = 100;  
   
 // Method for odd thread  
 public synchronized void printOdd() throws InterruptedException {  
 while (number < MAX) {  
 // If number is even, wait for even thread to print  
 while (number % 2 == 0) {  
 wait();  
 }  
 System.out.println(Thread.currentThread().getName() + " - " + number);  
 number++; // Increment after printing  
 notify(); // Notify the other thread to continue  
 }  
 }  
  
 // Method for even thread  
 public synchronized void printEven() throws InterruptedException {  
 while (number <= MAX) {  
 // If number is odd, wait for odd thread to print  
 while (number % 2 != 0) {  
 wait();  
 }  
 System.out.println(Thread.currentThread().getName() + " - " + number);  
 number++; // Increment after printing  
 notify(); // Notify the other thread to continue  
 }  
 }  
}  
  
public class Main{  
 public static void main(String[] args) {  
 PrintNumbers printNumbers = new PrintNumbers();  
   
 // Odd number thread  
 Thread oddThread = new Thread(() -> {  
 try {  
 printNumbers.printOdd();  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 }  
 }, "Odd Thread");  
   
 // Even number thread  
 Thread evenThread = new Thread(() -> {  
 try {  
 printNumbers.printEven();  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 }  
 }, "Even Thread");  
   
 // Start both threads  
 oddThread.start();  
 evenThread.start();  
 }  
}

Output:



# What is multithreading?

It is a process of executing multiple threads at the same time. Multithreading is used for multitasking purposes. It acquires less memory and provides fast and efficient performance. Multithreading’s main advantages are as follows:

Here threads share the same address space.

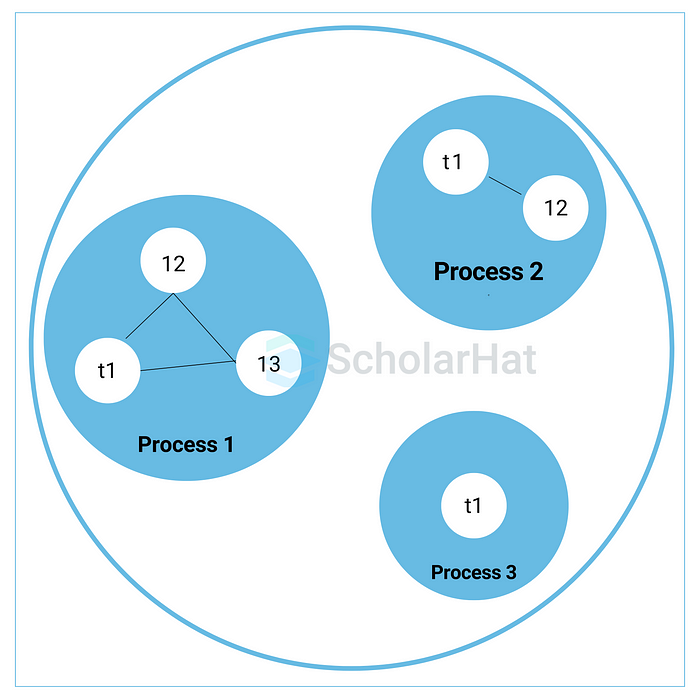
* lightweight thread.
* The low cost of communication between the processes.

# 2. What is the thread?

A thread is nothing but a lightweight subprocess. It is a different path of execution. Each and every thread runs in a different-different stack frame. The process can contain multiple threads. Threads share usually the process resources, but still, they execute independently everytime.

# 3. Differentiate between process and thread.

* A Program in the execution is called the process while A thread is a subset of the process.
* Processes are independent while threads are the subset of process.
* Processes have different address spaces in memory, on the other hand, threads contain a shared address space.
* Context switching is much faster between the threads as compared to processes.
* The change in the Parent process doesn’t affect the child process while in the parent thread changes can affect the child thread.



# 4. What do you understand by inter-thread communication?

* inter-thread communication is nothing but the process of communication between synchronized threads.
* It is used to avoid thread polling in Java.
* The thread is paused running in its critical section, and another thread is allowed to enter or it lock the same critical section to be executed.
* inter-thread communication can be achieved by wait(), notify(), and notifyAll() methods.

# 5. What is the purpose of the wait() method in Java?

It is the method provided by the Object class in Java. It is used for inter-thread communication in Java. java. lang.Object. wait(), this method is used to pause the current thread and wait until another thread does not call the notify() or notifyAll() method. The syntax of the wait() method in javs is given below.

## Syntax

# 6. Why must the wait() method be called from the synchronized block

We have to call the wait method otherwise it will throw java.lang.IllegalMonitorStateException type exception. Because of this, we need the wait() method for inter-thread communication with notify() and notifyAll() methods. Hence It must be present in the synchronized block for proper and correct communication.

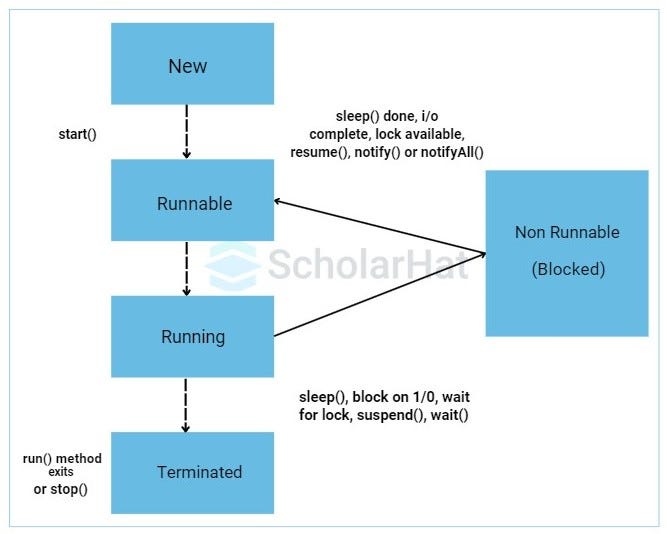
# 7. What are the advantages of multithreading?

Multithreading programming has the following advantages:

* It allows an application/program to be always reactive for input, even if already running with some background tasks
* It allows the faster execution of tasks, as threads execute independently.
* It provides better utilization of cache memory as threads share the common memory resources.
* It reduces the number of the required server as one server can execute multiple threads at a time.

# 8. What are the states in the lifecycle of a Thread?

* **New**: In this state, a Thread class object is created using a new operator, but the thread is not alive. The thread doesn’t start until we call the start() method.
* **Runnable**: In this state, the thread is run after calling the start() method. However, the thread is not yet selected by the thread scheduler.
* **Running**: In this state, the thread scheduler picks the thread from the ready state.
* And the thread is running.
* **Waiting/Blocked**: In this state, a thread is not running but still alive, or it is waiting for the other thread to finish.
* **Dead/Terminated**: A thread is in terminated or dead state when the run() method exits.



# 9. Explain the difference between preemptive scheduling and time slicing.

If we talk about preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. On the other hand, time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. Then the scheduler determines which task should execute next, based on priority and other factors.

# 10. What is context switching?

It is the state of the process (or thread) is stored so that it can be restored. The execution can be resumed from the same point after that. Context switching provides multiple processes to share the same CPU.

# 11. Differentiate between the Thread class and Runnable interface for creating a Thread.

* The thread can be created by extending the Thread class
* The thread can be created by implementing the Runnable interface

However, the primary differences between both ways are given below:

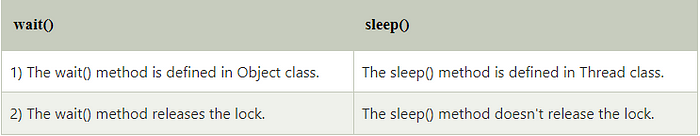
# 12. What does join() method?

This method waits for a thread to die. It occurred the currently running threads to stop executing until the thread it joins with completes its task. The join() method is overloaded in the Thread class in the following ways.

* The public void join() method throws InterruptedException.
* The public void join(long milliseconds) method throws InterruptedException.

# 13. Describe the purpose and working of the sleep() method.

# 14.What is the difference between wait() and sleep() method?



# 15.Can we call the run() method instead of start()?

# 16. What about the daemon threads?

* The daemon threads are the low-priority threads that provide the background support and services to the user threads.
* Daemon thread gets automatically terminated by the JVM if the program remains with the daemon thread only, and all other user threads are ended/died.
* There are two methods for daemon thread available in the Thread class:
* public void setDaemon(boolean status): It is used to mark the thread daemon thread or a user thread.
* public boolean daemon (): It checks whether the thread is a daemon or not.

# 17. What are the two ways of implementing thread in Java?

There are basically two ways of implementing thread in Java as given below:

**1. Extending the Thread class**

## Example

public void run () { System.out.println("Welcome to Scholarhat."); } public static void main (String args[]) { MultithreadingDemo obj=new MultithreadingDemo(); obj.start(); } }

## Output

**2. Implementing Runnable interface in Java**

## Example

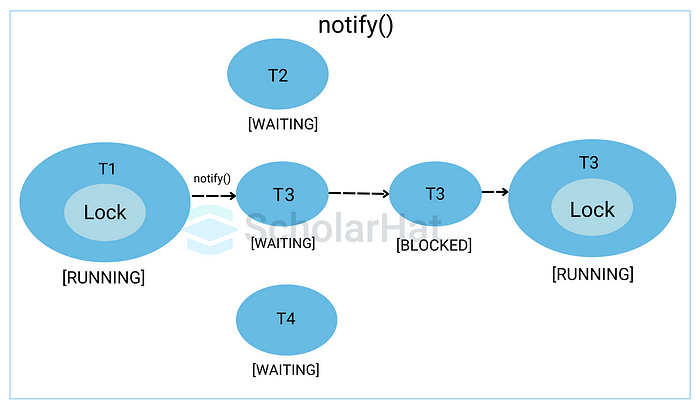
class MultithreadingDemo implements Runnable { public void run () { System.out.println("Welcome to Scholarhat.."); } public static void main (String args[]) { MultithreadingDemo obj=new MultithreadingDemo(); Threadtobj =new Thread(obj); tobj.start(); } }

## Output

# 18.What’s the difference between notify() and notifyAll()?

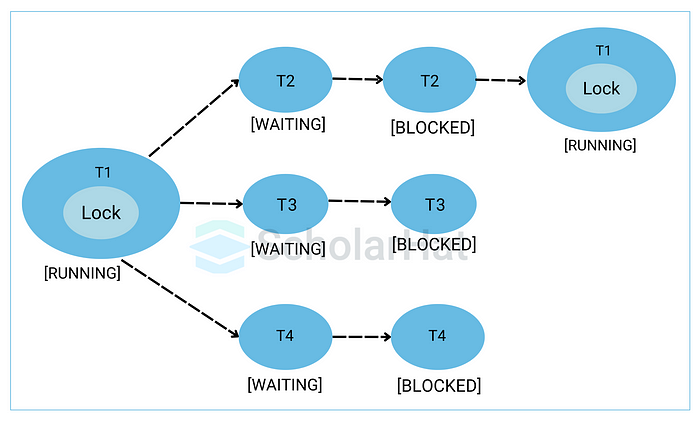
## 1.notify():

* It sends a notification and wakes up only a single thread instead of multiple threads that are waiting on the object’s monitor.
* The notify() method is defined in the Object class, which is Java’s top-level class.
* It’s used to wake up only one thread that’s waiting for an object, and that thread then begins execution.
* The thread class notify() method is used to wake up a single thread.



## 2.notifyAll():

* It sends notifications wakes up all threads and allows them to compete for the object’s monitor instead of a single thread.
* The notifyAll() wakes up all threads that are waiting on this object’s monitor.
* A thread waits on an object’s monitor by calling one of the wait methods.
* The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.

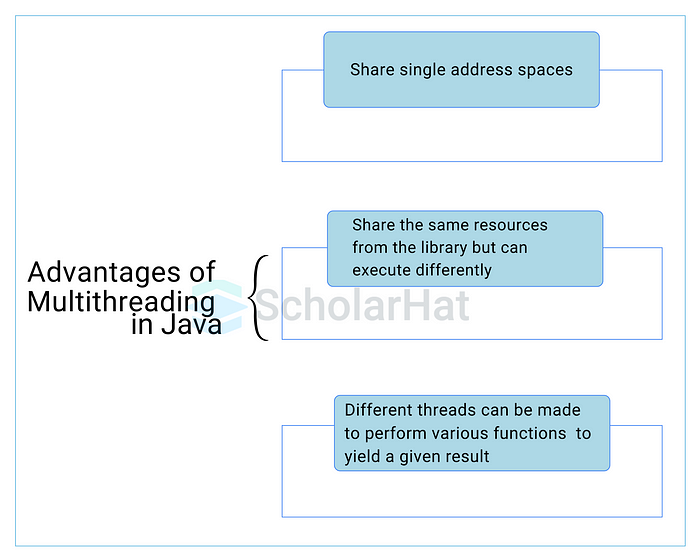


# 19. What do you understand about a “thread” in Java programming?

A “thread is the smallest unit of processing and is a lightweight subprocess executed independently. Each thread in the program runs on a different stack frame, and multiple threads form the complete program.

# 20. What are some fundamental advantages of multithreading in Java?

The key benefits of Java multithreading include:



# 21. What is the fundamental difference between a process and a thread?

A process is part of the main program, while threads are subsets of processes. Processes have different address spaces in the memory, while threads share the same address space.

# 22. What is inter-thread communication?

Communication between synchronized threads is referred to as inter-thread communication. It is a core feature that helps to avoid thread-polling in Java. A particular thread can be paused through inter-thread communication to allow another thread to enter the block.

# 23. What are some functions used to perform inter-thread communication in Java?

This is one of the most common Java Multithreading interview questions asked in tech interviews. Some common functions used to perform inter-thread communication in Java are — notify(), wait(), and notifyAll().

# 24. What do you understand about the wait() function in Java?

The wait() function, specified by the object class, is used to pause the current thread and wait until another thread calls the notify() function. Note that the wait() method needs to be called from a synchronized block to avoid an exception from occurring.

# 25. What do you understand by context switching?

Context switching is a feature through which the current state of a thread is saved for it to be restored and executed later. Through context switching, multiple processes can share the same CPU.

# 26. What is the function of the join() method?

The join() method causes the current thread to stop running until the thread due congregates and completes its task. It is a commonly used function that facilitates the execution of multiple threads in an organized manner.

# 27. What is the function of the sleep() method?

This is yet another popular Java Multithreading Interview Question. The sleep() method is used to pause a particular thread from executing and prioritizes another thread that needs to be executed before the current thread executes.

# 28. What do you understand about Deadlock situations in Java Multithreading?

Deadlock is a situation where each thread is waiting for resources held by other waiting threads. Due to this situation, no threads are executed, causing a pause in program execution and breaking the code at runtime.

# 29. How do you detect deadlock situations in Java?

Deadlock situations can be detected by running the executable code on cmd and subsequently collecting the thread dump. If deadlock situations are present, the cmd will throw up a message.

# 30. How can deadlock situations be avoided in Java?

This is one of the most common Java Multithreading interview questions asked in technical interviews. Deadlock situations in Java can be avoided by:

* By way of avoiding nested thread locks and providing locks to only one thread at a time
* By using thread join — the function helps to wait for threads to execute before other threads are executed, thereby preventing multiple threads from waiting for resources used by other threads.

# Java Multithreading interview questions for experienced

# 31. How do threads communicate with each other?

Threads can communicate using three methods i.e., wait(), notify(), and notifyAll().

# 32. Can two threads execute two methods (static and non-static concurrently)?

Yes, it is possible. If both the threads acquire locks on different objects, then they can execute concurrently without any problem.

# 33. What is the purpose of the finalize() method?

The finalize () method is basically a method of Object class specially used to perform cleanup operations on unmanaged resources just before garbage collection. It is not at all intended to be called a normal method. After the complete execution of the finalize() method, the object gets destroyed automatically.

# 34. What is the synchronized method and synchronized block? Which one should be preferred?

Synchronized Method: In this method, the thread acquires a lock on the object when they enter the synchronized method and releases the lock either normally or by throwing an exception when they leave the method. No other thread can use the whole method unless and until the current thread finishes its execution and release the lock. It can be used when one wants to lock on the entire functionality of a particular method.

# 35. What is Livelock? What happens when it occurs?

Similar to deadlock, livelock is also another concurrency problem. In this case, the state of threads changes between one another without making any progress. Threads are not blocked but their execution is stopped due to the unavailability of resources.

# 36. What is BlockingQueue?

* BlockingQueue basically represents a queue that is thread-safe.
* The producer thread inserts resource/element into the queue using put() method unless it gets full and consumer thread takes resources from the queue using take() method until it gets empty.
* But if a thread tries to dequeue from an empty queue, then a particular thread will be blocked until some other thread inserts an item into the queue, or if a thread tries to insert an item into a queue that is already full, then a particular thread will be blocked until some threads take away an item from the queue.
* Synchronized Block: In this method, the thread acquires a lock on the object between parentheses after the synchronized keyword, and releases the lock when they leave the block. No other thread can acquire a lock on the locked object unless and until the synchronized block exists. It can be used when one wants to keep other parts of the programs accessible to other threads.
* Synchronized blocks should be preferred more as it boost the performance of a particular program. It only locks a certain part of the program (critical section) rather than the entire method and therefore leads to less contention.

## Example

package org.arpit.java2blog; import java.util.concurrent.ArrayBlockingQueue; import java.util.concurrent.BlockingQueue; public class BlockingQueuePCExample { public static void main (String[] args) { BlockingQueue queue=new ArrayBlockingQueue<>(5); Producer producer=new Producer(queue); Consumer consumer=new Consumer(queue); Thread producerThread = new Thread(producer); Thread consumerThread = new Thread(consumer); producerThread.start(); consumerThread.start(); } static BlockingQueue queue=null; public Producer (BlockingQueue queue) { super(); this.queue = queue; } @Override public void run () { try { System.out.println("Producing element 1"); queue.put("Element 1"); Thread.sleep(1000); System.out.println("Producing element 2"); queue.put("Element 2"); Thread.sleep(1000); System.out.println("Producing element 3"); queue.put("Element 3"); } catch (InterruptedException e) { e.printStackTrace(); } } } static BlockingQueue queue=null; public Consumer (BlockingQueue queue) { super(); this.queue = queue; } @Override public void run () { while(true) { try { System.out.println("Consumed "+queue.take()); } catch (InterruptedException e) { e.printStackTrace(); } } } } }

## Output

Producing element 1 Consumed Element 1 Producing element 2 Consumed Element 2 Producing element 3 Consumed Element 3

# 37. What is thread starvation?

Thread starvation is basically a situation or condition where a thread won’t be able to have regular access to shared resources and therefore is unable to proceed or make progress. This is because other threads have high priority and occupy the resources for too long. This usually happens with low-priority threads that do not get CPU for its execution to carry on.

# 38. Can you start a thread twice?

No, it’s not at all possible to restart a thread once a thread gets started and completes its execution. Thread only runs once and if you try to run it for a second time, then it will throw a runtime exception i.e., java.lang.IllegalThreadStateException.

# 39. Explain context switching.

Context switching is basically an important feature of multithreading. It is referred to as switching of CPU from one thread or process to another one. It allows multiple processes to share the same CPU. In context switching, the state of thread or process is stored so that the execution of the thread can be resumed later if required.

# 40. What is CyclicBarrier and CountDownLatch?

# 41. What do you mean by inter-thread communication?

Inter-thread communication, as the name suggests, is a process or mechanism using which multiple threads can communicate with each other. It is especially used to avoid thread polling in java and can be obtained using wait(), notify(), and notifyAll() methods.

# 42. What is Thread Scheduler and Time Slicing?

Thread Scheduler: It is a component of JVM that is used to decide which thread will execute next if multiple threads are waiting to get the chance of execution. By looking at the priority assigned to each thread that is READY, the thread scheduler selects the next run to execute. To schedule the threads, it mainly uses two mechanisms: Preemptive Scheduling and Time slicing scheduling.

Time Slicing: It is especially used to divide CPU time and allocate them to active threads. In this, each thread will get a predefined slice of time to execute. When the time expires, a particular thread has to wait till other threads get their chances to use their time in a round-robin fashion. Every running thread will get executed for a fixed time period.

# 43. What is a shutdown hook?

A shutdown hook is simply a thread that is invoked implicitly before JVM shuts down. It is one of the most important features of JVM because it provides the capacity to do resource cleanup or save application state JVM shuts down. By calling the halt(int) method of the Runtime class, the shutdown hook can be stopped. Using the following method, one can add a shutdown hook.

# 44. What is busy spinning?

Busy Spinning, also known as Busy-waiting, is a technique in which one thread waits for some condition to happen, without calling wait or sleep methods and releasing the CPU. In this condition, one can pause a thread by making it run an empty loop for a certain time period, and it does not even give CPY control. Therefore, it is used to preserve CPU caches and avoid the cost of rebuilding cache.

# 45. What is ConcurrentHashMap and Hashtable? In java, why is ConcurrentHashMap considered faster than Hashtable?

# 46. Explain thread priority.

Thread priority simply means that threads with the highest priority will get a chance for execution prior to low-priority threads. One can specify the priority but it’s not necessary that the highest priority thread will get executed before the lower-priority thread. Thread scheduler assigns processor to thread on the basis of thread priority. The range of priority changes between 1–10 from lowest priority to highest priority.

# 47. What do you mean by the ThreadLocal variable in Java?

ThreadLocal variables are special kinds of variables created and provided by the Java ThreadLocal class. These variables are only allowed to be read and written by the same thread. Two threads cannot be able to see each other’s ThreadLocal variable, so even if they will execute the same code, then there won’t be any race condition and the code will be thread-safe.

# 48. What is semaphore?

Semaphore is regarded as a thread synchronization construct that is usually required to control and manage the access to the shared resource using counters. It simply sets the limit of the thread. The semaphore class is defined within the package java.util.concurrent and can be used to send signals between threads to avoid missed signals or to guard critical sections. It can also be used to implement resource pools or bounded collection.

# 49. Explain Thread Group. Why should we not use it?

ThreadGroup is a class that is used to create multiple groups of threads in a single object. This group of threads is present in the form of three structures in which every thread group has a parent except the initial thread. Thread groups can contain other thread groups also. A thread is only allowed to have access to information about its own thread group, not other thread groups.

# 50. What will happen if we don’t override the thread class run() method?

Nothing will happen as such if we don’t override the run() method. The compiler will not show any error. It will execute the run() method of thread class and we will just don’t get any output because the run() method is with an empty implementation.

Conclusion:

So, here we have covered the mostly asked Java Interview Questions from basic to advanced for all interested candidates. For a complete understanding of Java refer to our[Java Certification Program.](https://www.scholarhat.com/course/java-programming-course)

# FAQs

# Q1. Is multithreading important in Java for an interview?

# Q2. Can we call run() method of a thread class?

# Q3. Why multithreading is faster in Java?

# Q1. What is Deadlock in Java Multithreading?

In Java, a **deadlock** occurs when two or more threads are blocked forever, waiting for each other to release resources, such as locks, that they need to continue execution. This results in a situation where no threads can proceed, leading to a system freeze or hanging of the program.

A **deadlock** happens when the following four conditions hold simultaneously, often referred to as the **“deadlock conditions”**:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode, i.e., only one thread can use a resource at any given time.
2. **Hold and Wait**: A thread holding a resource is waiting for another resource held by another thread.
3. **No Preemption**: Resources cannot be forcibly taken away from threads; they can only be released voluntarily by the thread holding them.
4. **Circular Wait**: A set of threads waits for each other in a circular chain. Each thread holds one resource and is waiting for the next resource held by the next thread in the chain.

## Example of Deadlock

Here is an example where two threads, T1 and T2, leading to a deadlock.

public class DeadLockExample {  
  
 public static void main(String[] args) {  
  
 // Define two resources to create potential deadlock conditions  
 Object resource1 = new Object();  
 Object resource2 = new Object();  
  
 // Thread 1 tries to acquire lock on resource1 first, then on resource2  
 Thread t1 = new Thread(() -> {  
 synchronized (resource1) { // Acquire lock on resource1  
 System.out.println("Thread 1 acquired lock for resource-1");  
 try {  
 System.out.println("Thread 1 sleeping for 3 sec");  
 Thread.sleep(3000); // Simulate some work by sleeping  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.out.println("Thread 1 woke up");  
  
 synchronized (resource2) { // Attempt to acquire lock on resource2  
 System.out.println("Thread 1 trying to acquire the lock for resource 2");  
 }  
 }  
 });  
  
 // Thread 2 tries to acquire lock on resource2 first, then on resource1  
 Thread t2 = new Thread(() -> {  
 synchronized (resource2) { // Acquire lock on resource2  
 System.out.println("Thread 2 acquired lock for resource-2");  
 try {  
 System.out.println("Thread 2 sleeping for 3 sec");  
 Thread.sleep(3000); // Simulate some work by sleeping  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.out.println("Thread 2 woke up");  
  
 synchronized (resource1) { // Attempt to acquire lock on resource1  
 System.out.println("Thread 2 trying to acquire the lock for resource 1");  
 }  
 }  
 });  
  
 // Start both threads  
 t1.start();  
 t2.start();  
 }  
}

**Explanation**

**Resource Definition**:

* Two resources (resource1 and resource2) are defined as shared objects between threads. These represent shared locks.

**Thread 1**:

* Acquires a lock on resource1 first.
* Simulates work by sleeping for 3 seconds while holding the lock on resource1.
* After waking up, attempts to acquire a lock on resource2.

**Thread 2**:

* Acquires a lock on resource2 first.
* Simulates work by sleeping for 3 seconds while holding the lock on resource2.
* After waking up, attempts to acquire a lock on resource1.

**Deadlock**:

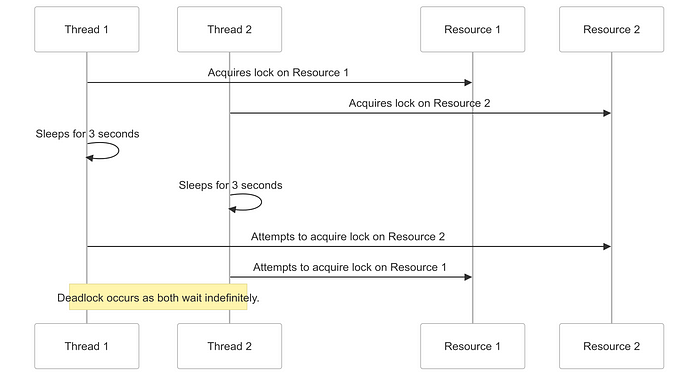
* Both threads hold one lock and attempt to acquire the other. Since neither releases their initial lock, a deadlock occurs.

**Solution (commented in the code)**:

* Ensure both threads acquire locks in the same order (e.g., resource1 first, then resource2) to prevent deadlock.

**Deadlock Diagram**

Here’s a simplified diagram of the deadlock scenario:



**How to Avoid Deadlock**

To avoid deadlocks, consider the following strategies:

1. **Lock Ordering**: Always acquire locks in a fixed order, so that cyclic dependencies are avoided.
2. **Timeouts**: Implement timeouts for acquiring locks, so that if a thread fails to acquire a lock within a specified time, it releases the locks it already holds and tries again.
3. **Deadlock Detection**: Use mechanisms that detect when a deadlock occurs and resolve it.
4. **Avoid Nested Locks**: Reduce the use of nested locks and try to avoid holding more than one lock at a time.

# Q2. Explain Class Lock and Object Lock in Java Multithreading.

**Overview and Purpose**

In Java multithreading, **locks** are mechanisms used to manage thread synchronization, ensuring that multiple threads do not access critical sections of code simultaneously. There are two primary types of locks:

**Object Lock:**

* When a thread enters a synchronized method or block on an instance, it acquires an **object-level lock**.
* Other threads trying to access synchronized methods/blocks of the same object must wait for the lock to be released.
* However, other threads can access synchronized methods/blocks of **different objects** of the same class.

**Class Lock:**

* When a thread enters a static synchronized method or a synchronized block on the class, it acquires the **class-level lock**.
* This lock is tied to the class Class object (like ClassName.class).
* No thread can execute any other static synchronized method/block of that class until the lock is released.

// Class Lock Example  
class ClassLockExample {  
 // Synchronized static method - locks the Class object  
 public static synchronized void staticMethod() {  
 System.out.println(Thread.currentThread().getName() + " has acquired the Class lock");  
 try {  
 Thread.sleep(3000); // Simulate some work  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println(Thread.currentThread().getName() + " is releasing the Class lock");  
 }  
}  
  
// Object Lock Example  
class ObjectLockExample {  
 // Synchronized instance method - locks the instance (Object lock)  
 public synchronized void instanceMethod() {  
 System.out.println(Thread.currentThread().getName() + " has acquired the Object lock");  
 try {  
 Thread.sleep(3000); // Simulate some work  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println(Thread.currentThread().getName() + " is releasing the Object lock");  
 }  
}  
  
public class LockExamples {  
 public static void main(String[] args) {  
 // Demonstrating Class Lock  
 Thread classLockThread1 = new Thread(() -> ClassLockExample.staticMethod(), "Thread-1");  
 Thread classLockThread2 = new Thread(() -> ClassLockExample.staticMethod(), "Thread-2");  
  
 // Demonstrating Object Lock  
 ObjectLockExample obj = new ObjectLockExample();  
 Thread objectLockThread1 = new Thread(() -> obj.instanceMethod(), "Thread-3");  
 Thread objectLockThread2 = new Thread(() -> obj.instanceMethod(), "Thread-4");  
  
 // Start threads  
 classLockThread1.start();  
 classLockThread2.start();  
 objectLockThread1.start();  
 objectLockThread2.start();  
 }  
}  
// Output  
/\*   
Thread-3 has acquired the Object lock  
Thread-1 has acquired the Class lock  
Thread-3 is releasing the Object lock  
Thread-1 is releasing the Class lock  
Thread-4 has acquired the Object lock  
Thread-2 has acquired the Class lock  
Thread-4 is releasing the Object lock  
Thread-2 is releasing the Class lock  
\*/

**Explanation:**

**Class Lock**

* A **class lock** is applied when a static synchronized method is used. The lock is applied on the Class object, meaning all threads accessing any static synchronized method of that class will block until the lock is released.
* In the example:
* ClassLockExample.staticMethod() acquires the lock on the Class object (ClassLockExample.class).
* Threads accessing this method will execute sequentially, ensuring thread safety for shared resources associated with the class.

**Object Lock**

* An **object lock** is applied when a synchronized instance method is used. The lock is specific to the instance of the class. Different instances can have their own locks, allowing threads to execute methods on different objects simultaneously.
* In the example:
* obj.instanceMethod() acquires the lock on the obj instance.
* Only one thread can execute instanceMethod on the same object at a time.

**Output Scenario**

**Class Lock**:

* Thread-1 acquires the class lock, prints the message, and holds it for 3 seconds.
* Thread-2 waits until Thread-1 releases the lock before proceeding.

**Object Lock**:

* Thread 3 acquires the object lock obj and holds it for 3 seconds.
* Thread-4 waits until Thread-3 releases the lock before executing.

# Q3. Write a Java program that uses two threads to print numbers alternately. ( 1 to 10 )

public class Main{  
  
 private static final int MAX\_NUMBER = 10; // Maximum number to be printed  
 private int currentNumber = 1; // Current number being printed  
 private int threadTurn = 1; // Indicates which thread's turn it is (1 for thread-1, 2 for thread-2)  
  
 public static void main(String[] args) {  
 Main object = new Main();  
  
 // Thread to print numbers for thread ID 1  
 Thread t1 = new Thread(() -> object.printNumber(1));  
   
 // Thread to print numbers for thread ID 2  
 Thread t2 = new Thread(() -> object.printNumber(2));  
  
 // Start both threads  
 t1.start();  
 t2.start();  
 }  
  
 private synchronized void printNumber(int threadId) {  
 // Loop until the current number reaches the maximum limit  
 while (currentNumber < MAX\_NUMBER) {  
 // Wait if it's not the current thread's turn  
 while (threadTurn != threadId) {  
 try {  
 wait(); // Release the lock and wait until notified  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 // Print the current number with the thread ID  
 System.out.println("Thread " + threadId + " - " + currentNumber);  
  
 // Increment the current number  
 currentNumber += 1;  
  
 // Change the turn to the next thread  
 threadTurn = threadTurn % 2 + 1;  
  
 // Notify all waiting threads to check their turn  
 notifyAll();  
 }  
 }  
}  
// Output :  
/\*  
Thread 1 - 1  
Thread 2 - 2  
Thread 1 - 3  
Thread 2 - 4  
Thread 1 - 5  
Thread 2 - 6  
Thread 1 - 7  
Thread 2 - 8  
Thread 1 - 9  
Thread 2 - 10  
\*/

**Explanation**

**Shared Variables**:

* MAX\_NUMBER: Defines the upper limit for numbers to be printed.
* currentNumber: Tracks the current number being printed.
* threadTurn: Ensures synchronization between threads, deciding which thread prints the number.

**Thread Initialization**:

* t1 and t2 represent the two threads responsible for printing numbers. They are configured to call the synchronized printNumber method.

**Synchronized Method**:

* **Purpose**: Prevents concurrent access to shared variables (currentNumber and threadTurn).
* **Logic**:
* If it’s not the thread’s turn (threadTurn != threadId), the thread calls wait() and releases the lock.
* When notified, the thread resumes execution and checks if it’s its turn.
* The thread prints the number, increments currentNumber, and switches the turn to the other thread (threadTurn = threadTurn % 2 + 1).
* Calls notifyAll() to wake up other threads waiting for their turn.

**Flow**: Threads alternate in printing numbers until currentNumber reaches MAX\_NUMBER.

# Q4. What is ThreadLocal in Java?

* **Definition**: A ThreadLocal is a Java class that provides thread-local variables, ensuring each thread has its own isolated instance of the variable.
* **Purpose**: It eliminates the need for synchronization when threads need their own dedicated copies of variables. This is particularly useful in scenarios where thread-specific data is required, such as user session tracking or transaction management.
* **Thread Safety**: It allows sharing variables among methods within the same thread without being exposed to other threads, ensuring thread safety without synchronization.
* **Performance**: Reduces overhead associated with synchronization mechanisms like locks.

## How ThreadLocal Works

1. Each thread maintains its own copy of the variable in a **map-like structure** within the Thread object.
2. When you call get() on a ThreadLocal variable, the current thread fetches its own copy of the variable.
3. When you call set(), it sets the value only for the current thread.

**Internal Working**: Each thread has a ThreadLocalMap object inside it, which stores the ThreadLocal instances as keys and their respective values.

## ThreadLocal Class Methods

The ThreadLocal class provides four key methods:

**void set(T value)**Sets the value of the ThreadLocal variable for the current thread.

ThreadLocal<String> threadLocal = new ThreadLocal<>();   
threadLocal.set("Thread1 Value");

**T get()**Returns the value of the ThreadLocal variable for the current thread. If no value is set, it returns null unless overridden by initialValue().

String value = threadLocal.get();   
System.out.println("ThreadLocal Value: " + value);

**void remove()**Removes the current thread’s value for the ThreadLocal variable to free up memory and avoid potential memory leaks.

threadLocal.remove();   
System.out.println("Value after remove: " + threadLocal.get());

**Code Example:**

public class ThreadLocalExample {  
 private static ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);  
  
 public static void main(String[] args) {  
 Runnable task = () -> {  
 for (int i = 0; i < 5; i++) {  
 threadLocal.set(threadLocal.get() + 1);  
 System.out.println(Thread.currentThread().getName() + " - " + threadLocal.get());  
 }  
 };  
  
 Thread t1 = new Thread(task, "Thread-1");  
 Thread t2 = new Thread(task, "Thread-2");  
  
 t1.start();  
 t2.start();  
 }  
}  
// Output:  
/\*  
Thread-1 - 1  
Thread-1 - 2  
Thread-2 - 1  
Thread-1 - 3  
Thread-2 - 2  
Thread-1 - 4  
Thread-2 - 3  
Thread-1 - 5  
Thread-2 - 4  
Thread-2 - 5  
\*/

## Explanation:

* Each thread increments its own version of the ThreadLocal variable without interfering with the other.

## ****Memory Leak Concerns****

If you do not call remove() after the thread has finished its work, the values stored in the ThreadLocal map may cause memory leaks.

**Best Practice:** Always call remove() when the ThreadLocal variable is no longer needed.

try {  
 threadLocal.set("value");  
 // Perform operations  
} finally {  
 threadLocal.remove();  
}

# Q5. What is the Producer-Consumer Problem in Java? Explain with Example.

The **Producer-Consumer Problem** is a classic synchronization problem in operating systems and concurrent programming. It involves two types of processes (or threads) that share a common buffer:

1. **Producer**: Produces items (data) and adds them to the buffer.
2. **Consumer**: Consumes (removes) items from the buffer.

import static java.lang.Thread.sleep;  
import java.util.LinkedList;  
import java.util.Queue;  
  
// Shared resource class that acts as a buffer for producer and consumer  
class SharedResource {  
  
 private Queue<Integer> queue; // Queue to store produced items  
 private int size; // Maximum size of the queue  
  
 public SharedResource(int size) {  
 this.queue = new LinkedList<>(); // Initialize the queue  
 this.size = size; // Set the size limit for the queue  
 }  
  
 // Producer method to add an item into the queue  
 public synchronized void produce(int value) {  
  
 while (queue.size() == size) { // If the queue is full, wait until space is available  
 System.out.println("Queue is full, " + Thread.currentThread().getName() + " is waiting...");  
 try {  
 wait(); // Release the lock and wait until notified  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 // Add the value to the queue and notify all waiting threads  
 queue.add(value);  
 System.out.println(value + " Item Added in Queue by " + Thread.currentThread().getName());  
 notifyAll(); // Notify the consumer thread that an item is available  
 }  
  
 // Consumer method to remove an item from the queue  
 public synchronized void consume() {  
 while (queue.isEmpty()) { // If the queue is empty, wait until an item is produced  
 System.out.println("Queue is empty, " + Thread.currentThread().getName() + " is waiting...");  
 try {  
 wait(); // Release the lock and wait until notified  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 // Remove the value from the queue and notify all waiting threads  
 Integer consumedValue = queue.remove();  
 System.out.println(consumedValue + " Item Consumed by " + Thread.currentThread().getName());  
 notifyAll(); // Notify the producer thread that space is available  
 }  
}  
  
// Main class demonstrating the Producer-Consumer Problem  
public class ProducerAndConsumerProblem {  
  
 public static void main(String[] args) {  
 // Shared resource with a queue size of 3  
 SharedResource sharedResource = new SharedResource(3);  
  
 // Producer thread  
 Thread producer = new Thread(() -> {  
 for (int i = 0; i < 5; i++) { // Produce 5 items  
 try {  
 sleep(1000); // Simulate production delay  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 sharedResource.produce(i); // Call the produce method  
 }  
 }, "THREAD-PRODUCER");  
  
 // Consumer thread  
 Thread consumer = new Thread(() -> {  
 for (int i = 0; i < 5; i++) { // Consume 5 items  
 try {  
 sleep(1000); // Simulate consumption delay  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 sharedResource.consume(); // Call the consume method  
 }  
 }, "THREAD-CONSUMER");  
  
 // Start both threads  
 producer.start();  
 consumer.start();  
 }  
}  
  
// Output  
/\*  
Queue is empty, THREAD-CONSUMER is waiting...  
0 Item Added in Queue by THREAD-PRODUCER  
0 Item Consumed by THREAD-CONSUMER  
1 Item Added in Queue by THREAD-PRODUCER  
1 Item Consumed by THREAD-CONSUMER  
2 Item Added in Queue by THREAD-PRODUCER  
2 Item Consumed by THREAD-CONSUMER  
3 Item Added in Queue by THREAD-PRODUCER  
3 Item Consumed by THREAD-CONSUMER  
4 Item Added in Queue by THREAD-PRODUCER  
4 Item Consumed by THREAD-CONSUMER  
\*/

## ****Explanation****

**SharedResource Class**

* Acts as the shared buffer (queue) between the producer and consumer threads.
* It uses synchronized methods to ensure thread safety.
* produce() method: Adds items to the queue and waits if the queue is full.
* consume() method: Removes items from the queue and waits if the queue is empty.
* Uses wait() and notifyAll() for inter-thread communication.

**Producer Thread**

* Simulates producing items and adding them to the shared resource (queue).
* Produces a total of 5 items with a 1-second delay between each production.

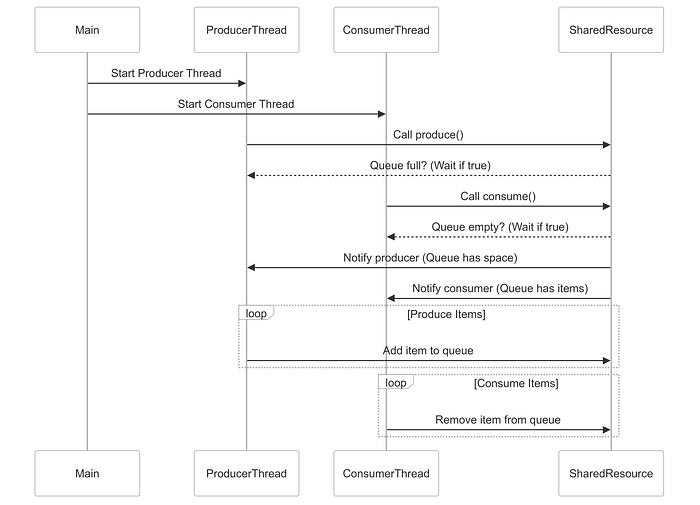
**Consumer Thread**

* Simulates consuming items from the shared resource (queue).
* Consumes a total of 5 items with a 1-second delay between each consumption.

**Main Thread**

* Creates and starts both the producer and consumer threads.

## Sequence Diagram



<https://medium.com/@vikas.taank_40391/5-essentials-java-multithreading-questions-q3-e3006f05c13b>

<https://medium.com/@vikas.taank_40391/5-essentials-java-multithreading-questions-q3-e3006f05c13b>

<https://blog.stackademic.com/java-executor-service-best-practices-539eb73e5351>

<https://saannjaay.medium.com/mastering-java-multithreading-concurrency-future-objects-no-question-beyond-239dfd410460>

<https://saannjaay.medium.com/mastering-java-8-streams-filter-and-map-use-cases-with-coding-examples-879c02320e65>

<https://dip-mazumder.medium.com/java-concurrency-101-understanding-multithreading-fundamentals-e5ed48b04ca5>

<https://medium.com/@basecs101/deadlock-in-java-multi-threading-latest-88078753fe1f>

https://medium.com/javarevisited/a-complete-guide-on-executorservice-in-java-67528f1a535b