Mutithreading:

Semaphores in java ?

**Semaphores in Java**

A **Semaphore** in Java is a concurrency control mechanism used to restrict access to a shared resource by multiple threads. It is part of the **java.util.concurrent** package and is used to limit the number of threads that can access a resource at the same time.

**How Semaphores Work?**

* **Counting Semaphore** maintains a counter:
  + If the counter > 0 → A thread **acquires** a permit and proceeds.
  + If the counter = 0 → Threads must **wait** until a permit is available.
* **Binary Semaphore** (special case, similar to a mutex) allows only **one thread** at a time.

**Example Usage of Semaphore**

import java.util.concurrent.Semaphore;

class SharedResource {

private final Semaphore semaphore = new Semaphore(2); // Allow max 2 threads

void useResource(String threadName) {

try {

semaphore.acquire(); // Acquire a permit

System.out.println(threadName + " is accessing the resource...");

Thread.sleep(2000); // Simulate resource usage

System.out.println(threadName + " is done.");

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

semaphore.release(); // Release the permit

}

}

}

public class SemaphoreExample {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

for (int i = 1; i <= 5; i++) {

final String threadName = "Thread-" + i;

new Thread(() -> resource.useResource(threadName)).start();

}

}

}

**Key Points**

1. **Semaphore Initialization:** new Semaphore(permits) defines the number of threads that can access a resource simultaneously.
2. **Acquiring Permits:** semaphore.acquire() decreases the permit count.
3. **Releasing Permits:** semaphore.release() increases the permit count.
4. **Blocking Behavior:** If no permit is available, threads wait until one is released.
5. **Fairness Mode:** new Semaphore(permits, true) ensures fair access (FIFO).

**Use Cases**

* **Database connection pools**
* **Rate limiting**
* **Controlling concurrent file access**

Semaphores help **synchronize threads** efficiently in multi-threaded applications! 🚀

Interview questions on multithreading?

A,

Basic Multithreading Interview Questions

1. What is multithreading? Why is it used?

• process of executing one or more parts of a single program is called multithreading.

2. What is a thread?

• A thread is the smallest unit of a process that can be scheduled for execution. Threads in a single process share memory and resources, but each has its own execution stack.

3. What is the difference between a process and a thread?

• A process is a self-contained execution environment with its own memory space, while a thread is a lightweight unit within a process that shares its memory and resources with other threads in the same process.

4. How can you create a thread in Java?

• By extending the Thread class and overriding its run() method.

• By implementing the Runnable interface and defining the run() method.

5. What is the difference between start() and run() methods in Java?

• The start() method creates a new thread and invokes the run() method in that thread.

• The run() method, if called directly, is executed in the current thread as a normal method call.

6. What are the benefits of multithreading?

• Better CPU utilization.

• Improved application responsiveness.

• Simplifies modeling for certain tasks, like handling multiple connections in a server.

7. What are the lifecycle states of a thread in Java?

• New: Thread is created but not started.

• Runnable: Thread is ready to run but waiting for CPU time.

• Running: Thread is executing.

• Blocked/Waiting: Thread is waiting for resources or signals.

• Terminated: Thread has completed execution.

8. What is a daemon thread?

• A daemon thread is a low-priority thread that runs in the background and terminates when all user threads are completed (e.g., garbage collector).

Intermediate Multithreading Interview Questions

1. What is thread synchronization, and why is it needed?

• Thread synchronization is a mechanism to control access to shared resources by multiple threads to prevent data inconsistencies and race conditions.

2. What is a race condition?

• A race condition occurs when two or more threads try to access shared resources simultaneously, leading to unpredictable or incorrect behavior.

3. What is the difference between synchronized methods and synchronized blocks?

• A synchronized method locks the entire object, blocking all other threads from accessing any synchronized method on that object.

• A synchronized block allows more fine-grained control by locking only a specific section of code.

4. What are wait(), notify(), and notifyAll() in Java?

• These are Object class methods used for inter-thread communication.

• wait(): Makes the current thread wait until another thread invokes notify() or notifyAll().

• notify(): Wakes up a single thread waiting on the object’s monitor.

• notifyAll(): Wakes up all threads waiting on the object’s monitor.

5. What is a thread pool?

• A thread pool is a collection of pre-created threads used for executing tasks. It improves application performance by reusing threads and limiting the overhead of thread creation.

6. What is deadlock in multithreading?

• Deadlock occurs when two or more threads are waiting indefinitely for resources locked by each other, preventing further execution.

7. What are some ways to avoid deadlocks?

• Avoid nested locks.

• Use a timeout for locking.

• Acquire locks in a consistent order.

• Minimize the use of locks.

8. What is the difference between volatile and synchronized in Java?

• volatile: Ensures visibility of changes to a variable across threads but does not guarantee atomicity.

• synchronized: Provides both atomicity and visibility by locking access to code blocks or methods.

9. What is the difference between concurrency and parallelism?

• Concurrency refers to executing multiple tasks in overlapping time periods (not necessarily simultaneously).

• Parallelism refers to executing multiple tasks simultaneously on different cores or processors.

10. What are the differences between Thread and Executor framework?

• Thread: Requires manual creation and management of threads.

• Executor: Provides a high-level API to manage threads using thread pools.

Advanced Multithreading Interview Questions

1. What is the ReentrantLock class in Java? How is it different from synchronized?

• ReentrantLock is a more flexible locking mechanism compared to synchronized. It provides features like:

• Try to acquire lock with tryLock().

• Ability to interrupt a thread waiting for a lock.

• Fair locking to ensure threads acquire locks in the order they requested.

2. What are Callable and Future in Java? How are they different from Runnable?

• Callable: An interface that allows tasks to return results and throw exceptions.

• Future: Represents the result of an asynchronous computation.

• Runnable: Does not return a result and cannot throw checked exceptions.

3. What is the Fork/Join framework in Java?

• A framework designed for parallel task execution, particularly suited for dividing a task into smaller subtasks recursively and merging their results.

4. What is ThreadLocal?

• ThreadLocal provides thread-local variables, ensuring that each thread has its own independent copy of the variable.

5. What are the differences between CyclicBarrier and CountDownLatch?

• CyclicBarrier: Allows a group of threads to wait for each other at a common barrier point and can be reused.

• CountDownLatch: A one-time latch that allows one or more threads to wait until a specific set of tasks is completed.

6. What is the ConcurrentHashMap? How is it different from HashMap?

• ConcurrentHashMap allows concurrent access to its segments, making it thread-safe without locking the entire map.

• HashMap is not thread-safe and must be synchronized externally for concurrent access.

7. What is a BlockingQueue?

• A BlockingQueue is a thread-safe queue designed for producer-consumer scenarios. It blocks threads when:

• Trying to add elements to a full queue.

• Trying to remove elements from an empty queue.

8. What are atomic classes in Java?

• Atomic classes (e.g., AtomicInteger, AtomicLong) provide lock-free thread-safe operations on single variables.

9. Explain the Compare-And-Swap (CAS) mechanism.

• CAS is a low-level concurrency technique used to update a variable atomically. It checks if a variable has a specific value and, if so, updates it to a new value.

10. What is the difference between synchronized collections and Concurrent collections?

• Synchronized collections: Use synchronized blocks or methods, blocking threads for both read and write operations.

• Concurrent collections: Provide fine-grained locks or lock-free mechanisms for better concurrency (e.g., ConcurrentHashMap, CopyOnWriteArrayList).

11. What is the purpose of the ExecutorService?

• It provides methods for managing and controlling thread lifecycle, including submitting tasks, shutting down executors, and scheduling periodic tasks.

12. What is work-stealing in the Fork/Join framework?

• Work-stealing allows idle threads to “steal” tasks from busy threads to improve resource utilization.

13. Explain how the Phaser works in Java.

• Phaser is a flexible synchronization barrier that allows threads to dynamically register and deregister. It’s useful in scenarios where the number of parties (threads) changes over time.

14. What are the limitations of multithreading?

• Complex debugging and testing.

• Risk of deadlocks and race conditions.

• Overhead due to context switching.

• Increased memory consumption due to thread stacks.

By preparing for these questions, you can cover the full spectrum of multithreading concepts from basic to advanced levels.

1,What is process?

A, any program in running state is called process.

2,What is relation between process and thread ?

A,process uses thread to increase accuracy and performance of process.

Process divided into multiple parts called thread.

3,what is **Process-based multitasking**:

A,

* Executes several tasks simultaneously, where each task is a separate, independent process.
* Example: Running a Java IDE while using a text editor or web browser.
* Focuses on the "big picture."

4, **what is Thread-based multitasking**:?

* Executes several tasks simultaneously, where each task is a separate, independent part (thread) of the same program.
* Example: JUnit uses threads to run test cases in parallel, or threads in a game manage different objects like cars and motorbikes.
* Handles the details within the same program.

5,explain how to run two functions at same time using thread?

Ans,

, class Hi extends Thread {

    public void run() {

        for (int i = 1; i <= 5; i++) {

            System.out.println("Hi");

            try {

                Thread.sleep(1000); // Pause for 1 second

            } catch (Exception e) {

                System.out.println(e);

            }

        }

    }

}

class Hello extends Thread {

    public void run() {

        for (int i = 1; i <= 5; i++) {

            System.out.println("Hello");

            try {

                Thread.sleep(1000); // Pause for 1 second

            } catch (Exception e) {

                System.out.println(e);

            }

        }

    }

}

public class ThreadDemo {

    public static void main(String[] args) {

        Hi obj1 = new Hi();

        Hello obj2 = new Hello();

        obj1.start(); // Start the "Hi" thread

        obj2.start(); // Start the "Hello" thread

    }

}

Sleep is used to slow down the output printing.

**Using only lambda expressions:**

public class ThreadDemo {

public static void main(String[] args) {

Thread t1 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi");

try {

Thread.sleep(1000); // Pause for 1 second

} catch (Exception e) {

System.out.println(e);

}

}

});

Thread t2 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000); // Pause for 1 second

} catch (Exception e) {

System.out.println(e);

}

}

});

t1.start(); // Start the "Hi" thread

t2.start(); // Start the "Hello" thread

}

}

6,use runnable interface to call a function instead of using extend thread class?

5,which is better process based or thread based?

A,

Thread communication is inexpensive and context switching from one thread to another is lower in cost.

So thread based is better.

6,what is thread?

A,thread is light weight rocess in with in process.

5.1,write program using implement runnable?

Program

Answer:

class Hi implements Runnable {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi");

try {

Thread.sleep(1000);

} catch (Exception e) {}

}

}

}

class Hello implements Runnable {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000);

} catch (Exception e) {}

}

}

}

public class ThreadDemo {

public static void main(String[] args) {

Runnable obj1 = new Hi();

Runnable obj2 = new Hello();

Thread t1 = new Thread(obj1);

Thread t2 = new Thread(obj2);

t1.start();

try {

Thread.sleep(10);

} catch (Exception e) {}

t2.start();

}

}

5.3,  
write runnable implementation using lamda expressions?  
a,  
public class Main {

public static void main(String[] args) {

Runnable obj1 = () -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi");

try {

Thread.sleep(1000);

} catch (Exception e) {}

}

};

Runnable obj2 = () -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000);

} catch (Exception e) {}

}

};

Thread t1 = new Thread(obj1);

Thread t2 = new Thread(obj2);

t1.start();

t2.start();

}

}

7,What is the Runnable interface in Java, and how is it used?

Answer:

• Runnable is a functional interface in Java that represents a task to be executed by a thread. It contains a single method, run(), which must be implemented when the interface is used.

8, What is the difference between extending Thread and implementing Runnable?

• Answer:

• Extending Thread:

• The class inherits all methods of the Thread class.

• It doesn’t allow extending another class as Java does not support multiple inheritance.

Implementing Runnable:

• The class focuses solely on the task to be executed (run() method) without inheriting unnecessary methods.

• Allows the class to extend another class if needed.

9, What is the purpose of Thread.sleep() in the code?

• Answer:

• Thread.sleep(milliseconds) pauses the execution of the current thread for the specified time.

Thread.sleep(1000);

10,

Can you explain thread safety in the context of this example?

• Answer:

• In this example, there are no shared resources between Hi and Hello, so thread safety is not a concern.

• However, if threads share resources (e.g., variables), synchronization mechanisms like synchronized blocks or methods would be needed to ensure thread safety.

11,

What is the difference between run() and start() methods in the Thread class?

• Answer:

• run(): Contains the task to be executed by the thread. Calling run() directly does not start a new thread; it executes in the current thread.

• start(): Creates a new thread and invokes the run() method on that thread.

• In the code, t1.start() and t2.start() are used to properly start the threads.

12,

What are some common issues in multi-threading, and how can they be avoided?

• Answer:

• Race Conditions: When multiple threads access shared resources simultaneously. Use synchronization to avoid this.

• Deadlocks: When two threads are waiting for each other to release resources. Avoid circular dependencies.

• Thread Interference: When threads modify shared data inconsistently. Ensure thread-safe access to shared data.

• Example: This code does not involve shared data, so it avoids such issues.

13,how to use lambda expression in runnable interface multi threading?

Ans,

blic class ThreadDemo {

public static void main(String[] args) {

Runnable obj1 = () -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi");

try {

Thread.sleep(1000);

} catch (Exception e) {

}

}

};

Runnable obj2 = () -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000);

} catch (Exception e) {

}

}

};

Thread t1 = new Thread(obj1);

Thread t2 = new Thread(obj2);

t1.start();

try {

Thread.sleep(10);

} catch (Exception e) {

}

t2.start();

}

}

13,.1What does join do?

A,if u have t1 t2 and bye statement.

If we use t1.join and t2.join bye will execute last.

13,explain working of t1.join and t2.join along using isalive() using code?

A,

public class ThreadDemo {

public static void main(String[] args) {

Thread t1 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi");

try {

Thread.sleep(1000);

} catch (Exception e) {

System.out.println(e);

}

}

});

Thread t2 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000);

} catch (Exception e) {

System.out.println(e);

}

}

});

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (Exception e) {

}

System.out.println(t1.isAlive()); // Check if t1 is alive

System.out.println("Bye");

}

}

14,e xplain the role of the join() method in multithreading. Why is it used in the given code?

• The join() method ensures that the current thread waits for the specified thread to finish before proceeding.

• In the example, it ensures that t1 and t2 complete execution before checking t1.isAlive() and printing “Bye

15,three lifecycle ?

A,

What are the different states of a thread in Java? Can you explain each state with examples?

• New: Thread created but not started (Thread t = new Thread();).

• Runnable: Thread ready to run, waiting for CPU time.

• Running: Thread is executing.

• Blocked/Waiting: Thread waiting for a resource or another thread.

• Terminated: Thread has completed executio

16, What does the isAlive() method do in a thread? How is it used in the example code?

• isAlive() checks if the thread is still running or in a runnable state.

• In the example, it verifies if t1 is active after join() ensures its completion, returning false.

17,how min and max priorities are managed in multithreading ?

A,

public class ThreadDemo {

public static void main(String[] args) throws Exception {

Thread t1 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hi " + Thread.currentThread().getPriority());

try {

Thread.sleep(1000);

} catch (Exception e) {

}

}

});

Thread t2 = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

System.out.println("Hello");

try {

Thread.sleep(1000);

} catch (Exception e) {

}

}

});

t1.setPriority(Thread.MIN\_PRIORITY); // Setting t1 priority to minimum

t2.setPriority(Thread.MAX\_PRIORITY); // Setting t2 priority to maximum

System.out.println(t1.getPriority()); // Prints t1's priority

System.out.println(t2.getPriority()); // Prints t2's priority

t1.start();

try {

Thread.sleep(10);

} catch (Exception e) {

}

t2.start();

t1.join();

t2.join();

System.out.println(t1.isAlive());

System.out.println("Bye");

}

}

18,how synchronised method works in multithreading?

A,

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class SyncDemo {

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

Thread t1 = new Thread(() -> {

for (int i = 1; i <= 1000; i++) {

counter.increment();

}

});

Thread t2 = new Thread(() -> {

for (int i = 1; i <= 1000; i++) {

counter.increment();

}

});

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println("Count: " + counter.getCount());

}

}

Interview questions by backend developer?

A,  
**Q1. What are issues without volatile keyword in Java?**

Without the volatile keyword, changes made to a variable by one thread may not be visible to other threads immediately. This can lead to issues such as:

* **Visibility Problems**: Threads may see stale values of the variable.
* **Reordering**: The compiler or processor may reorder instructions, leading to unexpected behavior.

**Q2. What is volatile keyword in Java?**

The volatile keyword in Java is used to mark a variable as being stored in main memory. This ensures that:

* **Visibility**: Changes to the variable are immediately visible to all threads.
* **No Caching**: The variable is not cached in thread-local memory.

**Q3. Why we need multiple threads?**

* **Concurrency**: To perform multiple tasks simultaneously.
* **Responsiveness**: To keep applications responsive (e.g., UI applications).
* **Resource Utilization**: To utilize multiple CPU cores effectively.
* **Separation of Concerns**: To separate different tasks into different threads.

**Q4. What are different ways to create Threads in Java?**

* **Extending Thread Class**: Create a new class that extends Thread and override the run method.
* **Implementing Runnable Interface**: Create a class that implements Runnable and pass an instance of it to a Thread object.
* **Using Executor Framework**: Use the ExecutorService to manage and execute threads.

**Q5. List all thread methods in Java?**

* **start()**: Starts the thread.
* **run()**: Contains the code to be executed by the thread.
* **sleep(long millis)**: Causes the thread to sleep for the specified time.
* **join()**: Waits for the thread to die.
* **interrupt()**: Interrupts the thread.
* **isAlive()**: Checks if the thread is alive.
* **getName()**: Returns the name of the thread.
* **setName(String name)**: Sets the name of the thread.
* **getPriority()**: Returns the priority of the thread.
* **setPriority(int priority)**: Sets the priority of the thread.

**Q6. What is ThreadLocal class in Java?**

ThreadLocal is a class that provides thread-local variables. Each thread accessing such a variable has its own, independently initialized copy of the variable.

**Q7. Explain differences between synchronized keyword, Lock, and Semaphore for managing thread synchronization?**

* **synchronized keyword**: Provides a simple way to lock a method or block of code. It is intrinsic and cannot be interrupted.
* **Lock**: Provides more flexible and sophisticated locking mechanisms. It can be interrupted and supports multiple conditions.
* **Semaphore**: Controls access to a resource by multiple threads. It allows a specified number of threads to access a resource concurrently.

**Q8. What is Livelock in multithreading?**

Livelock occurs when two or more threads continuously change their state in response to each other without making any progress. It is similar to deadlock but the threads are not blocked.

**Q9. What is Deadlock in multithreading?**

Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. This happens when there is a circular dependency on resources.

**Q10. What happens if run() method is called without start() in Java threads?**

If the run() method is called directly, it will be executed in the current thread, not in a new thread. The start() method is needed to create a new thread and execute the run() method in that new thread.

**Q11. What is the use of a Thread Pool in Java?**

A thread pool manages a pool of worker threads. It allows for reusing threads for multiple tasks, reducing the overhead of creating and destroying threads. It also helps in managing the number of concurrent threads.

**Q12. What is IllegalMonitorStateException?**

IllegalMonitorStateException is thrown when a thread tries to wait, notify, or notifyAll on an object without holding the intrinsic lock on that object.

**Q13. What are atomic variables?**

Atomic variables are variables that provide atomic operations for reading and writing. They are part of the java.util.concurrent.atomic package and ensure thread-safe operations without using synchronization.

**Q14. Object level locking vs class level locking?**

* **Object Level Locking**: Locks the instance of the class. Only one thread can execute synchronized instance methods on the same object.
* **Class Level Locking**: Locks the class itself. Only one thread can execute synchronized static methods on the class.

**Q15. What is CompletableFuture, how is it different from Runnable?**

CompletableFuture is a class that represents a future result of an asynchronous computation. It provides methods to handle the result once it is available. Unlike Runnable, which does not return a result, CompletableFuture can return a result and handle exceptions.

**Q16. Example of CompletableFuture using thenApply and other methods?**

import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {

public static void main(String[] args) {

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 10)

.thenApply(result -> result \* 2)

.thenApply(result -> result + 5);

future.thenAccept(result -> System.out.println("Result: " + result));

}

}

**Q17. How would you ensure that a piece of code is executed by only one thread at a time?**

* **synchronized keyword**: Use the synchronized keyword to lock the method or block of code.
* **Lock**: Use ReentrantLock to lock the code.

**Q18. What is difference between Executor's submit() and execute() method?**

* **execute()**: Executes a Runnable task without returning a result.
* **submit()**: Executes a Runnable or Callable task and returns a Future representing the result.

**Q19. Is it safe to iterate over collection returned by Collections.synchronizedCollection() method, or should we synchronize the iterating code?**

It is not safe to iterate over the collection without additional synchronization. You should synchronize the iterating code to avoid ConcurrentModificationException.

**Q20. Executor Service vs Completable Future?**

* **ExecutorService**: Manages a pool of threads and provides methods to execute tasks.
* **CompletableFuture**: Represents a future result of an asynchronous computation and provides methods to handle the result.

**Q21. How do you join two threads in Java?**

Use the join() method to wait for a thread to finish before continuing execution.

thread1.join();

thread2.join();

**Q22. You have threads T1, T2, and T3. How will you ensure that thread T2 runs after T1 and thread T3 runs after T2?**

Use the join() method to ensure the order of execution.

thread1.start();

thread1.join();

thread2.start();

thread2.join();

thread3.start();

**Q23. Can we start a thread twice in Java?**

No, once a thread has been started, it cannot be started again. Attempting to start a thread twice will result in an IllegalThreadStateException.

**Q24. Can we run a thread twice in Java?**

No, a thread cannot be run twice. Once a thread has completed execution, it cannot be restarted.

**Q25. Why are wait(), notify(), and notifyAll() defined in the Object class and not in the Thread class?**

These methods are defined in the Object class because they are used for inter-thread communication on shared objects. Every object in Java has a monitor, and these methods operate on the monitor of the object.

**Q26. What are the different ways to achieve synchronization in Java?**

* **synchronized keyword**: Use the synchronized keyword to lock methods or blocks of code.
* **Lock**: Use ReentrantLock for more flexible locking.
* **Atomic Variables**: Use atomic variables for thread-safe operations without synchronization.

**Q27. Is ++ operation thread safe?**

No, the ++ operation is not thread-safe because it involves multiple steps: reading the value, incrementing it, and writing it back. These steps can be interleaved by multiple threads, leading to incorrect results.

**Q28. You have threads T1, T2, and T3. How will you ensure that thread T2 runs after T1 and thread T3 runs after T2?**

Use the join() method to ensure the order of execution.

thread1.start();

thread1.join();

thread2.start();

thread2.join();

thread3.start();

**Q29. Can we run a thread twice in Java?**

No, a thread cannot be run twice. Once a thread has completed execution, it cannot be restarted.

**Q30. Why are wait(), notify(), and notifyAll() defined in the Object class and not in the Thread class?**

These methods are defined in the Object class because they are used for inter-thread communication on shared objects. Every object in Java has a monitor, and these methods operate on the monitor of the object.

**Q31. What are the different ways to achieve synchronization in Java?**

* **synchronized keyword**: Use the synchronized keyword to lock methods or blocks of code.
* **Lock**: Use ReentrantLock for more flexible locking.
* **Atomic Variables**: Use atomic variables for thread-safe operations without synchronization.

**Q32. Why are wait() and notify() called from synchronized methods in Java?**

wait() and notify() must be called from synchronized methods or blocks because they require the current thread to hold the intrinsic lock on the object. This ensures that the thread has exclusive access to the object's monitor, preventing race conditions and ensuring proper coordination between threads.

**Scenario Questions**

**Q33. Multithreading Scenario Question1?**

This question seems to be a placeholder. Could you please provide more details or specify the scenario you have in mind?

**Q34. Can you give a scenario where thread starvation occurs?**

Thread starvation occurs when a thread is perpetually denied access to resources it needs for execution. This can happen in scenarios where:

* **High-Priority Threads**: High-priority threads continuously preempt lower-priority threads.
* **Resource Contention**: A thread is waiting for a resource that is constantly being used by other threads.

**Q35. Can you give a scenario where thread starvation occurs?**

This appears to be a duplicate of Q34. Please let me know if you need a different scenario or more details.

**Q36. Can we override start method in multithreading?**

Yes, you can override the start() method in a thread, but it is not recommended. The start() method is responsible for creating a new thread and invoking the run() method. Overriding it can lead to unexpected behavior and may prevent the thread from starting correctly.

**Q37. You are given a paragraph, which contains n number of words, you are given m threads. What you need to do is, each thread should print one word and give the control to the next thread, this way each thread will keep on printing one word, in case the last thread comes, it should invoke the first thread. Printing will repeat until all the words are printed in the paragraph. Finally, all threads should exit gracefully. What kind of synchronization will you use?**

You can use a combination of synchronized blocks and wait()/notify() methods to achieve this synchronization. Each thread will print a word and then notify the next thread to continue.

**Q38. We need to fetch a customer’s employee list from an API that returns only 100 employees at a time. The current solution uses a single thread invoking the API with different offsets to retrieve all employees (e.g., 30k), which increases processing time. How would you reduce the time taken for retrieving all the employees for a customer?**

You can use multiple threads to fetch the employee list in parallel. Each thread can request a different offset, reducing the overall processing time. Using an ExecutorService to manage the threads can help in efficiently handling the parallel requests.

**Q39. There are 10 API calls, I need to prioritize the first three responses and other 7 responses as low priority?**

You can use an ExecutorService with a priority queue to manage the API calls. Assign higher priority to the first three API calls and lower priority to the remaining seven. This ensures that the first three calls are processed before the others.

**Q40. We want to send requests to a 3rd party REST API in parallel, but there shouldn’t be more than 3 in-flight requests at any time. How would you achieve this?**

You can use a Semaphore to limit the number of concurrent requests. Initialize the semaphore with a permit count of 3. Each thread acquires a permit before sending a request and releases it after the request is completed.

import java.util.concurrent.Semaphore;

public class ApiRequestManager {

private static final int MAX\_CONCURRENT\_REQUESTS = 3;

private final Semaphore semaphore = new Semaphore(MAX\_CONCURRENT\_REQUESTS);

public void sendRequest(Runnable requestTask) {

try {

semaphore.acquire();

new Thread(() -> {

try {

requestTask.run();

} finally {

semaphore.release();

}

}).start();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

This code ensures that no more than 3 requests are in-flight at any given time.

Would you like more details on any of these topics or have any other questions?