Sequential Final QA

1. What is a thread? What do you understand about a “thread” in Java programming?

A thread is the smallest unit of execution within a process. It shares the same memory and resources with other threads in the same process. Threads allow concurrent execution, making programs faster and responsive.

---

2. What is multithreading?

Multithreading is the ability of process to run multiple threads simultaneously within a process. It enables concurrent execution of threads, improving CPU utilization and making applications more responsive. It’s is used in tasks like real-time processing or handling multiple client requests.

---

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

A process is an independent program with its own memory and resources, while threads are smaller execution units within a process that share memory and resources with the other threads in same process. Processes are isolated and slower to create, whereas threads are lightweight and allow easy communication with other threads within the same process.

---

4. Explain thread priority? What is the range of thread priority in Java?
   Thread priority in Java is a number that indicates the **importance** of a thread. It helps the **thread scheduler** decide which thread to run first when multiple threads are waiting to execute.

Thread priorities range from **1 to 10**, where:

- `Thread.MIN_PRIORITY = 1` (lowest priority)
- `Thread.NORM_PRIORITY = 5` (default priority)
- `Thread.MAX_PRIORITY = 10` (highest priority)

---

5. What is a daemon thread in Java?

In Java, a daemon thread is a background thread that performs tasks such as garbage collection, cleanup, or other low-priority activities. These threads are

designed to support the runtime environment of the application and do not prevent the Java Virtual Machine (JVM) from exiting. When all non-daemon (or user) threads have finished execution, the JVM will automatically terminate, even if daemon threads are still running.

---

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

Yes, you **can** call the `run()` method directly in Java, but it will not create a new thread. Instead, it simply executes the code within the `run()` method on the calling thread, just like a normal method call. If you want to execute the code in a separate thread, you **must** use the `start()` method.

---

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

Yes, two threads can execute a static method and a non-static method of the same class at the same time. A static method locks the class-level monitor, while a non-static method locks the object-level monitor. Since these are separate locks, there is no interference between the threads. This ensures both methods can run without causing a conflict.

---

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

The `synchronized` keyword in Java is used to ensure that only one thread at a time can access a critical section of code. It helps prevent thread interference and maintain consistency when multiple threads access shared resources.

Synchronized Method:

A synchronized method locks the entire method, ensuring that only one thread can execute it at a time.

Synchronized Block:

A synchronized block is a more specific way to achieve synchronization. Instead of locking the entire method, it locks only a specific section of code or an object. This approach provides better performance by limiting the scope of the lock.

---

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

If the `run()` method of the thread class is not overridden, it will execute its default implementation, which does nothing. This means the thread will be

created but won't perform any tasks. To define the thread's behavior, you must provide your own implementation of the `run()` method.

---

10. What are the benefits of using multithreading? What are the advantages of multithreading? What are some fundamental advantages of multithreading in Java?

Multithreading in Java allows multiple tasks to run concurrently, boosting performance and responsiveness. By executing tasks in parallel, it makes better use of CPU resources and reduces idle time.

Threads share the same memory, minimizing overhead and enabling efficient background processing for tasks like logging or monitoring.

Additionally, multithreading keeps applications responsive, especially in GUIs, where user interactions can continue while background operations are running. Overall, it simplifies multitasking and enhances the scalability of applications.

---

11. Explain the difference between Thread Scheduler and preemptive scheduling and time slicing ?

1. Thread Scheduler:

- The **Thread Scheduler** is part of the Java Virtual Machine (JVM) that manages the execution of threads.
- It decides which thread to run based on factors like thread priority and availability.
- The selection of threads by the scheduler is system-dependent, and it uses scheduling policies like **preemptive scheduling** or **time slicing**.

2. **Preemptive Scheduling**:

- **preemptive scheduling is one of the policy of Thread Scheduler.**
- In **preemptive scheduling**, the scheduler gives priority to high-priority threads.
- A thread can be forcefully paused (preempted) if a higher-priority thread becomes ready to execute.
- This approach ensures that important tasks are prioritized.

3. **Time Slicing**:

    - **Time Slicing is one of the policy of Thread Scheduler.**

- In **time slicing**, each thread is given a specific amount of CPU time (a time slice).
- Threads take turns executing, regardless of priority, within their allocated time.
- This ensures fair execution for all threads, focusing on equal distribution rather than priority.

---

12. How do you create a thread in Java? How to create thread in Java?

In Java, threads can be created in two ways:

1. **Extending the Thread class**: Create the subclass of Thread class and override the `run()` method and start the thread using the `start()` method.

   This approach limits flexibility because the class cannot extend any other class (Java doesn't support multiple inheritance).

2. **Implementing the Runnable interface**: Create a class that implements Runnable, define the `run()` method, and pass the instance of the class in to a `Thread` object before calling `start()`.

   This is preferred for larger applications because it allows the class to extend other classes and improves reusability and maintainability.

---

13. What are the states in the lifecycle of a Thread?

A thread execution goes through five states:

1. **New**: In the state thread is created but not started.
2. **Runnable**: Thread is ready for execution, awaiting CPU allocation.
3. **Running**: Thread is actively executing its task.
4. **Blocked/Waiting**: Thread waiting for a resource or signal.
5. **Terminated**: Execution is complete.

---

14. What does join() method do? What is the function of the join() method?

The `join()` method in Java is used to synchronize threads. Its primary function is to ensure that the current thread pauses execution until the thread on which `join()` is called completes its execution.

For example, if Thread A calls `ThreadB.join()`, then Thread A will pause and wait for Thread B to finish before resuming its own execution. This is useful when you

need one thread to finish a task before the next thread proceeds.

▼ Code

```
public class JoinExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread: " + i);
                try {
                    Thread.sleep(500); // Simulating some work
                } catch (InterruptedException e) {
                    System.out.println(e.getMessage());
                }
            }
        });

        thread.start();

        try {
            thread.join(); // Current thread waits for 'thread' to complete
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Main thread resumes after thread finishes.");
    }
}
```

- Join() method ensures thread synchronization and avoids race conditions.
- It throws `InterruptedException` if the waiting thread is interrupted. So to avoid exception we should have to call join within try-catch or throw the `InterruptedException` .

---

15. Describe the purpose and working of the sleep() method? What is the function of the sleep() method?
    The `sleep()` method in Java is used to pause the execution of the current thread for a specified period of time. It is part of the `Thread` class and is primarily

used to simulate delays or give other threads a chance to execute in a multitasking environment.

When you invoke `Thread.sleep(milliseconds)`, the thread on which the method is called temporarily transitions to the **TIMED_WAITING** state. During this state:

1. The thread will not execute any code.
2. Other threads can continue running in parallel.
3. Once the specified time is complete, the thread moves back to the **RUNNABLE** state.

- If another thread interrupts the sleeping thread, the method throws InterruptedException exception.
- The sleeping thread holds a lock, even if the thread in sleeping state, which can lead to deadlocks if not managed carefully.
- The `sleep()` method is especially useful in debugging, testing, or simulating timed operations in multithreaded programs.

▼ Code

```
public class SleepExample {
    public static void main(String[] args) {
        System.out.println("Main thread starts.");

        for (int i = 1; i <= 5; i++) {
            System.out.println("Step " + i);
            try {
                Thread.sleep(1000); // Pauses for 1000ms (1 second)
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e.getMessage());
            }
        }

        System.out.println("Main thread ends.");
    }
}
```

16. What is the purpose of the wait() method in Java? What do you understand about the wait() function in Java?
    The
    `wait()` method in Java is a key component of inter-thread communication mechanism, especially when working with shared resources. It allows one thread to pause its execution until another thread signals it to resume. This method is part of the `Object` class, not the `Thread` class, which makes it available for all objects.

    Purpose of `wait()`:

- It is used to achieve coordination between threads.
- It ensures efficient use of CPU by pausing a thread instead of running it in a loop (busy-waiting).
- It allows a thread to wait until certain conditions are met or until notified by another thread.
- When the thread calling `wait()` method then it must be inside a synchronized block or method for the same object.
- It need to call `notify()` **or** `notifyAll()`: by the other threads to wake up the waiting thread.
- **It Throws** `InterruptedException`: If the thread is interrupted while waiting, this exception is thrown.

The `wait()` method plays a critical role in managing complex multithreading scenarios, ensuring threads communicate effectively.

▼ Code

```
class WaitNotifyExample {
    public static void main(String[] args) {
        Object lock = new Object();

        Thread producer = new Thread(() -> {
            synchronized (lock) {
                System.out.println("Producer thread working...");
                try {
                    System.out.println("Producer waiting...");
                    lock.wait(); // Pauses here, releasing the lock
```

```
            } catch (InterruptedException e) {
                System.out.println("Interrupted: " + e.getMessage());
            }
            System.out.println("Producer resumed.");
        }
    });

    Thread consumer = new Thread(() -> {
        synchronized (lock) {
            System.out.println("Consumer thread working...");
            try {
                Thread.sleep(2000); // Simulates some processing delay
            } catch (InterruptedException e) {
                System.out.println("Interrupted: " + e.getMessage());
            }
            System.out.println("Consumer notifying...");
            lock.notify(); // Signals producer to resume
        }
    });

    producer.start();
    consumer.start();
  }
}
```

17. Why must the wait() method be called from the synchronized block?

The `wait()` method must be called from a synchronized block or synchronized method because it needs the thread to own the monitor (lock) of the object on which `wait()` is called.

Reason 1: Coordination via Locks

- The `wait()`, `notify()`, and `notifyAll()` methods are designed for thread communication and rely on the concept of monitors (locks).
- A thread can only "wait" on an object if it already holds the lock of that object. This ensures proper coordination between the waiting thread and the notifying thread.

Reason 2: Prevent Race Conditions

- When a thread calls `wait()`, it releases the lock it holds on the object and enters the waiting state.
- If the thread doesn't hold the lock initially, calling `wait()` without synchronization could lead to unpredictable behavior, such as a race condition where multiple threads modify shared data concurrently.
  If you attempt to call `wait()` outside of a synchronized block or method, Java throws an `IllegalMonitorStateException`. This is because the `wait()` method must enforce thread safety when managing locks and communication.

▼ Code

```
class WaitExample {
    public static void main(String[] args) {
        Object lock = new Object();

        Thread thread = new Thread(() -> {
            synchronized (lock) { // Thread must hold the lock
                try {
                    System.out.println("Thread is waiting...");
                    lock.wait(); // Wait releases the lock
                } catch (InterruptedException e) {
                    System.out.println("Interrupted: " + e.getMessage());
                }
            }
        });

        thread.start();
    }
}
```

---

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

The `wait()` and `sleep()` methods in Java are essential for managing threads, but they serve different purposes and function differently. The `wait()` method, which belongs to the `Object` class, is used for thread communication. It pauses the execution of the current thread and releases the lock on the object it is waiting

on, allowing other threads to proceed. The paused thread can resume only after another thread calls `notify()` or `notifyAll()` on the same object. In contrast, the `sleep()` method, which belongs to the `Thread` class, is used to pause the execution of the current thread for a specified time without releasing any locks it holds. While `wait()` is called within a synchronized block or method to ensure proper coordination between threads, `sleep()` does not require synchronization and resumes execution automatically after the sleep duration. Both methods throw exceptions (`InterruptedException` for `sleep()` and `IllegalMonitorStateException` for improper use of `wait()`), but their application differs—`wait()` is used for thread coordination, whereas `sleep()` is for simulate delays.

---

19. What's the difference between notify() and notifyAll() ?

The `notify()` and `notifyAll()` methods in Java are part of the `Object` class and are used to manage thread communication by waking up threads waiting on an object's monitor. However, they differ in the way they handle waiting threads:

- `notify()`: This method wakes up a single, thread that is waiting on the object's monitor. It does not guarantee which thread will be chosen if multiple threads are waiting. This is generally used when you only need to signal one thread to proceed, from multiple threads are waiting.
- `notifyAll()`: This method wakes up all threads waiting on the object's monitor. Only the threads that can reacquire the lock will proceed, and the others will continue to wait. This is useful when multiple threads are waiting for the same condition to change and you want to ensure all of them get a chance to act.

---

20. What is a race condition? How can it be avoided?
    A
    race condition occurs in multithreaded programs when two or more threads access shared resources (e.g., variables, files, or objects) simultaneously, and the outcome of the program depends on the sequence or timing of their execution. This leads to unpredictable or incorrect behavior because threads are in "race" to access or modify the resource.

    ▼ Code

```
class Counter {
    private int count = 0;
```

```
    public void increment() {
        count++; // Not thread-safe
    }

    public int getCount() {
        return count;
    }
}

public class RaceConditionExample {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

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

Explain the concept of thread synchronization and how it is implemented in Java?

Thread synchronization in Java is a mechanism to control the access of multiple threads to shared resources in order to prevent conflicts, ensure data consistency, and avoid race conditions. When multiple threads interact with shared data or resources simultaneously, synchronization ensures that only one thread can access the critical section of code at a given time, making the operations thread-safe.

To avoid the race condition we can use the below some synchronization ways.

1. **Use Synchronization:**
   Use the
   `synchronized` keyword to ensure that only one thread at a time can access the shared resource.

▼ Code

```
public synchronized void increment() {
    count++;
}
```

2. **Use Lock Mechanism:**
   Use classes like
   `ReentrantLock` to provide explicit locking for critical sections.

▼ Code

```
private final ReentrantLock lock = new ReentrantLock();

public void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

3. **Atomic Variables:**
   Use
   `java.util.concurrent.atomic` classes like `AtomicInteger` for operations that need to be thread-safe.

▼ Code

```
private AtomicInteger count = new AtomicInteger(0);

public void increment() {
    count.incrementAndGet();
}
```

4. **Thread-safe Collections:**
   When working with collections, use thread-safe implementations like `ConcurrentHashMap` or use wrapper methods like `Collections.synchronizedList()`.

---

21. What is a deadlock? How can it be avoided? What do you understand about Deadlock situations in Java Multithreading?

A deadlock in Java multithreading occurs when two or more threads are blocked forever, and waiting for each other to release a resource or lock. It typically happens in scenarios where multiple threads hold certain locks and simultaneously try to acquire locks held by others, creating a cyclic that prevents any thread from proceeding.
To prevent deadlocks, you can apply the following strategies:

1. **Lock Ordering**:
   Always acquire locks in a specific, consistent order across all threads. For example, if you have `resource1` and `resource2`, ensure that all threads acquire `resource1` first, followed by `resource2`.
2. **Try-Lock Mechanism**:
   Use `tryLock()` from the `ReentrantLock` class to attempt to acquire a lock without blocking indefinitely.
3. **Thread-safe Classes**:
   Use thread-safe classes like `ConcurrentHashMap` and avoid creating custom synchronization whenever possible.
4. **Avoid Nested Locks**:
   Minimize situations where a thread holds one lock and tries to acquire another.

---

22. How do you detect deadlock situations in Java?

Since Java does not provide built-in deadlock detection mechanisms, developers typically rely on monitoring tools, logging, and debugging techniques to identify deadlocks.

Deadlocks in Java occur when two or more threads are indefinitely blocked, each waiting for the other to release a resource, leading to a cyclic dependency.

1. They can be detected using thread dumps, which reveal thread states and locks,
2. or through monitoring tools like VisualVM and JConsole that identify blocked threads.
3. Developers can also use the `ThreadMXBean` class to programmatically detect deadlocked threads during runtime.

---

23. What is Livelock? What happens when it occurs?

    A

    livelock is a situation in multithreading where threads are actively responding to each other but fail to make progress because they are continuously changing states. Unlike a deadlock, where threads are completely stuck waiting for resources, in a livelock, threads remain active yet unable to accomplish their tasks.

    This happens when threads keep responding to each other over and over, causing a loop of repeated actions.

    ▼ Code

```
class Resource {
    private boolean inUse = false;

    public synchronized void toggle() {
        inUse = !inUse;
    }

    public synchronized boolean isInUse() {
        return inUse;
    }
}
```

```
public class LivelockExample {
    public static void main(String[] args) {
        Resource resource = new Resource();

        Thread thread1 = new Thread(() -> {
            while (resource.isInUse()) {
                System.out.println("Thread 1 waiting...");
                resource.toggle(); // Toggles usage to avoid conflict
            }
        });

        Thread thread2 = new Thread(() -> {
            while (!resource.isInUse()) {
                System.out.println("Thread 2 waiting...");
                resource.toggle(); // Toggles usage to avoid conflict
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

we can avoid the Livelock via

1. **Time Constraints:** Set a time limit for how long threads should keep retrying, so they don't keep trying forever for no reason.
2. **Priority Mechanisms:** Give threads a clear order of priority, so one thread can finish its work before the other starts.

---

24. What do you understand by inter-thread communication? What is inter-thread communication? What are some functions used to perform inter-thread communication in Java? How do threads communicate with each other? What do you mean by inter-thread communication ?

    Inter-thread communication is the mechanism where the threads witin the same process communicate with each other using methods like `wait()`, `notify()`, and `notifyAll()`. It's used when one thread needs to wait for another to

complete something before it can continue. .This It helps avoid issues like conflicts, data inconsistency and maintain consistency.

In Java, inter-thread communication is done using these key methods (from `Object` class):

- `wait()` – causes the thread to wait until another thread calls `notify()`
- `notify()` – wakes up a single waiting thread
- `notifyAll()` – wakes up all waiting threads

These methods must be called inside a `synchronized` block.

---

25. What is context switching? What do you understand by context switching?

Context switching is the process of saving the state of a currently running thread or process and restoring the state of another. It allows multitasking by switching between threads or processes, enabling efficient CPU utilization. However, frequent context switching may lead to performance issue.

---

26. What is BlockingQueue?

BlockingQueue is a thread-safe queue in `java.util.concurrent` package that supports operations like inserting or removing elements with built-in blocking. That is specifically designed to support operations where Producers wait when the queue is full, and consumers wait when it's empty. It simplifies handling producer-consumer problems by managing thread coordination automatically and preventing race conditions.

---

27. What is a thread pool? Why is it used?
    A
    thread pool is a collection of pre-created worker threads that are managed and reused by a framework (such as `java.util.concurrent.ExecutorService`) to perform multiple tasks. Instead of creating a new thread for every task, thread pools reuse existing threads, which enhances performance and reduces the overhead of thread creation and destruction.
    Additionally, thread pools help handle tasks efficiently by queueing them when no thread is available.

Here are some common types of thread pools in Java:

1. **Fixed Thread Pool**: A pool with a fixed number of threads, where extra tasks are queued until a thread becomes available

(`Executors.newFixedThreadPool()`).

2. **Cached Thread Pool**: A pool that creates new threads as needed but reuses idle threads when available (`Executors.newCachedThreadPool()`).
3. **Single-Thread Executor**: A single thread executes tasks sequentially, ensuring only one task runs at a time (`Executors.newSingleThreadExecutor()`).
4. **Scheduled Thread Pool**: A pool designed for tasks that need to run after a delay or periodically (`Executors.newScheduledThreadPool()`).

Each type serves specific needs based on the application's concurrency requirements!

---

28. What is ConcurrentHashMap and Hashtable? Why is ConcurrentHashMap considered faster than Hashtable?

- It is part of the `java.util.concurrent` package and is specifically designed for high-performance, thread-safe operations in multi-threaded environments.
- Instead of locking the entire map, it uses a concept called **segmented locking** (prior to Java 8). The map is divided into smaller segments (or buckets), and only the segment being modified is locked. In Java 8 and later, it uses a more sophisticated mechanism called **CAS (Compare-And-Swap)** for better performance.
- Read operations generally do not require locks, and write operations lock only the affected bucket, allowing better concurrency.
- It is a legacy class in Java that implements a hash table.
- It is synchronized, meaning all methods are thread-safe and can be accessed by only one thread at a time.

ConcurrentHashMap is faster than Hashtable because it uses a more efficient locking mechanism. Instead of locking the entire map, it employs segmented locking (pre-Java 8) or CAS (Compare-And-Swap) operations (Java 8 and later). This allows multiple threads to access different parts of the map concurrently, significantly reducing conflict.

---

29. What do you mean by the ThreadLocal variable in Java?
    A
    ThreadLocal variable in Java gives each thread its own separate copy of the variable, so they don't interfere with each other.
    For example, if two threads use the same

ThreadLocal variable, each thread gets its own separate value. Hence one thread does won't affect the other. This is useful for keeping data specific to each thread, like user sessions, transaction IDs, or other thread-specific info.

▼ Code

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadLocalExample {

    // Define a ThreadLocal variable
    private static ThreadLocal<Integer> threadLocalVariable = ThreadLocal

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        Runnable task = () -> {
            // Get the current thread's value
            int value = threadLocalVariable.get();
            System.out.println(Thread.currentThread().getName() + " - Initial v

            // Update the current thread's value
            threadLocalVariable.set(value + 1);
            System.out.println(Thread.currentThread().getName() + " - Update
        };

        // Submit multiple tasks to the executor
        for (int i = 0; i < 5; i++) {
            executor.submit(task);
        }

        executor.shutdown();
    }
}
```

30. What is the purpose of the finalize() method?
    The
    `finalize()` method in Java is part of the garbage collection mechanism. It is called by the garbage collector before an object is removed from memory. The purpose of `finalize()` is to allow an object to clean up resources it holds, such as closing files, network connections, or releasing other system resources, **before** it is destroyed.
    It is rarely used in modern Java programming because:

    a. You can't predict when or even if the `finalize()` method will be called.

    b. It can introduce delays in garbage collection.

    c. Starting from Java 9, `finalize()` is marked as deprecated, and alternatives such as **try-with-resources** or explicit resource cleanup methods

---

31. What is CyclicBarrier and CountDownLatch?

Both **CyclicBarrier** and **CountDownLatch** are synchronization utilities in Java designed to coordinate threads, but they serve different purposes.

CountDownLatch:

- CountDownLatch is used when one or more threads need to wait until a specific number of operations, performed by other threads, are complete.
- It has a counter initialized with a number, indicating the number of tasks to wait for.
- Threads call `countDown()` when they finish their tasks, and the waiting thread calls `await()` to block until the counter reaches zero.

Example Use Case: Suppose a main thread needs to start processing only after all worker threads have finished their tasks.

CyclicBarrier:

- It is used to make a group of threads wait for each other at a common point (the barrier) before all threads can proceed together.
- Unlike CountDownLatch, the barrier is reusable, meaning once all threads reach the barrier, it can be reset and used again.
- Threads call `await()` to wait at the barrier until the required number of threads reaches it.

Example Use Case: Suppose multiple threads are working on parts of a dataset. You want all of them to wait until everyone finishes their current task (phase) before moving on to the next one.

---

32. What is semaphore?

A **Semaphore** in Java is a thread synchronization construct that controls access to a shared resource by maintaining a set of permits. Threads can acquire permits using the `acquire()` method and release permits using the `release()` method. If no permits are available, a thread attempting to acquire one will block until a permit is released by another thread.

Semaphores come in two types:

1. **Counting Semaphore**: Allows multiple threads to access a resource, up to a specified limit (e.g., controlling access to a pool of database connections).
2. **Binary Semaphore**: Similar to a lock, it allows only one thread to access the resource at a time.

Example Use Case:

Imagine a printing system with multiple printers but limited slots. A Semaphore can ensure that only a specific number of threads (e.g., equal to the available printers) can perform the print operation simultaneously.

---

33. What is a shutdown hook ?
    A
    shutdown hook in Java is a special mechanism provided by the JVM to execute custom cleanup tasks before the program terminates. These hooks are essentially threads that are registered using the `Runtime.getRuntime().addShutdownHook(Thread hook)` method. They get triggered when the JVM begins its shutdown process, whether due to normal termination (e.g., program completion) or calling `System.exit()`.

---

34. What is busy spinning?

    Busy spinning is a technique in multithreading where a thread keeps checking for a condition (like a resource becoming available) in a loop without pausing. Instead of waiting passively, it stays active and keeps running instructions until the condition is met.
    It avoids the overhead of thread context switching, which can improve performance in certain scenarios.

Example:
Busy spinning can be useful in
real-time systems where the condition is expected to be resolved very quickly, and the cost of blocking the thread is higher than the cost of spinning.

---

35. Explain Thread Group. Why should we not use it?
    A
    Thread Group in Java is a mechanism to group multiple threads under a single unit. It allows you to manage threads collectively, such as starting, stopping, or interrupting all threads within the group. You can also create a hierarchy of thread groups by associating sub-groups to parent groups. Thread Groups provided a way to organize threads in earlier Java versions, they are now considered outdated, and modern concurrency frameworks offer safer and more robust solutions.

---

36. What are some common problems associated with multithreading?

Multithreading can be powerful, but it comes with challenges. Some common problems include:

1. **Race Conditions**: When multiple threads access shared resources and try to modify them concurrently, it can lead to unpredictable behavior if proper synchronization is not used.
2. **Deadlocks**: When two or more threads wait indefinitely for each other to release locks, preventing further progress.
3. **Thread Interference**: **Thread Interference** happens when multiple threads try to use or change the same data at the same time. If they’re not managed properly, this can cause wrong or incomplete results.
4. **Starvation**: When a thread is unable to access resources because other higher-priority threads are consuming them continuously.
5. **Context Switching Overhead**: Frequent context switching between threads can degrade performance due to the CPU's time spent managing threads rather than executing tasks.
6. **Priority Inversion**: When a high-priority thread gets stuck waiting because a lower-priority thread is holding a needed resource. This messes up the normal priority order and can cause delays.

---

37. Describe different types of locks and their uses ?

- **Intrinsic Locks (Monitors):** These come with the synchronized keyword and help lock methods or blocks of code so only one thread can use them at a time. Good for basic thread safety but not very flexible.
- **ReentrantLock:**
  It allows the same thread to acquire the lock multiple times without getting blocked. It also offers more control than `synchronized`, like setting timeouts, checking if the lock is held
- **ReadWriteLock:** Allows many threads to read at once, but only one to write. Best when you read more than you write.
- **StampedLock:** A lightweight lock that improves performance in read-heavy scenarios. Allows optimistic reads without locking unless needed.
- **Semaphore:** Limits the number of threads accessing a resource at once. Useful for controlling things like database connections.
- **SpinLock:** Keeps trying to get the lock in a loop without sleeping. Fast for quick tasks but wastes CPU on long waits.

---

38. How would you handle a situation where multiple threads need to access and modify a shared resource?

To handle a situation where multiple threads need to access and modify a shared resource, you would use synchronization mechanisms to ensure thread safety and prevent conflicts or errors. Here's a simple approach:

1. **Use Locks**: Utilize locks like `synchronized` blocks, `ReentrantLock`, or other locking mechanisms to ensure only one thread can access the resource at a time.
2. **Atomic Variables**: For simple updates, use atomic classes like `AtomicInteger` or `AtomicBoolean` from the `java.util.concurrent.atomic` package. These classes handle concurrency without explicit locking.
3. **ReadWriteLock**: It allows multiple readers while restricting writes to one thread at a time.
4. **Avoid Deadlocks**: Be careful with nested locks and ensure threads acquire and release locks in the same order to prevent deadlocks.
5. **Semaphore**: If a specific number of threads need access to the resource at the same time, use a `Semaphore` to control concurrency.

---

39. What strategies would you use to improve the performance of a multithreaded application?

    Improving the performance of a multithreaded application we have to ensure some points like:

    - **Avoid Over-synchronization**: Synchronize only where necessary. Overusing synchronization can lead slow down the application.
    - **Reduce Context Switching**: Due to too many frequent context switching the the performance of application is reduces.
    - **Thread Pooling**: Use thread pools (e.g., `ExecutorService`) instead of creating threads manually. This reuses threads and avoids the overhead of frequent thread creation and destruction.
    - **Performance Monitoring**: Continuously monitor the application using profiling tools to identify thread-related issues like deadlocks or contention hotspots.
    - **Thread-safe Collections**: Use thread-safe collections like `ConcurrentHashMap`, `CopyOnWriteArrayList`, or `ConcurrentLinkedQueue` when multiple threads need to access and modify shared data. These collections handle synchronization internally, ensuring safe operations without manual locking.

---

40. Explain the producer-consumer problem and how you would solve it in Java ?
    The
    Producer-Consumer Problem is a classic example of multithreading synchronization in which two types of threads—producers and consumers—share a common buffer or queue:

    - **Producer Threads**: Produce data (e.g., items) and add it to the buffer.
    - **Consumer Threads**: Consume data (e.g., items) from the buffer.
    - **Challenge**: Ensure that producers don't overwrite the buffer when it's full, and consumers don't read from an empty buffer, while maintaining thread safety and avoiding deadlocks or race conditions.

To solve this problem, you can use synchronization tools like:

1. **Using `wait()` and `notify()`:**
   These methods, used with intrinsic locks (

`synchronized`), help coordinate producers and consumers. Producers wait when the buffer is full, and consumers wait when the buffer is empty.

▼ Code

```
import java.util.LinkedList;

class ProducerConsumer {
    private final LinkedList<Integer> buffer = new LinkedList<>();
    private final int MAX_CAPACITY = 5;

    public synchronized void produce(int value) throws InterruptedExceptio
        while (buffer.size() == MAX_CAPACITY) {
            wait(); // Wait if buffer is full
        }
        buffer.add(value);
        System.out.println("Produced: " + value);
        notify(); // Notify consumers
    }

    public synchronized void consume() throws InterruptedException {
        while (buffer.isEmpty()) {
            wait(); // Wait if buffer is empty
        }
        int value = buffer.removeFirst();
        System.out.println("Consumed: " + value);
        notify(); // Notify producers
    }
}

public class ProducerConsumerTest {
    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    pc.produce(i);
                }
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    pc.consume();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

2. **Using** `BlockingQueue`:
   The `java.util.concurrent.BlockingQueue` provides a ready-to-use solution for the producer-consumer problem. It handles synchronization internally, eliminating the need for `wait()` and `notify()`.

▼ Code

Both methods achieve the same goal, but the `BlockingQueue` approach is modern and recommended.
It prevents errors associated with manual use of `wait()` and `notify()`.
It simplifies the code by handling synchronization automatically

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ProducerConsumerBlockingQueue {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);
```

```
        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    queue.put(i); // Adds item to the queue
                    System.out.println("Produced: " + i);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    int value = queue.take(); // Takes item from the queue
                    System.out.println("Consumed: " + value);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

41. How would you implement a concurrent data structure, like a thread-safe queue?
    To implement a
    thread-safe queue, you need to ensure that multiple threads can safely access and modify the queue without causing race conditions or inconsistencies. Below is a simple way to achieve this:

    Using `synchronized` for Basic Implementation

    You can manually synchronize methods to make a basic thread-safe queue:

    ▼ Code

```
import java.util.LinkedList;
import java.util.Queue;

public class ThreadSafeQueue<T> {
    private final Queue<T> queue = new LinkedList<>();
    private final int capacity;

    public ThreadSafeQueue(int capacity) {
        this.capacity = capacity;
    }

    // Add an element to the queue
    public synchronized void enqueue(T item) throws InterruptedException
        while (queue.size() == capacity) {
            wait(); // Wait if the queue is full
        }
        queue.add(item);
        notifyAll(); // Notify waiting threads
    }

    // Remove and return an element from the queue
    public synchronized T dequeue() throws InterruptedException {
        while (queue.isEmpty()) {
            wait(); // Wait if the queue is empty
        }
        T item = queue.poll();
        notifyAll(); // Notify waiting threads
        return item;
    }
}
```

Using `BlockingQueue` for Easier Implementation

The `BlockingQueue` class from `java.util.concurrent` provides a built-in thread-safe queue. It internally handles synchronization, making the implementation simpler and more reliable:

▼ Code

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueExample {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

        // Producer thread
        new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    queue.put(i); // Add element to the queue
                    System.out.println("Produced: " + i);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        // Consumer thread
        new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    System.out.println("Consumed: " + queue.take()); // Remove e
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

---

12. What are some design patterns used in multithreading?

There are some common design patterns used in multithreading:

1. **Producer-Consumer Pattern**: Helps synchronize the workflow between producer threads (which generate data) and consumer threads (which

process the data). Shared queues and blocking mechanisms are often used to implement this.

2. **Singleton Pattern**: Ensures that only one instance of a class is created. In multithreaded environments, this involves making the instance creation thread-safe using synchronization or the `volatile` keyword.
3. **Thread Pool Pattern**: Reuses a fixed pool of threads to execute tasks, reducing the overhead of creating and destroying threads repeatedly. This is commonly implemented using `ExecutorService`.
4. **Read-Write Lock Pattern**: Separates read and write operations using locks, allowing multiple threads to read data simultaneously while restricting writes to one thread at a time.
5. **Future Pattern**: This represents the result of an asynchronous computation that may not be completed yet. Threads can retrieve the result once it's ready, implemented via `Future` or `CompletableFuture`.
6. **Fork-Join Pattern**: Used for parallel tasks where a large task is divided into smaller tasks and processed concurrently by threads. This is implemented in the `ForkJoinPool`.

---

43. What are some ways to handle exceptions in multithreading?

- **Try-Catch Blocks**:
  Place try-catch blocks within the thread's `run()` method and handle exceptions during execution.
- **UncaughtExceptionHandler**:
  Set a global handler for threads using `Thread.setUncaughtExceptionHandler()`. This ensures uncaught exceptions are caught and handled properly.
- **Future Interface**:
  Use the `Future` interface for threads created with `Callable`. You can call `Future.get()` to retrieve exceptions thrown during execution.

▼ Not mostly asked Question List

1. Write unit tests with various scenarios, use thread-safe testing libraries, stress tests, and tools like profilers and debuggers to detect race conditions and deadlocks.
