Practise Questions with Answers

Multithreading

Q1. What is the difference between creating a thread using **Thread**, **Runnable**, and **ExecutorService**?

Answer:

1. Creating a Thread using Thread class

Extend the Thread class and override the run() method.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Points

- Directly represents a thread
- Task and thread logic are tightly coupled

- Java does not support multiple inheritance, so extending Thread blocks extending another class
- Not flexible for large applications

2. Creating a Thread using Runnable interface

Implement Runnable and pass it to a Thread object.

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task running");
    }
}

public class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyTask());
        t.start();
    }
}
```

Points

- Separates task from thread
- Allows extending another class

- Better design than extending Thread
- Still requires manual thread management

3. Using ExecutorService (Thread Pool)

submit tasks to an executor instead of creating threads manually.

```
ExecutorService executor = Executors.newFixedThreadPool(3);


executor.submit(() -> {
    System.out.println("Task executed by thread pool");
});


executor.shutdown();
```

Points

- Introduced in java.util.concurrent
- Manages a pool of threads
- You submit tasks, not threads
- Best suited for real-world applications

- Which approach is preferred in real applications and why?

Answer:

Preferred: ExecutorService

Reasons:

- Efficient thread reuse

- Better performance
- Cleaner and scalable design
- Built-in lifecycle management
- Easy error handling and monitoring

Runnable + ExecutorService is the industry standard.

- What problems does ExecutorService solve compared to manual thread creation?

Answer:

Problems with Manual Thread Creation

- Too many threads → OutOfMemoryError
- High CPU context switching
- No control over thread lifecycle
- Difficult to manage and scale

How ExecutorService Solves them

Problem-	Solution
Creating too many threads	Uses thread pools
Performance overhead	Reuses threads
Thread lifecycle handling	shutdown(), awaitTermination()
Task scheduling	Built-in queue management
Error handling	Future, exception tracking

Q2. You submit a task using ExecutorService.submit().

- What does the submit() method return?

Answer: submit() returns a Future<V> object.

- V is the result type of the task
- If the task is:
    - Callable<V> → returns Future<V>
    - Runnable → returns Future<?> (result is null)

```
Future<Integer> future = executor.submit(() -> 10 + 20);
```

- How do you retrieve the result of the task?

Answer:
Use Future.get()

```
Integer result = future.get();
System.out.println(result); // 30
```

- get() retrieves the returned value
- It throws:
    - InterruptedException
    - ExecutionException (if task throws exception)

- What happens if Future.get() is called before the task is complete?

Answer:
The calling thread blocks (waits) until:

- Task finishes execution, or
- Task throws an exception, or

- Task is cancelled

System.out.println("Waiting...");

Integer result = future.get(); // blocks here

System.out.println("Done: " + result);

Non-blocking check options:

```
if (future.isDone()) {
   System.out.println(future.get());
}
```

Q3. Why is synchronization required in multi-threaded programs?

Answer: In a multi-threaded program, multiple threads share the same resources (objects, variables, files, etc.).

Synchronization is required to:

1. Prevent race conditions – where multiple threads modify shared data at the same time, leading to inconsistent results.
2. Ensure data consistency – only one thread accesses critical sections at a time.
3. Maintain thread safety – avoid unpredictable behavior and bugs.
4. Provide mutual exclusion – control access to shared resources.

Without synchronization, the final output may depend on thread execution order, which is not guaranteed.

- Explain the purpose of wait() and notify(). Show via an example

Answer: wait() and notify() are used for inter-thread communication.

Method. Purpose

wait()	Causes the current thread to release the lock and wait until another thread notifies it
notify()	Wakes up one waiting thread
notifyAll()	Wakes up all waiting threads

They are commonly used in Producer–Consumer problems.

Example: Producer–Consumer using wait() and notify()

```
class SharedResource {
    private int data;
    private boolean hasData = false;

    synchronized void produce(int value) {
        while (hasData) {
```

```
            try {
                wait();   // Producer waits
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        hasData = true;
        System.out.println("Produced: " + data);
        notify();  // Notify consumer
    }

    synchronized void consume() {
        while (!hasData) {
            try {
                wait();   // Consumer waits
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
        System.out.println("Consumed: " + data);
        hasData = false;
        notify();  // Notify producer
    }
}

public class Test {
    public static void main(String[] args) {
        SharedResource resource = new
SharedResource();

        Thread producer = new Thread(() ->
resource.produce(100));
        Thread consumer = new
Thread(resource::consume);

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

}

- Why must they be called inside a synchronized block?

Answer:

1. They work on object monitors (locks)
    - A thread must own the lock of the object before calling wait() or notify().
2. Avoids IllegalMonitorStateException
    - If called without synchronization, JVM throws:

java.lang.IllegalMonitorStateException

3. Ensures thread safety
    - Guarantees that shared data is accessed safely while threads wait or notify.
4. Atomic operation
    - Releasing the lock and entering the waiting state happens atomically.

Internally:

- wait() → releases the lock + pauses the thread
- notify() → signals a waiting thread but does not release the lock immediately

Q4. Why are collections like ArrayList and HashMap not thread-safe?

Answer: ArrayList and HashMap are not thread-safe because:

1. No internal synchronization
    - Their methods are not synchronized, so multiple threads can modify them simultaneously.
2. Race conditions
    - Concurrent updates (add/remove/put) can corrupt internal data structures.
    - Example: Two threads adding elements at the same time may overwrite data.
3. Performance-first design
    - These collections are designed for single-threaded or externally synchronized use.
    - Adding synchronization internally would reduce performance for common use cases.

Example problem:

- Multiple threads updating a HashMap can cause data inconsistency or even an infinite loop during resizing (older implementations).

- Name one thread-safe collection.

Answer: Some thread-safe collections in Java:

- Vector
- Hashtable
- Collections.synchronizedList(new ArrayList<>())
- ConcurrentHashMap
- CopyOnWriteArrayList

Example answer:

ConcurrentHashMap is a commonly used thread-safe collection.

- When would you prefer ConcurrentHashMap over Collections.synchronizedMap()?

Answer: I will prefer ConcurrentHashMap when:

1. High concurrency is required
    - Multiple threads need to read and write simultaneously.
2. Better performance
    - ConcurrentHashMap allows concurrent reads and partial locking for writes.
    - Collections.synchronizedMap() uses a single lock, causing contention.

3. No blocking during reads
    - Reads in ConcurrentHashMap are generally non-blocking.
    - In synchronizedMap, even reads require acquiring the lock.
4. Safe iteration
    - Iterators in ConcurrentHashMap are weakly consistent (no ConcurrentModificationException).
    - synchronizedMap requires manual synchronization during iteration.

Q5. What is a **deadlock** in Java?

Answer : A deadlock occurs in Java when two or more threads are blocked forever, each waiting for a resource held by another thread.

In other words, threads are waiting on each other indefinitely, and none can proceed.

Example:

```
class Resource {
   String name;
   Resource(String name) { this.name = name; }
}

public class DeadlockExample {
```

```
    public static void main(String[] args) {
        Resource r1 = new Resource("Resource1");
        Resource r2 = new Resource("Resource2");

        // Thread 1 tries to lock r1 then r2
        Thread t1 = new Thread(() -> {
            synchronized (r1) {
                System.out.println("Thread 1 locked " +
r1.name);
                try { Thread.sleep(100); } catch
(InterruptedException e) {}
                synchronized (r2) {
                    System.out.println("Thread 1 locked " +
r2.name);
                }
            }
        });

        // Thread 2 tries to lock r2 then r1
        Thread t2 = new Thread(() -> {
            synchronized (r2) {
                System.out.println("Thread 2 locked " +
r2.name);
```

```
            try { Thread.sleep(100); } catch
(InterruptedException e) {}
            synchronized (r1) {
                System.out.println("Thread 2 locked " +
r1.name);
            }
        }
    });

    t1.start();
    t2.start();
  }
}
```

Explanation:

1. Thread 1 locks Resource1 and waits for Resource2.
2. Thread 2 locks Resource2 and waits for Resource1.
3. Both threads wait forever, causing a deadlock.

- Explain a real-world scenario where a deadlock can occur. Can you show a piece of code where a deadlock can potentially occur.

Answer:

Real-world scenario

Think of a dining scenario:

- Two people (threads) want two forks (resources) to eat.
- Person A picks up Fork 1, waits for Fork 2.
- Person B picks up Fork 2, waits for Fork 1.
- Both are waiting forever — a deadlock occurs.

Example of a potential deadlock:

```
class Resource {
    String name;

    Resource(String name) { this.name = name; }

    synchronized void use(Resource other) {
        System.out.println(Thread.currentThread().getName() + " using " + name);
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + " wants " + other.name);
        synchronized (other) {  // Waiting for other resource
            System.out.println(Thread.currentThread().getName() + " acquired " + other.name);
        }
    }
}

public class DeadlockDemo {
    public static void main(String[] args) {
```

```
        Resource r1 = new Resource("Resource1");
        Resource r2 = new Resource("Resource2");

        Thread t1 = new Thread(() -> r1.use(r2), "Thread-1");
        Thread t2 = new Thread(() -> r2.use(r1), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Explanation:

- Thread-1 locks r1 and waits for r2.
- Thread-2 locks r2 and waits for r1.
- Both threads are stuck → deadlock.

- Mention one way to prevent deadlocks in applications.

Answer:

One way to prevent deadlocks

1. Lock ordering / resource hierarchy
    - Always acquire locks in a fixed global order.
    - Example: If every thread locks r1 first and then r2, deadlock cannot occur.

Other strategies:

- Use tryLock() with timeout (from java.util.concurrent.locks.Lock)
- Reduce the scope of synchronized blocks
- Avoid nested locks if possible

Using a fixed lock order (Lock Ordering / Resource Hierarchy):

- Always acquire multiple locks in a predefined global order.
- This ensures that no circular waiting occurs, which is the main cause of deadlocks.

Example:

```
class Resource {
    String name;
    Resource(String name) { this.name = name; }
}

public class PreventDeadlock {
    public static void main(String[] args) {
        Resource r1 = new Resource("Resource1");
        Resource r2 = new Resource("Resource2");
```

```
        // Both threads lock r1 first, then r2
        Thread t1 = new Thread(() -> {
            synchronized (r1) {
                System.out.println("Thread 1 locked " +
r1.name);
                synchronized (r2) {
                    System.out.println("Thread 1 locked " +
r2.name);
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (r1) {
                System.out.println("Thread 2 locked " +
r1.name);
                synchronized (r2) {
                    System.out.println("Thread 2 locked " +
r2.name);
                }
            }
        });
```

```
        t1.start();
        t2.start();
    }
}
```

Result: No deadlock occurs because both threads acquire locks in the same order (r1 → r2).