1. Thread Interruption

1.1 What is Thread Interruption?

Thread interruption is a mechanism in Java that allows one thread to signal another thread to stop its current operation. This is particularly useful for terminating long-running or blocked threads gracefully without abruptly terminating them.

1.2 How to Interrupt a Thread

To interrupt a thread in Java, you use the `Thread.interrupt()` method. When this method is called on a thread, the thread's interrupted status is set to `true`. Threads should periodically check this status to determine if they should stop execution.

1.3 Handling Interruptions

Certain blocking methods like `Thread.sleep()`, `Object.wait()`, or `Thread.join()` throw an `InterruptedException` when a thread is interrupted. You can catch this exception to handle the interruption.

Example:

```
class MyTask implements Runnable {
    @Override
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                // Perform task
                System.out.println("Task is running...");
                Thread.sleep(1000); // Simulate a long-running
task
            }
        } catch (InterruptedException e) {
```

```
            System.out.println("Task was interrupted!");
            Thread.currentThread().interrupt(); // Preserve the
interrupt status
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyTask());
        thread.start();
        try {
            Thread.sleep(3000); // Let the thread run for a while
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread.interrupt(); // Interrupt the thread
    }
}
```

1.4 Best Practices

- **Cooperative Interruption:** Threads should regularly check their interrupted status and exit gracefully if interrupted.
- **Preserve Interrupt Status:** If you catch `InterruptedException`, you should re-interrupt the thread by calling `Thread.currentThread().interrupt()`.

---

2. Fork/Join Framework

2.1 Introduction

The Fork/Join Framework, introduced in Java 7, is a framework for parallel processing that allows developers to take full advantage of multi-core processors. It is designed to recursively split tasks into smaller subtasks, process them in parallel, and then combine the results.

2.2 Key Components

- **`ForkJoinPool`**: A specialized thread pool for running `ForkJoinTask` instances.
- **`ForkJoinTask`**: An abstract class representing a task that can be split into smaller tasks. Two main subclasses:
  - **`RecursiveTask<V>`**: Used when the task returns a result.
  - **`RecursiveAction`**: Used when the task does not return a result.

2.3 How It Works

1. **Forking:** A task is divided into smaller subtasks using the `fork()` method, which are then executed in parallel.
2. **Joining:** After the subtasks are completed, their results are combined using the `join()` method.

Example:

```
class FibonacciTask extends RecursiveTask<Integer> {
    private final int n;

    FibonacciTask(int n) {
        this.n = n;
    }

    @Override
    protected Integer compute() {
        if (n <= 1) {
            return n;
        }
        FibonacciTask f1 = new FibonacciTask(n - 1);
        f1.fork(); // Fork the first subtask
        FibonacciTask f2 = new FibonacciTask(n - 2);
        return f2.compute() + f1.join(); // Join the result of
the first subtask
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        FibonacciTask task = new FibonacciTask(10);
        int result = pool.invoke(task);
        System.out.println("Fibonacci result: " + result);
    }
}
```

2.4 Advantages

- **Efficiency:** The Fork/Join framework is optimized for work-stealing, where idle threads can "steal" tasks from busy threads, improving overall efficiency.
- **Scalability:** It scales well on multi-core processors, making it ideal for computationally intensive tasks.

---

3. Deadlock Prevention

3.1 What is a Deadlock?

A deadlock is a situation in which two or more threads are blocked forever, each waiting for the other to release a resource. In a deadlock scenario, no thread can proceed because each thread is holding a resource the other needs.

3.2 Conditions for Deadlock

For a deadlock to occur, the following four conditions must hold simultaneously:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode.
2. **Hold and Wait:** A thread holding at least one resource is waiting to acquire additional resources held by other threads.
3. **No Preemption:** Resources cannot be forcibly taken from a thread; they must be released voluntarily.

4. **Circular Wait:** A set of threads is waiting in a circular chain, where each thread is waiting for a resource held by the next thread in the chain.

3.3 Deadlock Prevention Strategies

1. **Avoid Circular Wait:** Impose an ordering on the acquisition of resources and ensure that all threads acquire resources in that order.
2. **Lock Timeout:** Use timed locks (e.g., `tryLock()` with a timeout) to avoid waiting indefinitely.
3. **Lock Hierarchy:** Design your system so that locks are always acquired in a specific order, preventing circular wait conditions.
4. **Conservative Locking:** Use a single lock to protect multiple resources when feasible, reducing the complexity of lock acquisition.
5. **Avoid Hold and Wait:** Ensure that threads request all the resources they need at once, rather than holding some and waiting for others.

Example:

```
class Account {
    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public synchronized void withdraw(double amount) {
        balance -= amount;
    }

    public static void transfer(Account from, Account to, double
amount) {
        synchronized (from) {
            synchronized (to) {
                from.withdraw(amount);
                to.deposit(amount);
            }
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Account account1 = new Account();
        Account account2 = new Account();

        // Avoiding Deadlock by acquiring locks in a consistent
order
        Thread t1 = new Thread(() -> Account.transfer(account1,
account2, 100));
        Thread t2 = new Thread(() -> Account.transfer(account2,
account1, 50));

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

3.4 Best Practices

- **Resource Ordering:** Always acquire resources in a consistent order.
- **Lock Timeouts:** Use timeouts to avoid indefinite waiting for resources.
- **Monitor Thread States:** Use tools like `jconsole` or `VisualVM` to monitor thread states and detect potential deadlocks.