Thread Concepts and Thread Pools in Java

1. Introduction to Threads

Threads allow concurrent execution of tasks in a Java program, enabling efficient use of CPU resources. A thread represents an independent path of execution within a program, making it possible to perform multiple tasks simultaneously.

1.1. Thread Creation

There are two main ways to create threads in Java:

1. **Implementing `Runnable`**: A class implements the `Runnable` interface and overrides the `run()` method.
2. **Extending `Thread`**: A class extends the `Thread` class and overrides the `run()` method.

Example: Creating a Thread using `Runnable`

```
class Transaction implements Runnable {
    private String transactionType;
    private double amount;

    public Transaction(String transactionType, double amount) {
        this.transactionType = transactionType;
        this.amount = amount;
    }

    @Override
    public void run() {
        // Simulate transaction processing
    }
}
```

Example: Creating a Thread by Extending `Thread`

```
class Teller extends Thread {
    private String transactionType;
    private double amount;

    public Teller(String transactionType, double amount) {
        this.transactionType = transactionType;
        this.amount = amount;
    }

    @Override
```

```
    public void run() {
        // Simulate transaction processing
    }
}
```

Key Takeaway: `Runnable` provides flexibility by allowing the class to extend another class, while extending `Thread` is more straightforward when the thread's behaviour is tightly coupled with the class logic.

1.2. Starting a Thread

After creating a thread, it needs to be started using the `start()` method, which triggers the `run()` method and moves the thread into the `RUNNABLE` state.

```
Teller teller1 = new Teller("Deposit", 1000);
teller1.start();  // Start the thread
```

Key Highlight: The `start()` method should always be used to begin a thread's execution, as calling `run()` directly will not create a new thread.

2. Thread Life Cycle

A thread in Java goes through several states during its life cycle:

1. **NEW**: The thread is created but not yet started.
2. **RUNNABLE**: The thread is ready to run.
3. **BLOCKED**: The thread is waiting to acquire a lock to enter a synchronised block/method.
4. **WAITING** or **TIMED_WAITING**: The thread is waiting for another thread's action or a specified time.
5. **TERMINATED**: The thread has finished executing.

Example: Transitioning through Thread States

```
System.out.println(getName() + " is in the NEW state");

public void run() {
    System.out.println(getName() + " is in the RUNNABLE state");
    Thread.sleep(2000);
    synchronised(this) {
        System.out.println(getName() + " is in the BLOCKED state");
    }
    System.out.println(getName() + " is in the TERMINATED state");
}
```

Key Highlight: Understanding the thread life cycle helps in managing thread states effectively, particularly when dealing with complex multithreaded applications.

3. Thread Synchronisation

Synchronisation is essential when multiple threads access shared resources, ensuring that only one thread can execute a critical section at a time. This prevents race conditions and data inconsistency.

Example: Synchronizing Access to a Shared Resource

```
public synchronised void updateBalance(String transactionType, double amount) {
    if (transactionType.equals("Withdrawal")) {
        totalBalance -= amount;
    } else if (transactionType.equals("Deposit")) {
        totalBalance += amount;
    }
    System.out.println("Updated balance: $" + totalBalance);
}
```

Key Highlight: The `synchronised` keyword ensures that only one thread can execute the `updateBalance` method at a time, preventing other threads from interfering with the current thread's operation.

4. Thread Pools

Thread pools are used to manage a group of reusable threads efficiently. Instead of creating and destroying threads for each task, a pool of threads is maintained, reducing the overhead and improving performance.

4.1. Creating a Thread Pool

A thread pool can be created using the `Executors` factory class, which provides various methods for creating different types of thread pools.

Example: Creating a Fixed Thread Pool

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

This creates a pool of 3 threads that can handle multiple transactions concurrently.

4.2. Submitting Tasks to the Thread Pool

Tasks can be submitted to the thread pool using the `submit()` method. The pool assigns tasks to available threads, and if all threads are busy, tasks wait in a queue until a thread becomes available.

```
tellerPool.submit(new Transaction("Deposit", 1000));
tellerPool.submit(new Transaction("Withdrawal", 500));
```

Key Highlight: Thread pools optimise resource usage by reusing threads for multiple tasks, leading to better performance and scalability in applications with numerous concurrent tasks.

4.3. Shutting Down the Thread Pool

After all tasks are submitted, the thread pool should be properly shut down to release resources.

```
tellerPool.shutdown();
```

Key Highlight: Always ensure to shut down the thread pool to prevent resource leaks and ensure that all submitted tasks are completed.

5. Conclusion

Understanding and implementing thread concepts like thread creation, life cycle management, synchronisation, and thread pools is crucial for developing efficient and concurrent applications in Java. By applying these concepts effectively, you can manage multiple tasks simultaneously, optimise performance, and ensure data consistency in complex, multithreaded environments.