Multithreading lab report

Amo Samuel

In this exercise, I explored some fundamental concepts in Java's concurrency model, particularly focusing on threads and thread pools. Here's a simple explanation of these concepts, along with how I implemented them in my code.

Threads

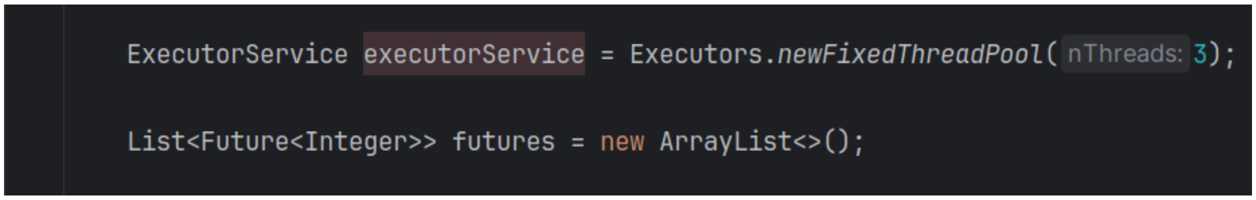
A thread in Java is essentially a separate path of execution in a program. When we run multiple threads, they can perform different tasks simultaneously. This is particularly useful when we want to perform operations that might take a while (like processing a large dataset or handling multiple user requests) without blocking the rest of the program.

In the code, each operation to withdraw money from the BankAccount is performed in its own thread. This allows it to process multiple withdrawal requests at the same time, rather than one after the other.

Thread Pools

A thread pool is a collection of pre-instantiated reusable threads. Instead of creating a new thread for each task (which can be resource-intensive), a thread pool manages a pool of worker threads. These threads can be reused to execute multiple tasks.

In the code, I used a fixed thread pool with three threads:



```
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 3);

List<Future<Integer>> futures = new ArrayList<>();
```

This means that at any given time, a maximum of three threads can execute tasks concurrently. If there are more tasks than available threads, the extra tasks have to wait until a thread becomes available.

Implementation Breakdown

Creating the Bank Account

The BankAccount class models a simple bank account with methods for depositing and withdrawing money. Both the deposit and withdrawal methods are synchronized to ensure that only one thread can modify the balance at a time, preventing issues like race conditions.

Creating Callable Tasks

I created a BankAccountOperations class that implements Callable<Integer>. The call() method in this class performs the withdrawal operation on the BankAccount and returns the withdrawn amount.

By using Callable, I can submit tasks that return a result (the amount withdrawn) and handle them in the main thread after the tasks complete.

Submitting Tasks to the Thread Pool

In the Main class, I created a BankAccount instance and submitted multiple BankAccountOperations tasks to the thread pool using executorService.submit().

Each task tries to withdraw a certain amount from the account. The results of these operations are captured in a list of Future<Integer> objects.

Processing the Results

After all tasks are submitted, I shut down the thread pool to prevent new tasks from being submitted.

I then iterated over the list of Future objects to get the results of each withdrawal operation. The total amount withdrawn is calculated, and the final balance in the bank account is displayed.

Conclusion

By using threads and thread pools, I was able to simulate a scenario where multiple users are trying to withdraw money from the same bank account simultaneously. The synchronized methods in the BankAccount class ensure that these operations are thread-safe, meaning the account balance is correctly managed despite the concurrent access. The thread pool helps manage the execution of these tasks efficiently, ensuring that system resources are used optimally.