

Practice 7

Deadline: 2 weeks from now. Should be checked onsite (during labs).

Task 1: Account Deposit

We provided `Account.java`, `DepositThread.java` and `Test.java` on Blackboard site. Please read the code to understand its semantics. Then execute `Test.java` and observe the results. Is this program thread safe? If not, how should we fix it?

In this practice, we'll be fixing `Account.java` and `DepositThread.java` so that the program works as expected. Specifically, you should use the following two approaches:

- 1. Using the `synchronized` keyword
- 2. Using `ReentrantLock`

Please show us your fixes using these two approaches and demonstrate that your program now outputs the correct result, i.e., `balance` should be 1000.

Finally, we want to explore which approach is faster. Modify `Test.java` to use the `JMH` library we introduced in lab 2 to profile and compare these two approaches on the same code of the `main` method. A sample profiling run might look like this:

Benchmark	Mode	Cnt	Score	Error	Units
Test.testAccountLock	avgt	3	1897.608	± 228.313	ms/op
Test.testAccountSync	avgt	3	1860.767	± 272.979	ms/op

Task 2: Producer-Consumer Problem

Producer and Consumer are two separate processes. Both processes share a common buffer or queue. The producer continuously produces certain data and pushes it onto the buffer, whereas the consumer consumes those data from the buffer. This problem has several complexities to deal with:

- Both producer and consumer may try to update the queue at the same time. This could lead to data loss or inconsistencies.
- Producers might be slower than consumers. In such cases, the consumer would process elements faster and wait.
- In some cases, the consumer can be slower than a producer. This situation leads to a queue overflow issue.
- In real scenarios, we may have multiple producers, multiple consumers, or both. This may cause the same message to be processed by different consumers.

To implement the producer-consumer pattern, we need to solve the following problems:

- Synchronization on a `queue` while adding and removing data
- On queue empty, the consumer has to wait until the producer adds new data to the queue
- When the queue is full, the producer has to wait until the consumer consumes data

We refer to such a queue `BlockQueue`. Complete the TODOs in `MyBlockingQueue.java`, which uses `Queue` from the Java Collections Framework together with the Locking mechanism, to implement the exact behavior of a `BlockQueue`. Executing the `main` method and observe the results.

```
.....
Thread:Thread-5, Produced:19, Queue:[19, 19, 19, 19, 19, 19, 19, 19]
Thread:Thread-17, Consumed:19, Queue:[19, 19, 19, 19, 19, 19, 19]
Thread:Thread-12, Consumed:19, Queue:[19, 19, 19, 19, 19, 19]
Thread:Thread-18, Consumed:19, Queue:[19, 19, 19, 19, 19]
Thread:Thread-14, Consumed:19, Queue:[19, 19, 19, 19]
Thread:Thread-15, Consumed:19, Queue:[19, 19, 19]
Thread:Thread-13, Consumed:19, Queue:[19, 19]
Thread:Thread-16, Consumed:19, Queue:[19]
Thread:Thread-19, Consumed:19, Queue:[ ]
```

Try to change the values of the following variables.

```
int CAPACITY = 200;
int PRODUCER_WORK = 20;
int PRODUCER_CNT = 10;
int PRODUCER_OFF = 10;
int CONSUMER_WORK = 20;
int CONSUMER_CNT = 10;
int CONSUMER_OFF = 10;
```

Example:

- `PRODUCER_CNT = 10->100`
- `CONSUMER_CNT = 10->100`
- `PRODUCER_OFF = 10->1000`
- `PRODUCER_WORK = 20->21`

What will happen and why? **Please explain the results.**

Evaluation

The practice will be checked by teachers or SAs. What will be tested:

1. That you understand every line of your own code, not just copy from somewhere
2. That your program compiles correctly (javac)
3. Correctness of the program logic
4. That the result is obtained in a reasonable time

Late submissions after the deadline will incur a 20% penalty, meaning that you can only get 80% of this practice's score.