Concurrency-3 Introduction to Synchronisation, Mutex, Synchronized, Atomic Data-types

---

Agenda

Synchronisation Problem

Adder Subtracter Recap

The adder and subtractor problem is a sample problem that is used to demonstrate the need for synchronisation in a system. The problem is as follows:

- Create a count class that has a count variable.
- Create two different classes Adder and Subtractor.
- Accept a count object in the constructor of both the classes.
- In Adder, iterate from 1 to 100 and increment the count variable by 1 on each iteration.
- In Subtractor, iterate from 1 to 100 and decrement the count variable by 1 on each iteration.
- Print the final value of the count variable.

What would the ideal value of the count variable be? What is the actual value of the count variable? Try to add some delay in the Adder and Subtractor classes using inspiration from the code below. What is the value of the count variable now?

Adder.java

```java
public class Adder implements Runnable {
    private Count count;

    public Adder(Count count) {
        this.count = count;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            count.increment();
        }
    }
}
```

Subtracter.java

```java
public class Subtractor implements Runnable {
    private Count count;

    public Subtractor(Count count) {
        this.count = count;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            count.decrement();
        }
    }
}
```

Runner.java

```java
public class Runner {
    public static void main(String[] args) {
        Count count = new Count();
        Adder adder = new Adder(count);
        Subtractor subtractor = new Subtractor(count);

        Thread adderThread = new Thread(adder);
        Thread subtractorThread = new Thread(subtractor);

        adderThread.start();
        subtractorThread.start();

        adderThread.join();
        subtractorThread.join();

        System.out.println(count.getCount());
    }
}
```

Synchronisation Problem

In multithreaded environments, synchronization problems can arise due to concurrent execution of multiple threads, leading to unpredictable and undesirable behavior. There are certain conditions that can lead to synchronization problems. These Conditions are:

1. Critical Section

- A critical section is a part of the code that must be executed by only one thread at a time to avoid data inconsistency or corruption.
- If multiple threads access and modify shared data simultaneously within a critical section, it can lead to unpredictable results.
- Ensuring mutual exclusion by using synchronization mechanisms like locks or semaphores helps prevent multiple threads from entering the critical section simultaneously. Race Condition:

2. Race Conditions

- A race condition occurs when the final outcome of a program depends on the relative timing of events, such as the order in which threads are scheduled to run.
- In a race condition, the correctness of the program depends on the timing of the thread executions, and different outcomes may occur depending on the interleaving of thread execution.
- Proper synchronization mechanisms, like locks or atomic operations, are needed to prevent race conditions by enforcing a specific order of execution for critical sections. Preemption:

3. Preemption

- Preemption refers to the interrupting of a currently executing thread to start or resume the execution of another thread.
- In multithreaded environments, preemption can lead to issues if not handled carefully. For example, a thread might be preempted while in the middle of updating shared data, leading to inconsistent or corrupted state.
- To avoid issues related to preemption, critical sections should be protected using mechanisms like locks or disabling interrupts temporarily to ensure that a thread completes its operation without being interrupted.

To address these synchronization problems, various synchronization mechanisms are employed, such as locks, semaphores, and atomic operations. These tools help ensure that only one thread can access critical sections at a time, preventing race conditions and mitigating the impact of preemption on shared data. Additionally, proper design practices, like minimizing the use of shared mutable data and using thread-safe data structures, can contribute to reducing synchronization issues in multithreaded environments.

Properties of a Good Synchronization Solution

1. **Mutual Exclusion:**

   - *Definition:* Only one thread should be allowed to execute its critical section at any given time. Suppose there are three threads, and they are waiting to enter the critical sections of the Adder, Subtractor, and Multiplier. But a blocker should be there to allow only one thread in a critical section at a time.
   - *Importance:* Ensures that conflicting operations on shared resources do not occur simultaneously, preventing data corruption or inconsistency.

2. **Progress:**
   - *Definition:* The overall system should keep moving and making progress. It should not stop at any stage and be waiting for a long period. If no thread is in its critical section and some threads are waiting to enter the critical section, then the selection of the next thread to enter the critical section should be definite.

- *Importance:* Guarantees that the system makes progress and avoids deadlock situations where threads are unable to proceed.

3. **Bounded Waiting:**
   - *Definition:* There exists a limit on the number of times other threads are allowed to enter their critical sections after a thread has requested entry into its critical section and before that request is granted. No thread should be waiting infinitely. There should be a bound on how long they have to wait before they are allowed to enter the critical section.
   - *Importance:* Prevents the problem of starvation, where a thread is repeatedly delayed in entering its critical section by other threads.

4. **No Deadlock:**
   - *Definition:* A deadlock is a state where two or more threads are blocked forever, each waiting for the other to release a resource.
   - *Importance:* A good synchronization solution should avoid deadlocks, as they can lead to a complete system halt and result in unresponsive behavior.
5. **Efficiency:**
   - *Definition:* The synchronization solution should introduce minimal overhead and allow non-conflicting threads to execute concurrently.
   - *Importance:* Ensures that the system performs well and doesn't suffer from unnecessary delays or resource contention.
6. **Adaptability:**
   - *Definition:* The synchronization solution should be adaptable to different system configurations and workloads.
   - *Importance:* Facilitates the use of the synchronization mechanism in a variety of scenarios without requiring significant modifications.
7. **Low Busy-Waiting:**

- *Definition:* Minimizes the use of busy-waiting (spinning in a loop while waiting for a condition to be satisfied) to conserve CPU resources. When a thread has to continuously check if they can now enter the critical section. Checking if a thread can enter the critical section is not a productive use of time. The ideal solution should have some kind of notification system. For example if you have to check if a person is available or not: In way 1, you go and knock on the person’s door every 2 minutes to check if they are free. This is busy waiting In way 2, you go and tell the person that I am here. Please let me know when you are free. This is called a notification. This provides better usage of the time.
- *Importance:* Reduces unnecessary CPU consumption, making the system more efficient and avoiding the negative impact of busy-waiting on power consumption.

8. **Fairness:**
    - *Definition:* All threads should have a fair chance to enter their critical sections. No thread should be unfairly delayed or granted preferential access.
    - *Importance:* Ensures that the synchronization solution treats all threads fairly, preventing situations where some threads consistently get better access to shared resources.
9. **Scalability:**
    - *Definition:* The synchronization solution should scale well with an increasing number of threads and resources.
    - *Importance:* Allows the system to efficiently handle a growing number of threads without a significant degradation in performance.
10. **Portability:**
    - *Definition:* The synchronization solution should be portable across different platforms and operating systems.
    - *Importance:* Enables the synchronization mechanism to be used in diverse computing environments without requiring extensive modifications.

Solutions to Synchronisation Problem

1. Mutex Lock

Mutex means Mutual Exclusion. Mutex Lock is a lock that enables mutual exclusion. Mutex locks are a way to solve the synchronisation problem. Mutex locks are a way to ensure that only one thread can access a critical section at a time. Mutex locks are also known as mutual exclusion locks.

A thread can only access the critical section if it has the lock. If a thread does not have the lock, it cannot access the critical section. If a thread has the lock, it can access the critical section. If a thread has the lock, it can release the lock and allow another thread to access the critical section.

Suppose if we take the example of Adder and Subtractor here, Adder:

```
print('Hi')
x <- read(count)
x = x + 1
print('Bye')
```

Subtractor:

```
print('Hello')
x <- read(count)
x = x - 1
print('Bye')
```

The above adder-substracter if executed concurrently, can lead to wrong results due to interleaving of instructions. So, MUTEX SAYS:

- A thread must take a lock before it enters its critical section.
- They must remove the lock as soon as they leave the critical section.

Think of a room with a lock. Only one person can enter the room at a time. If a person has the key, they can enter the room. If a person does not have the key, they cannot enter the room. If a person has the key, they can leave the room and give the key to another person. This is the same as a mutex lock.

- So, A thread(person) must take a lock(have a key) before they enter their critical section(Room).
- They must remove the lock(key) as soon as they leave the (Room)critical section.
- By default, the program can not enforce synchronization. The developer has to do it.

So what do we have to do? Before entering a critical section, lock the thread and, at exit, unlock it. Example:

Adder:

```
print('Hi')
lock.lock()
x <- read(count)
x = x + 1
count = x
lock.unlock()
print('Bye')
```

Subtractor:

```
print('Hello')
lock.lock()
x <- read(count)
x = x - 1
count = x
lock.unlock()
print('Bye')
```

Now let’s discuss about the Properties of lock:

- Only one thread can unlock a thread at one time; other threads have to wait till that thread unlocks.
- Lock will automatically notify the second thread to run when the first one exits.
- It has no busy waiting
- It has mutual exclusion
- Bounded waiting
- The system is having overall progress

Code for reference:

```
Client class (main)

public static void main(String[] args) {
    Count count = new Count;
    Lock lock = new ReentranttLock();
    Adder adder = new Adder (count, lock);
    Subtractor subtractor = new Subtractor (count, lock);
    Thread t1 = new Thread (adder);
    Thread t2 = new Thread (subtractor);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    system.out.println(count.value);
```

Now, let's change the constructor of adder and subtractor Adder:

```
package addersubtractor;
    public class Adder implements Runnable {
        private Count count;
        private Lock lock;
        public Adder (Count count, Lock lock) {
                this.count = count;
                this.lock = lock
              }
         @Override
        public void run() {
            for (int i = 1 ; i <= 100; ++ 1) {
                lock.lock()
                count.value += i;
                lock.unlock();
            }
}
```

Subtractor:

```
package addersubtractor;
    public class Subtractor implements Runnable  {
         private Count count;
         private Lock lock;
        public Subtractor (Count count, Lock lock) {
            this.count = count;
            this.lock = lock;
            }
        @Override
        public void run() {
            for (int i = 1 ; i <= 100; ++ 1) {
                lock.lock();
                count.value -= i;
                lock.unlock()
            }
}
```

Now, when we run this, we will always get 0 as the answer, which was not the case earlier.

Properties of a mutex lock

- A thread can only access the critical section if it has the lock.
- Only one thread can have the lock at a time.
- Other threads cannot access the critical section if a thread has the lock and thus have to wait.
- Lock will automatically be released when the thread exits the critical section.

2. Synchronised keyword

The synchronized keyword is a way to solve the synchronisation problem. The synchronized keyword is a way to ensure that only one thread can access a critical section at a time.

A synchronized method or block can only be accessed by one thread at a time. If a thread is accessing a synchronized method or block, other threads cannot access the synchronized method or block. If a thread is accessing a synchronized method or block, other threads have to wait until the thread exits the synchronized method or block.

Following is an example of a synchronized method:

```
public class Count {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

In the above example, the increment() and decrement() methods are synchronized. This means that only one thread can access the increment() and decrement() methods at a time. If a thread is accessing the increment() method, other threads cannot access the increment() method. If a thread is accessing the decrement() method, other threads cannot access the decrement() method. If a thread is accessing the increment() method, other threads have to wait until the thread exits the increment() method. If a thread is accessing the decrement() method, other threads have to wait until the thread exits the decrement() method.

Similarly, the **synchronized keyword** can be used to synchronize a block of code. Following is an example of a synchronized block:

```
public class Count {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public void decrement() {
        synchronized (this) {
            count--;
        }
    }

    public int getCount() {
        return count;
    }
}
```

If you declare a method as synchronized, only one thread will be able to access any synchronized method in the class. This is because the synchronized keyword is associated with the object.

Coding Problem 1 - Thread Safe Counter (Homework)

// Implement a class that represents a counter and is accessed by multiple threads. // Ensure that the counter is updated in a thread-safe manner without using the synchronized keyword.

```
public class ThreadSafeCounter {
    private int count = 0;

    // TODO: Implement a thread-safe method to increment the counter.

    public static void main(String[] args) {
        // TODO: Create multiple threads that concurrently increment the counter.
        // Ensure that the counter is updated in a thread-safe manner without using
the synchronized keyword.
    }
}
```

Coding Problem 2 - Reentrantlock Basics (Homework)

Implement a program that uses ReentrantLock to achieve thread safety.

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {
    private int value = 0;
    private final Lock lock = new ReentrantLock();

    // TODO: Implement a method to update the value using ReentrantLock.

    public static void main(String[] args) {
        // TODO: Create multiple threads that concurrently update the value using
ReentrantLock.
        // Ensure that the value is updated in a thread-safe manner.
    }
}
```

Additonal Topics

1. Atomic Datatypes in java

In Java, the java.util.concurrent.atomic package provides a set of classes that support atomic operations on variables. These classes are designed to be thread-safe and eliminate the need for explicit synchronization in certain scenarios. One commonly used class is AtomicInteger. In this tutorial, we'll explore AtomicInteger and provide a simple code example.

Atomic Integer Basics The AtomicInteger class provides atomic operations on an integer variable. These operations are performed in a way that ensures atomicity, making them thread-safe without the need for explicit synchronization.

Example Usage:

```java
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerExample {

    public static void main(String[] args) {
        // Create an AtomicInteger with an initial value
        AtomicInteger atomicInteger = new AtomicInteger(0);

        // Perform atomic increment
        int newValue = atomicInteger.incrementAndGet();
        System.out.println("Incremented Value: " + newValue);

        // Perform atomic decrement
        newValue = atomicInteger.decrementAndGet();
        System.out.println("Decremented Value: " + newValue);

        // Perform atomic add
        int addValue = 5;
        newValue = atomicInteger.addAndGet(addValue);
        System.out.println("After Adding " + addValue + ": " + newValue);

        // Perform compare-and-set operation
        int expectedValue = 5;
        int updateValue = 10;
        boolean success = atomicInteger.compareAndSet(expectedValue, updateValue);
        if (success) {
            System.out.println("Value updated successfully. New Value: " +
atomicInteger.get());
        } else {
            System.out.println("Value was not updated. Current Value: " +
atomicInteger.get());
        }
    }
}
```

Benefits of Atomic Datatypes:

- Thread Safety: Operations on AtomicInteger are atomic, eliminating the need for explicit synchronization.
- Performance: Atomic operations are more efficient than using locks for simple operations on shared variables.
- Simplicity: Simplifies the development of thread-safe code in scenarios where simple atomic operations suffice.

Lets use Atomic Integers in Adder-Subtracter Example. **InventoryCounter.java**

```
public class InventorCounter {
    AtomicInteger counter = new AtomicInteger(0);
}
```

Adder.java

```
public class Adder implements Runnable{
    private InventorCounter ic;

    Adder(InventorCounter ic){
        this.ic = ic;
    }

    @Override
    public void run() {
        for(int i=0;i<=10000;i++){
            ic.counter.addAndGet(1);
        }
    }
}
```

Subtracter.java

```
public class Subtracter implements Runnable{
    private InventorCounter ic;

    Subtracter(InventorCounter ic){
        this.ic = ic;
    }

    @Override
    public void run() {
        for(int i=0;i<=10000;i++){
            ic.counter.addAndGet(-1);
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) throws InterruptedException {

        InventorCounter ic = new InventorCounter();
        Thread t1 = new Thread(new Adder(ic));
        Thread t2 = new Thread(new Subtracter(ic));
        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(ic.counter.get());
    }
}
```

In summary, the AtomicInteger class in Java provides a convenient and efficient way to perform atomic operations on integer variables, making it a valuable tool for concurrent programming. Similar classes, such as AtomicLong and AtomicBoolean, exist for other primitive types.

2. Volatile Keyword

Video Tutorial on Volatile

Volatile Keyword solves for problems like Memory Inconsistency Errors & Data Races. Let's understand this in more detail.

The Operating system may read from heap variables, and make a copy of the value in each thread's own storage. Each threads has its own small and fast memory storage, that holds its own copy of shared resource's value.

Once thread can modify a shared variable, but this change might not be immediately reflected or visible. Instead it is first update in thread's local cache. The operating system may not flush the first thread's changes to the heap, until the thread has finished executing, causing memory inconsistency errors.

Lets see it through this code in action: **SharedResourced.java**

```
public class SharedResource {
    volatile private boolean flag;
    SharedResource(){
        flag = false;
    }
    //Two More Methods
    public void toggleFlag(){
        flag = !flag;
    }
    public boolean getFlag(){
        return flag;
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();
        System.out.println("Shared Resource Created, Flag Value " +
sharedResource.getFlag());

        Thread A = new Thread(()->{
            //After 2S, toggle the value
            try{
                Thread.sleep(2000);
            }
            catch(InterruptedException e){
                e.printStackTrace();
            }
            sharedResource.toggleFlag();
            System.out.println("Thread A is finished, Flag is
"+sharedResource.getFlag());
        });

        Thread B = new Thread(()->{
            while(!sharedResource.getFlag()){
                //...busy-wait...
                // System.out.println("Inside Loop " + sharedResource.getFlag());

            }
            System.out.println("In Thread B, Flag is "+sharedResource.getFlag());
        });

        A.start();
        B.start();


    }
}
```

Solution - Volatile Keyword

- The volatile keyword is used as modifier for class variables.
- It's an indicator that this variable's value may be changed by multiple threads.

- This modifier ensures that the variable is always read from, and written to the main memory, rather than from any thread-specific cache.
- This provides memory consistency for this variables value across threads. Volatile doesn't gurantee atomicicty.

However, volatile does not provide atomicity or synchronization, so additional synchronization mechanisms should be used in conjunction with it when necessary.

When to use volatile

- When a variable is used to track the state of a shared resource, such as counter or a flag.
- When a varaible is used to communicate between threads.

When not use volatile

- When the variable is used by single thread.

- When a variable is used to store a large amount of data.

3. Concurrent Data Structures

There are data structures designed in Collections Framework which support Concurrency but we will limit our discussions to one of the widely asked data structures - Concurrent Hashmap. Java Collections provides various data structures for working with **key-value pairs**. The commonly used ones are -

- **Hashmap** (Non-Synchronised, Not Thread Safe)
  - discuss the Synchronized Hashmap method
- **Hashtable** (Synchronised, Thread Safe)
  - locking over entire table
- **Concurrent Hashmap** (Synchronised, Thread Safe, Higher Level of Concurrency, Faster)
  - locking at bucket level, fine grained locking

Hashmap and Synchronised Hashmap Method Synchronization is the process of establishing coordination and ensuring proper communication between two or more activities. Since a HashMap is not synchronized which may cause data inconsistency, therefore, we need to synchronize it. The in-built method ‘Collections.synchronizedMap()’ is a more convenient way of performing this task.

A synchronized map is a map that can be safely accessed by multiple threads without causing concurrency issues. On the other hand, a Hash Map is not synchronized which means when we implement it in a multi-threading environment, multiple threads can access and modify it at the same time without any coordination. This can lead to data inconsistency and unexpected behavior of elements. It may also affect the results of an operation.

Therefore, we need to synchronize the access to the elements of Hash Map using ‘synchronizedMap()’. This method creates a wrapper around the original HashMap and locks it whenever a thread tries to access or modify it.

```
Collections.synchronizedMap(instanceOfHashMap);
```

The `synchronizedMap()` is a static method of the Collections class that takes an instance of HashMap collection as a parameter and returns a synchronized Map from it. However,it is important to note that only the map itself is synchronized, not its views such as keyset and entrySet. Therefore, if we want to iterate over the synchronized map, we need to use a synchronized block or a lock to ensure exclusive access.

```
import java.util.*;
public class Maps {
   public static void main(String[] args) {
      HashMap<String, Integer> cart = new HashMap<>();
      // Adding elements in the cart map
      cart.put("Butter", 5);
      cart.put("Milk", 10);
      cart.put("Rice", 20);
      cart.put("Bread", 2);
      cart.put("Peanut", 2);
      // printing synchronized map from HashMap
      Map mapSynched = Collections.synchronizedMap(cart);
      System.out.println("Synchronized Map from HashMap: " + mapSynched);
   }
}
```

Hashtable vs Concurrent Hashmap HashMap is generally suitable for single threaded applications and is faster than Hashtable, however in multithreading environments we have you use **Hashtable** or **Concurrent Hashmap**. So let us talk about them.

While both Hashtable and Concurrent Hashmap collections offer the advantage of thread safety, their underlying architectures and capabilities significantly differ. Whether we’re building a legacy system or working on modern, microservices-based cloud applications, understanding these nuances is critical for making the right choice.

Let's see the differences between Hashtable and ConcurrentHashMap, delving into their performance metrics, synchronization features, and various other aspects to help us make an informed decision.

1. Hashtable Hashtable is one of the oldest collection classes in Java and has been present since JDK 1.0. It provides key-value storage and retrieval APIs:

```
Hashtable<String, String> hashtable = new Hashtable<>();
hashtable.put("Key1", "1");
hashtable.put("Key2", "2");
hashtable.putIfAbsent("Key3", "3");
String value = hashtable.get("Key2");
```

The primary selling point of Hashtable is thread safety, which is achieved through method-level synchronization.

Methods like put(), putIfAbsent(), get(), and remove() are synchronized. Only one thread can execute any of these methods at a given time on a Hashtable instance, ensuring data consistency.

2. Concurrent Hashmap ConcurrentHashMap is a more modern alternative, introduced with the Java Collections Framework as part of Java 5.

Both Hashtable and ConcurrentHashMap implement the Map interface, which accounts for the similarity in method signatures:

```
ConcurrentHashMap<String, String> concurrentHashMap = new ConcurrentHashMap<>();
concurrentHashMap.put("Key1", "1");
concurrentHashMap.put("Key2", "2");
concurrentHashMap.putIfAbsent("Key3", "3");
String value = concurrentHashMap.get("Key2");
```

ConcurrentHashMap, on the other hand, provides thread safety with a higher level of concurrency. It allows multiple threads to read and perform limited writes simultaneously **without locking the entire data structure**. This is especially useful in applications that have more read operations than write operations.

Performance Comparison Hashtable locks the entire table during a write operation, thereby preventing other reads or writes. This could be a bottleneck in a high-concurrency environment.

ConcurrentHashMap, however, allows concurrent reads and limited concurrent writes, making it more scalable and often faster in practice.

Coding Projects on Synchronisatoin

Coding Problem 3 - Ticket Booking System

Consider an online reservation system for booking tickets to various events. The system needs to handle concurrent requests from multiple users trying to reserve seats. To ensure thread safety and prevent race conditions, a Reentrant Lock can be employed. Requirements:

- The reservation system manages the availability of seats for different events.
- Multiple users can attempt to reserve seats concurrently.
- A user should be able to reserve multiple seats for the same event.
- The system should prevent overbooking and ensure the integrity of seat reservations.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class ReservationSystem {
    private int availableSeats;
    private final Lock lock = new ReentrantLock();

    public ReservationSystem(int totalSeats) {
        this.availableSeats = totalSeats;
    }

    public void reserveSeats(String user, int numSeats) {
        lock.lock();
        try {
            if (numSeats > 0 && numSeats <= availableSeats) {
                // Simulate the reservation process
                System.out.println(user + " is reserving " + numSeats + " seats.");
                // Update available seats
                availableSeats -= numSeats;

                // Simulate the ticket issuance
                System.out.println(user + " reserved seats successfully.");
            } else {
                System.out.println(user + " could not reserve seats. Not enough 
available seats.");
            }
        } finally {
            lock.unlock();
        }
    }

    public int getAvailableSeats() {
        return availableSeats;
    }
}

public class OnlineReservationSystem {
    public static void main(String[] args) {
        ReservationSystem reservationSystem = new ReservationSystem(50);

        // Simulate multiple users trying to reserve seats concurrently
        Thread user1 = new Thread(() -> reservationSystem.reserveSeats("User1", 5));
        Thread user2 = new Thread(() -> reservationSystem.reserveSeats("User2", 10));
        Thread user3 = new Thread(() -> reservationSystem.reserveSeats("User3", 8));

        user1.start();
        user2.start();
        user3.start();

        try {
            user1.join();
            user2.join();
            user3.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("Remaining available seats: " + 
reservationSystem.getAvailableSeats());
    }
}
```

In this example, the ReservationSystem class utilizes a Reentrant Lock (lock) to ensure that the reservation process is thread-safe. The reserveSeats method is enclosed in a try-finally block to ensure that the lock is always released, even if an exception occurs. This real-world problem demonstrates how Reentrant Locks can be used to synchronize access to shared resources in a multi-threaded environment, ensuring data consistency and preventing race conditions in a scenario like an online reservation system.

Coding Problem 4 - Thread-safe Bank Transactions

Problem Statement: You are tasked with implementing a simple bank system that supports concurrent transactions. The bank has multiple accounts, and customers can deposit and withdraw money from their accounts concurrently. Implement a program that ensures the integrity of bank transactions by using threads. **Requirements:**

- Each account has a unique account number and an initial balance.
- Customers can concurrently deposit and withdraw money from their accounts.
- The bank should ensure that the account balance remains consistent and does not go below zero during concurrent transactions.
- Use threads to simulate multiple customers performing transactions simultaneously.

Solution

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BankAccount {
    private final int accountNumber;
    private int balance;
    private final Lock lock = new ReentrantLock();


    public BankAccount(int accountNumber, int initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }


    public int getAccountNumber() {
        return accountNumber;
    }


    public int getBalance() {
        return balance;
    }


    public void deposit(int amount) {
        lock.lock();
        try {
            balance += amount;
            System.out.println("Deposited $" + amount + " to account " + accountNumber
+ ". New balance: $" + balance);
        } finally {
            lock.unlock();
        }
    }
```

```java
    public void withdraw(int amount) {
        lock.lock();
        try {
            if (amount <= balance) {
                balance -= amount;
                System.out.println("Withdrawn $" + amount + " from account " +
accountNumber + ". New balance: $" + balance);
            } else {
                System.out.println("Insufficient funds for withdrawal from account " +
accountNumber);
            }
        } finally {
            lock.unlock();
        }
    }
}


class BankTransaction implements Runnable {
    private final BankAccount account;
    private final int transactionAmount;


    public BankTransaction(BankAccount account, int transactionAmount) {
        this.account = account;
        this.transactionAmount = transactionAmount;
    }


    @Override
    public void run() {
        // Simulate a bank transaction (deposit or withdrawal)
        if (transactionAmount >= 0) {
            account.deposit(transactionAmount);
        } else {
            account.withdraw(Math.abs(transactionAmount));
        }
    }
}


public class BankSimulation {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount(101, 1000);
        BankAccount account2 = new BankAccount(102, 1500);


        // Simulate concurrent bank transactions using threads
        Thread thread1 = new Thread(new BankTransaction(account1, 200));
        Thread thread2 = new Thread(new BankTransaction(account1, -300));
        Thread thread3 = new Thread(new BankTransaction(account2, 500));


        thread1.start();
        thread2.start();
        thread3.start();


        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }


        // Display final account balances
        System.out.println("Final balance for account " + account1.getAccountNumber()
+ ": Rs" + account1.getBalance());
        System.out.println("Final balance for account " + account2.getAccountNumber()
+ ": Rs" + account2.getBalance());
    }
}
```

In this example, BankAccount represents a bank account with deposit and withdraw methods protected by a ReentrantLock. The BankTransaction class simulates a bank transaction (deposit or withdrawal), and the BankSimulation class demonstrates how threads can be used to perform concurrent transactions on multiple accounts. The use of locks ensures the thread safety of the bank transactions

Additonal Reading

- Reentrant Locks
- Concurrent DataStructures
- Fairness of Re-entrant Locks

-- End --