C.NAKUL SRIRAJ
2300031693
SEC 13

2300030279
JEENEPALLY ADISESHU
SEC-13

Week 9. Synchronization and Lock

1) FizzBuzz with Multithreading You are required to implement a class named FizzBuzz that facilitates the printing of a series based on specific divisibility rules in a multi-threaded environment. This class will be used in conjunction with four separate threads, each responsible for printing a different type of output according to the rules outlined below. Requirements:

1. Class Definition: o FizzBuzz(int n): The constructor initializes the FizzBuzz object with an integer n, which represents the total number of elements in the sequence that should be printed (from 1 to n).

2. Output Functions: o void fizz(Runnable printFizz): This method should be called by a thread to print the word "fizz". o void buzz(Runnable printBuzz): This method should be called by a thread to print the word "buzz". o void fizzbuzz(Runnable printFizzBuzz): This method should be called by a thread to print the word "fizzbuzz". o void number(Runnable printNumber): This method should be called by a thread to print the current integer.

3. Output Rules: For each integer iii (1-indexed) in the range from 1 to n: o Print "fizzbuzz" if iii is divisible by both 3 and 5. o Print "fizz" if iii is divisible by 3 but not by 5. o Print "buzz" if iii is divisible by 5 but not by 3. o Print iii itself if it is not divisible by either 3 or 5.

4. Thread Behavior: o You will have four threads: ♣ Thread A: Calls fizz(). ♣ Thread B: Calls buzz(). ♣ Thread C: Calls fizzbuzz(). ♣ Thread D: Calls number(). o These threads should operate in a synchronized manner to ensure the correct output sequence is maintained. Implementation Details: • Each thread should wait for its turn to print its respective output based on the defined rules. • You need to manage the coordination between the threads to ensure that they output in the correct order according to the rules above. • Use appropriate synchronization techniques (like wait() and notify()) to achieve this.

Code:

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


public class FizzBuzz {
    private int n;
    private AtomicInteger current = new AtomicInteger(1);
```

```
public FizzBuzz(int n) {
    this.n = n;
}



public void fizz(Runnable printFizz) throws InterruptedException {
    while (true) {
        synchronized (this) {
            if (current.get() > n) {
                return;
            }
            if (current.get() % 3 == 0 && current.get() % 5 != 0) {
                printFizz.run();
                current.incrementAndGet();
                notifyAll();
            } else {
                wait();
            }
        }
    }
}
public void buzz(Runnable printBuzz) throws InterruptedException {
    while (true) {
        synchronized (this) {
            if (current.get() > n) {
                return;
            }
```

```
                if (current.get() % 5 == 0 && current.get() % 3 != 0) {
                    printBuzz.run();
                    current.incrementAndGet();
                    notifyAll();
                } else {
                    wait();
                }
            }
        }
    }


    public void fizzbuzz(Runnable printFizzBuzz) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (current.get() > n) {
                    return;
                }
                if (current.get() % 3 == 0 && current.get() % 5 == 0) {
                    printFizzBuzz.run();
                    current.incrementAndGet();
                    notifyAll();
                } else {
                    wait();
                }
            }
        }
    }



    public void number(Runnable printNumber) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (current.get() > n) {
                    return;
                }
                if (current.get() % 3 != 0 && current.get() % 5 != 0) {
                    printNumber.run();
                    current.incrementAndGet();
                    notifyAll();
                } else {
                    wait();
                }
            }
        }
    }

    public static void main(String[] args) {
        FizzBuzz fizzBuzz = new FizzBuzz(15);

        Runnable printFizz = () -> System.out.print("fizz ");
        Runnable printBuzz = () -> System.out.print("buzz ");
        Runnable printFizzBuzz = () -> System.out.print("fizzbuzz ");
        Runnable printNumber = () -> System.out.print(fizzBuzz.current.get() + " ");
```

```
Thread threadA = new Thread(() -> {
    try {
        fizzBuzz.fizz(printFizz);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

Thread threadB = new Thread(() -> {
    try {
        fizzBuzz.buzz(printBuzz);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

Thread threadC = new Thread(() -> {
    try {
        fizzBuzz.fizzbuzz(printFizzBuzz);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

Thread threadD = new Thread(() -> {
    try {
```

```
                fizzBuzz.number(printNumber);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        threadA.start();
        threadB.start();
        threadC.start();
        threadD.start();
    }
}
```

2) Bank Account with Synchronized Methods Implement a simple banking system where multiple threads can deposit and withdraw money from a shared bank account. Description: • Create a BankAccount class with synchronized methods deposit() and withdraw(). • Use these methods to ensure that money is not double-withdrawn when two threads try to withdraw simultaneously. • Simulate multiple threads attempting to deposit and withdraw money concurrently. Key Concepts: • Use of synchronized keyword to ensure thread safety. • Demonstrate thread safety by observing the balance before and after concurrent operations.

Code:

```
class BankAccount {
    private int balance;

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

    public synchronized void deposit(int amount) {
```

C.NAKUL SRIRAJ
2300031693
SEC 13

```
        balance += amount;
        System.out.println(Thread.currentThread().getName() + " deposited: " + amount
+ ", Current balance: " + balance);
    }


    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew: " +
amount + ", Remaining balance: " + balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " tried to withdraw: "
+ amount + " but insufficient balance. Current balance: " + balance);
        }
    }
    public synchronized int getBalance() {
        return balance;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        Thread t1 = new Thread(() -> {
            account.deposit(500);
            account.withdraw(800);
        });
        Thread t2 = new Thread(() -> {
            account.withdraw(700);
            account.deposit(200);
        });

        t1.setName("Thread 1");
        t2.setName("Thread 2");

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

3) Synchronization Using Locks Build a banking application where multiple threads represent different bank accounts accessing a shared resource (the total balance). • Implementation: o Create a BankAccount class with a method for withdrawing and depositing money. o Use ReentrantLock to synchronize access to the account balance to prevent race conditions. o Demonstrate a scenario where multiple threads try to withdraw funds simultaneously and show how locks ensure thread safety. Key Concepts: • Use of Locks and synchronization. • Avoiding race conditions using ReentrantLock.

Code:

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

class BankAccount {
    private int balance;
    private final Lock lock = new ReentrantLock();
    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
    }
    public void deposit(int amount) {
        lock.lock();
        try {
            balance += amount;
            System.out.println(Thread.currentThread().getName() + " deposited: " + amount + ", Current balance: " + balance);
        } finally {
            lock.unlock();
        }
    }
    public void withdraw(int amount) {
        lock.lock();
        try {
            if (balance >= amount) {
                balance -= amount;
                System.out.println(Thread.currentThread().getName() + " withdrew: " + amount + ", Remaining balance: " + balance);
            } else {
                System.out.println(Thread.currentThread().getName() + " tried to withdraw: " + amount + " but insufficient balance. Current balance: " + balance);
            }
        } finally {
            lock.unlock();
        }
    }
    public int getBalance() {
        return balance;
    }

    public static void main(String[] args) {
        BankAccount sharedAccount = new BankAccount(1000);

        Thread accountHolder1 = new Thread(() -> {
            sharedAccount.deposit(300);
            sharedAccount.withdraw(500);
        });

        Thread accountHolder2 = new Thread(() -> {
            sharedAccount.withdraw(400);
            sharedAccount.deposit(200);
        });

        Thread accountHolder3 = new Thread(() -> {
            sharedAccount.withdraw(800);
        });

        accountHolder1.setName("Account Holder 1");
        accountHolder2.setName("Account Holder 2");
        accountHolder3.setName("Account Holder 3");

        accountHolder1.start();
        accountHolder2.start();

        accountHolder3.start();



        try {

            accountHolder1.join();

            accountHolder2.join();

            accountHolder3.join();

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        System.out.println("Final balance: " + sharedAccount.getBalance());

    }

}
```