4. Java_MIDDLE (многопоточные приложения)

- Распределение операций между потоками (класс Runner, интерфейс Runnable)
- Проблема когерентности кэшэй (ключевое слово Volatile)
- Явная и неявная синхронизация
- Пул потоков
- Паттерн “producer - consumer”
- Использование методов wait(), notify()
- Механизм countDownLatch
- Класс ReentrantLock (методы lock(), unlock())
- Класс Semaphor (методы acquire(), release())
- Deadlock ситуации
- Прерывание потоков
- Класс Future, интерфейс Callable.

Class Threads:

пример создания дополнительных потоков

```
public static void main(String[] args) throws InterruptedException {
    MyThread first = new MyThread();
    first.setName("NUMBER_1");
    first.start();

    MyThread second = new MyThread();
    second.setName("NUMBER_2");
    second.start();
}
```

Class WithRunner :

использование интерфейса Runable

```
public class WithRunner {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Runner());
        thread.start();
```

```
    }
}

class Runner implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Hello from Runner");
        }
    }
```

Class Synchronized :

реализован пример обработки в 2 потока с синхронизацией метода

```
public void doWork() throws InterruptedException {
    Thread thread1 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                increment();
                System.out.println(counter);
            }
        }
    });

    Thread thread2 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                increment();
                System.out.println(counter);
            }
        }
    });
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
    System.out.println(counter);
}
```

Class LockSynchronized:

использование параметра Lock (синхронизация данных на конкретном объекте)

```
public class LockSynchronized {
    public static void main(String[] args) {
```

```
        new Worker().main();
    }
}

class Worker {
    Object lock1 = new Object();
    Object lock2 = new Object();
    Random random = new Random();
    private List<Integer> list1 = new ArrayList<>();
    private List<Integer> list2 = new ArrayList<>();

    public void add1() {
        // lock1 - Синхронизация по конкретному объекту
        synchronized (lock1) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            list1.add(random.nextInt(100));
        }
    }
    public void add2() {
        synchronized (lock2) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            list2.add(random.nextInt(100));
        }
    }
    public void work() {
        for (int i = 0; i < 1000; i++) {
            add1();
            add2();
        }
    }
    public void main() {
        long before = System.currentTimeMillis();

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                work();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                work();
            }
        });
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long after = System.currentTimeMillis();
        System.out.println("Time = " + (after - before) + " ms");
        System.out.println("List 1: " + list1.size());
```

```
        System.out.println("List 2: " + list2.size());
    }
}
```

Class Volatile:

использование параметра volatile (сделать параметр изменяемым для потоков, не кэшируя в каждом ядре)

```
public static void main(String[] args) {
        UsingVolatile newThread = new UsingVolatile();
        newThread.start();
        Scanner scan = new Scanner(System.in);
        scan.nextLine();
        newThread.shutDown();
    }
}

class UsingVolatile extends Thread {
    // running не будет кэшироваться для каждого ядра,
    // обращение к переменной происходит из главной памяти
    private volatile boolean running = true;

    @Override
    public void run() {
        while (running) {
            System.out.println("Hello");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void shutDown() {
        this.running = false;
    }
}
```

Class WaitAndNotify:

синхронизированная приостановка и последующий перезапуск потоков с ожиданием на основе методов wait() и notify()

```
class WaitAndNotify {
    public void produce() throws InterruptedException {
        synchronized (this) {
            System.out.println("Producer thread started");
            wait();    //this.wait    1)Отдает Intrinsic lock 2)Ждет пока
будет вызван метод Notify
            wait(1000);
            System.out.println("Producer thread resumed...");
        }

    }
    public void consume() throws InterruptedException {
        Thread.sleep(2000);
        Scanner scan = new Scanner(System.in);
```

```
        synchronized (this) {
            System.out.println("Waiting for returned Key Pressed");
            scan.nextLine();
            notify();
            Thread.sleep(5000);
        }
    }
```

Class ProducerConsumer:

реализация паттерна "производитель-потребитель"(один поток заполняет очередь, второй отбирает из неё данные) (при этом используются используются методы put, take), необходимости в реализации синхронизации нету. Используется очередь Blocking Queue.

```
private static BlockingQueue<Integer> queue = new ArrayBlockingQueue(10);

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                produce();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
}

private static void produce() throws InterruptedException {
    Random random = new Random();
    while (true) {
        queue.put(random.nextInt(100));
    }
}
private static void consume() throws InterruptedException {
    Random random = new Random();
    while (true) {
        Thread.sleep(100);
        if (random.nextInt(10) == 5) {
            System.out.println(queue.take());
            System.out.println("--------------------------");
```

```
            System.out.println("Queue size = " + queue.size());
        }
    }
}
```

Class ProducerConsumerPart2:

реализация паттерна "производитель-потребитель"(один поток заполняет очередь, второй отбирает из неё данные). При заполнении очереди, или наоборот когда очередь пустая, происходит приостановка и последующий перезапуск потоков на основе методов wait() и notify(). Используются не синхронизированные методы offer() и poll(), поэтому реализована синхронизация на основе lock объектов.

```
    public static void main(String[] args) throws InterruptedException {
        ProducerAndConsumer pc = new ProducerAndConsumer();
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    pc.produce();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    pc.consume();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}
class ProducerAndConsumer{
    private Queue<Integer> queue = new LinkedList<>();
    public final int LIMIT = 10;
    private final Object lock = new Object(); // Все обьъекты синхронизации
должны быть константами

    public void produce() throws InterruptedException {
        int value = 0;
        while (true){
            synchronized (lock){
                while (queue.size() == LIMIT){
                    lock.wait();
```

```
                }
                queue.offer(value++);
                lock.notify();
            }
        }
    }

    public void consume() throws InterruptedException {
        while(true){
            synchronized (lock){
                while(queue.size() == 0){
                    lock.wait();
                }
                int value = queue.poll();
                System.out.println(value);
                System.out.println("Queue size " + queue.size());
                lock.notify();
            }
            Thread.sleep(1000);
        }
    }
```

Class ThreadPool:

создание нескольких потоков с помощью класса ThreadPool. "Исполнитель задач" – ExecutorService. Назначить задачи и запустить их выполнение с помощью методов submit() и shutdown(). А также ожидание выполнения всех задач с помощью метода awaitTermination().

```
public static void main(String[] args) throws InterruptedException {
    // Пул потоков - в данном случае все задачи будут выполнены между 3-мя
исполнителями
    ExecutorService executorService = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 5; i++)
        // submit - заполнение процессами, задачами
        executorService.submit(new Work(i));
    // приступить к выполнению, аналог start
    executorService.shutdown();
    System.out.println("All tasks submited");
    // await - ожидание пока выполнится процесс (1 день)
    executorService.awaitTermination(1, TimeUnit.DAYS);
```

Class ThreadInterrupt:

Прерывание потоков, параметр isInterrupted.

```
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                if (Thread.currentThread().isInterrupted()){
                    System.out.println("Thread was Interrupted");
                }
            }
        });
```

```
        System.out.println("Thread started");
        t1.start();
        Thread.sleep(1000);
        t1.interrupt();
        t1.join();
        System.out.println("Thread finished");
    }
}
```

Class CallableFuture:

использование интерфейса Callable. Реализован способ возврата значения из метода при помощи класса Future.

```
public static void main(String[] args) {
    ExecutorService es = Executors.newFixedThreadPool(1);
    Future<Integer> future = es.submit(() -> {
        System.out.println("Starting");
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Finished");
        Random random = new Random();
        int randomValue = random.nextInt();
        if (randomValue < 5)
            throw new Exception("Something happen ...  ");
        // От наличия return, Java определяет какой интерфейс реализовать
Callable или Runnable
        return random.nextInt(10);
    });

    es.shutdown();
    try {
        int result = future.get(); // get дожидается окончания выполнения
потока
        System.out.println(result);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        Throwable ex = e.getCause();
        System.out.println(ex.getMessage());
    }
```

Class ReentrantLock:

инструмент синхронизации потоков. Объект Lock, методы lock() и unlock() - захватить или освободить объект синхронизации.

```
public class ReetrantLockExample {
    public static void main(String[] args) throws InterruptedException {
        Task task = new Task();
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                task.First();
            }
```

```
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                task.Second();

            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        task.show();

    }
}
class Task{
    private int counter;
    private Lock lock = new ReentrantLock();

    public  void increment(){
        for (int i =0; i<10000; i++){
            counter++;
        }
    }
    public void First(){
        lock.lock();
        increment();
        lock.unlock();
    }
    public void Second(){
        lock.lock();
        increment();
        lock.unlock();
    }
    public void show(){
        System.out.println(counter);
    }
}
```

Class CountDownLatchExample:

потокобезопасный обратный счетчик. Метод await() - ожидает пока счетчик не станет равным нулю для выполнения дальнейших операций. Countdown() - уменьшает счетчик на единицу.

```
public class CountDownLatchExample {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(3);
        ExecutorService es = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 3; i++)
            es.submit(new Proc(i, countDownLatch));
        es.shutdown();
        for (int i = 0; i < 3; i++) {
            Thread.sleep(1000);
            countDownLatch.countDown();
        }
    }
}
```

```
class Proc implements Runnable {
    private CountDownLatch countDownLatch;
    private int id;

    public Proc(int id, CountDownLatch countDownLatch) {
        this.id = id;
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run() {
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread with id " + id + " proceed");
    }
}
```

Class Semaphore:

Semaphore - позволяет разграничить доступ к ресурсу в один момент времени. В качестве примера используется ситуация с подключением к какому либо серверу. Было создано 200 потоков (имитируют пользователей) и объект Семафор на 10 разрешений. Тем самым одновременно будут выполняться только 10 операций (одновременнных подключений). Метод acquire() - занимает ячейку (создаёт подключение). Release() - освобождает место для другого пользователя.

```
public class SemaphoreExample {
    public static void main(String[] args) throws InterruptedException {
        Connection connection = Connection.getConnection();
        ExecutorService es = Executors.newFixedThreadPool(200);
        for (int i = 0; i < 200; i++) {
            es.submit(new Runnable() {
                @Override
                public void run() {
                    try {
                        connection.doWorkWithSemaphore();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
        es.shutdown();
        es.awaitTermination(1, TimeUnit.DAYS);
    }
}

class Connection {  //Singleton pattern
    private static Connection connection = new Connection();
    private int connectionsCount = 0;
    //Количество разрешений, сколько потоков одновременно может обрабатывать
```

```
данные
    private Semaphore semaphore = new Semaphore(10);

    private Connection() {
    }

    public static Connection getConnection() {
        return connection;
    }

    public void doWorkWithSemaphore() throws InterruptedException {
        // acquire - начать взаимодействовать с ресурсом
        semaphore.acquire();
        try {
            doWork();
        } finally {
            // release - закончить использовать ресурс (всегда в finally
блоке)
            semaphore.release();
        }
    }

    private void doWork() throws InterruptedException {
        synchronized (this) {
            connectionsCount++;
            System.out.println(connectionsCount);
        }
        Thread.sleep(5000);
        synchronized (this) {
            connectionsCount--;
        }
    }
}
```

Class DeadLockExample:

ситуация взаимной блокировки потоков (Deadlock). В качестве примера используется аналог банковских операций. Для успешной транзакции, необходимо одновременно синхронизироваться на двух объектах (в данном случае - 2-х банковских аккаунтах). Если в один момент объекты перехвачены разными потоками, возникнет взаимная блокировка. Каждый поток будет бесконечно ждать пока освободится другой объект. Синхронизация в даном случае реализована с помощью класса ReentrantLock. Используется метод tryLock(), объекта Lock.

```
public class DeadLockExample {
    public static void main(String[] args) throws InterruptedException {
        Runer runer = new Runer();
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                runer.firstThread();

            }
```

```
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                runer.secondThread();
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        runer.finished();

    }
}

class Runer {
    private Account account1 = new Account();
    private Account account2 = new Account();
    private Lock lock1 = new ReentrantLock();
    private Lock lock2 = new ReentrantLock();


    private void takeLocks(Lock lock1, Lock lock2) {
        boolean firstLockTaken = false;
        boolean secondLockTaken = false;
        while (true) {
            try {
                // tryLock - попытка забрать lock для синхронизации с потоком
                firstLockTaken = lock1.tryLock();
                secondLockTaken = lock2.tryLock();
            } finally {
                if (firstLockTaken && secondLockTaken) {
                    return;
                }
                // если одновременно не смогли забрать оба объекта lock,
необходимо освободить занятый Lock, чтобы другой
                // поток забрал его себе и успешно синхронизировался
                if (firstLockTaken) {
                    lock1.unlock();
                }
                if (secondLockTaken) {
                    lock2.unlock();
                }
            }
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void firstThread() {
        Random random = new Random();
        for (int i = 0; i < 10000; i++) {
            //synchronized (account1) {
            //    synchronized (account2) {
            //lock1.lock();
            //lock2.lock();
            takeLocks(lock1, lock2);
            try {
                Account.transfer(account1, account2, random.nextInt(100));
            } finally {
```

```
                lock1.unlock();
                lock2.unlock();
            }
        }
    }

    public void secondThread() {
        Random random = new Random();
        // DeadLock - Синхронизация в разных порядках, потоки не передают
данные друг другу,
        // один объект захвачен одним потоком, другой - другим
        for (int i = 0; i < 10000; i++) {
            takeLocks(lock2, lock1);
            try {
                Account.transfer(account2, account1, random.nextInt(100));
            } finally {
                lock1.unlock();
                lock2.unlock();
            }
        }
    }

    public void finished() {
        System.out.println(account1.getBalance());
        System.out.println(account2.getBalance());
        System.out.println("Total = " + (account1.getBalance() +
account2.getBalance()));
    }
}

class Account {
    private int balance = 10000;

    public void deposit(int amount) {
        balance += amount;
    }

    public void withDraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }

    public static void transfer(Account acc1, Account acc2, int amount) {
        acc1.withDraw(amount);
        acc2.deposit(amount);
    }
}
```