ЗАНЯТИЕ 1.14

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Тема: Создание многопоточного Java-приложения. Использование Java Stream API.

Упражнение №1

В данном упражнении предстоит изучить особенности многопоточного доступа к общему ресурсу (данным).

Создайте новое приложение. Добавьте в него новый класс **BankAccount** (банковский счет). В классе должны быть переменные number и balance, характеризующие банковский счет. Реализуйте конструктор, геттеры, сеттеры, а также метод **changeBalance**, который уменьшает баланс счета после покупки на величину value.

```
public class BankAccount {

    private long number; // номер счета
    private int balance; // баланс счета

    public BankAccount(long number, int balance) {
        this.number = number;
        this.balance = balance;
    }

    public void changeBalance(int value){
        this.balance -= value;
    }

    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }
}
```

Рисунок 1

Этот класс и будет тем самым общим ресурсом, с которым будут работать потоки. Потоки – это клиенты банка, у которых есть доступ к одному банковскому счету. Клиенты банка будут одновременно покупать товары, используя один банковский счет. Создайте класс **ClientRunnable**. В нем будет реализована работа с банковским счетом (покупка товаров). Данный класс должен реализовывать интерфейс **Runnable**. Переменные класса: ссылка на банковский счет, с которого будут оплачены товары, а также фиксированная сумма товара. Создайте конструктор.

```java
public class ClientRunnable implements Runnable{

    private BankAccount bankAccount; // ссылка на банковский счёт
    private int value = 350; // фиксированная сумма товара

    public ClientRunnable(BankAccount bankAccount) {
        this.bankAccount = bankAccount;

    }
```

Рисунок 2

Переопределите метод **run** в классе **ClientRunnable**. В методе run реализуйте имитацию покупки товаров в цикле (пока не закончатся деньги на балансе счета).

```java
@Override
public void run() {
    while(true){
        System.out.println(Thread.currentThread().getName() + " проверил баланс до покупки: " + bankAccount.getBalance());
        if(bankAccount.getBalance() - value >= 0){ // проверка позволяет ли баланс совершить покупку
            try {
                Thread.sleep( millis: 1500);// имитация работы по обработке платежа
                bankAccount.changeBalance(value); // уменьшение баланса на сумму товара
            } catch (InterruptedException ex) {

            }
        System.out.println(Thread.currentThread().getName() + " проверил баланс после покупки: " + bankAccount.getBalance());
        }else{
            break; // выходим из цикла, если денег для покупки недостаточно
        }
    }
```

Рисунок 3

В главном классе проекта, в методе **main** создайте два потока, которые будут одновременно покупать товары, используя один банковский счет. С помощью метода join реализуйте ожидание завершения работы потоков, чтобы вывести в главном потоке остаток баланса.

```java
public static void main(String[] args){
    BankAccount account = new BankAccount( number: 1234567812345678L,  balance: 1000);
    Runnable rn = new ClientRunnable( bankAccount: account);
    Thread client1 = new Thread( r: rn,  string: "Клиент 1");// первый клиент
    Thread client2 = new Thread( r: rn,  string: "Клиент 2");// второй клиент
    client1.start();
    client2.start();
    try {
        client1.join();
        client2.join();
    } catch (InterruptedException ex) {

    }
    System.out.println("Баланс после покупок: " + account.getBalance());
```

Рисунок 4

Когда потоки закончат свою работу (средств на счету будет недостаточно для покупок товаров) в главном потоке программы отобразится информация об остатке на счету.

Запустите программу.

```
Клиент 1 проверил баланс до покупки: 1000
Клиент 2 проверил баланс до покупки: 1000
Клиент 1 проверил баланс после покупки: 650
Клиент 1 проверил баланс до покупки: 650
Клиент 2 проверил баланс после покупки: 300
Клиент 2 проверил баланс до покупки: 300
Клиент 1 проверил баланс после покупки: -50
Клиент 1 проверил баланс до покупки: -50
Баланс после покупок: -50
------------------------------------------
BUILD SUCCESS
------------------------------------------
```

Рисунок 5

Неправильная работа с общим ресурсом (объектом класса BankAccount) привела к тому, что баланс счета стал отрицательным (чего быть не должно). Как это исправить?

Чтобы исключить подобную логическую ошибку, надо как-то блокировать объект класса BankAccount, чтобы пока он обрабатывается в одном потоке, другие не могли его изменять.

Одно из решений – использование ключевого слова **synchronized**. Им помечается определенный блок кода. Если блок кода помечен ключевым словом synchronized, это значит, что блок может выполняться только одним потоком одновременно. Синхронизацию можно реализовать по-разному: создать целый синхронизированный метод или написать блок кода, где синхронизация осуществляется по какому-то объекту.

Если один поток зашел внутрь блока кода, который помечен словом synchronized, он моментально захватывает монитор объекта, и все другие потоки, которые попытаются зайти в этот же блок или метод вынуждены ждать, пока предыдущий поток не завершит свою работу и не освободит монитор.

Что за монитор?

Каждый объект в Java имеет Intrinsic Lock (монитор). Когда синхронизированный метод вызывается из потока, ему нужно получить этот монитор. Монитор будет освобождён после того, как поток завершит выполнение метода. Таким образом, мы можем синхронизировать блок инструкций, работающий с объектом, принудив потоки получать монитор, прежде чем выполнять блок инструкций. Помните, что монитор одновременно может удерживаться только одним потоком, поэтому другие потоки, желающие получить его, будут приостановлены до завершения работы

текущего потока. Только после этого конкретный ожидающий поток сможет получить монитор и продолжить выполнение.

Реализуйте блокировку блока кода в методе run с помощью synchronized для объекта класса BankAccount:

```
@Override
public void run() {
    synchronized (this.bankAccount) {
        while(true){
            System.out.println(Thread.currentThread().getName() + " проверил баланс до покупки: " + bankAccount.getBalance());
            if(bankAccount.getBalance() - value >= 0){ // проверка позволяет ли баланс совершить покупку
                try {
                    Thread.sleep( millis: 1500);// имитация работы по обработке платежа
                    bankAccount.changeBalance(value); // уменьшение баланса на сумму товара
                } catch (InterruptedException ex) {

                }
            System.out.println(Thread.currentThread().getName() + " проверил баланс после покупки: " + bankAccount.getBalance());
            }else{
                break; // выходим из цикла, если денег для покупки недостаточно
            }
        }
    }
}
```

Рисунок 6

Запустите программу. Убедитесь, что она выполняется корректно.

Еще одним средством синхронизации доступа к общему ресурсу является использование специального объекта **семафор**.

В Java семафоры представлены классом **Semaphore**, который располагается в пакете java.util.concurrent.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Добавьте создание объекта Semaphore с 1 разрешением в код класса ClientRunnable.

```
public class ClientRunnable implements Runnable{

    private BankAccount bankAccount; // ссылка на банковский счёт
    private int value = 350; // фиксированная сумма товара
    // создание семафора с 1 разрешением на доступ к блоку кода
    private Semaphore semaphore = new Semaphore( i: 1);

    public ClientRunnable(BankAccount bankAccount) {
        this.bankAccount = bankAccount;

    }
```

Рисунок 7

Замените блок synchronized на использование семафора:

```
@Override
public void run() {
    try {
        semaphore.acquire(); // запрос разрешения на доступ
    } catch (InterruptedException ex) {
        Logger.getLogger( name:ClientRunnable.class.getName()).log( level:Level.SEVERE,  msg:null,  thrown:ex);
    }
        while(true){
            System.out.println(Thread.currentThread().getName() + " проверил баланс до покупки: " + bankAccount.getBalance());
            if(bankAccount.getBalance() - value >= 0){ // проверка позволяет ли баланс совершить покупку
                try {
                    Thread.sleep( millis:1500);// имитация работы по обработке платежа
                    bankAccount.changeBalance(value); // уменьшение баланса на сумму товара
                } catch (InterruptedException ex) {

                }
            System.out.println(Thread.currentThread().getName() + " проверил баланс после покупки: " + bankAccount.getBalance());
            }else{
                break; // выходим из цикла, если денег для покупки недостаточно
            }
        }
        semaphore.release(); // освобождение доступа
    }
}
```

Рисунок 8

Запустите программу и убедитесь, что она корректно отображает остаток на счете:

```
Клиент 2 проверил баланс до покупки: 1000
Клиент 2 проверил баланс после покупки: 650
Клиент 2 проверил баланс до покупки: 650
Клиент 2 проверил баланс после покупки: 300
Клиент 2 проверил баланс до покупки: 300
Клиент 1 проверил баланс до покупки: 300
Баланс после покупок: 300
------------------------------------------------
BUILD SUCCESS
```

Рисунок 9

Упражнение №2

В данном упражнении предстоит изучить механизм взаимодействия потоков исполнения с помощью методов **wait**, **notify** и **notifyAll**. Эти методы реализованы как final в классе Object, поэтому они доступны всем классам.

Все три метода могут быть вызваны только из synchronized кода.

Рассмотрим, как мы можем использовать эти методы. Возьмем стандартную задачу – «Производитель-Потребитель»: пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. Но при этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем методы **wait** и **notify**.

Создайте новое приложение. Добавьте в него класс Store (магазин). В классе магазин создайте переменную product.

```
 */
public class Store {

    private int product = 0;
```

Рисунок 10

Она будет отображать количество доступных товаров в магазине. Также создайте два синхронизованных метода в классе Store – **put** и **get** (для добавления и покупки товаров соответственно).

```
public synchronized void put() {
    while (product >= 3) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    product++;
    System.out.println( x: "Производитель добавил 1 товар");
    System.out.println("Товаров в магазине: " + product);
    notify();
}
```

Рисунок 11

То есть пока товаров на полке в магазине 3 и более, новых поступать не будет, поэтому вызывается метод wait. Этот метод освобождает монитор объекта Store и блокирует дальнейшее выполнение метода put, пока для этого же монитора не будет вызван метод notify из метода get.

Аналогично с методом **get**:

```
public synchronized void get() {
    while (product < 1) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    product--; // покупка товара
    System.out.println( x: "Покупатель купил 1 товар");
    System.out.println("Товаров в магазине: " + product);
    notify();
}
```

Рисунок 12

Для отслеживания наличия товаров в классе Store проверяем значение переменной product. По умолчанию товара нет, поэтому переменная равна 0.

Метод get() - получение товара должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе get проверяем в цикле while, отсутствует ли товар:

Если товар отсутствует, вызывается метод wait. Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же монитора не будет вызван метод notify.

Когда в методе put добавляется товар и вызывается notify, то метод get получает монитор и выходит из конструкции *while (product<1)*, так как товар добавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение. И в конце вызов метода notify дает сигнал методу put продолжить работу.

Создайте классы **Manufacturer** и **Consumer** для имитации производства и покупки товаров, соответственно. Классы будут реализовывать интерфейс **Runnable**. В качестве аргумента конструктора используется ссылка на Store.

```
public class Manufacturer implements Runnable{

    private Store store;

    public Manufacturer(Store store){
       this.store = store;
    }

    @Override
    public void run(){
        for (int i = 1; i < 10; i++) {
            store.put();
        }
    }
}
```

Рисунок 13

```
public class Consumer implements Runnable{

    private Store store;

    public Consumer(Store store) {
        this.store = store;
    }

    @Override
    public void run() {
        for (int i = 1; i < 10; i++) {
            store.get();
        }
    }

}
```

Рисунок 14

В главном классе, в методе **main** создайте и запустите потоки производителя и потребителя:

```
public static void main(String[] args) {
    Store store = new Store();
    new Thread(new Manufacturer(store)).start();
    new Thread(new Consumer(store)).start();
}
```

Рисунок 15

Запустите программу, убедитесь, что она корректно отображает результат.

Упражнение №3

Создайте новое приложение. В методе **main** создайте поток данных (stream) из массива слов. Используйте промежуточные операторы filter (для отбора коротких слов), map (для преобразования текста в верхний регистр), distinct (для удаления повторяющихся элементов – останутся только уникальные). Терминальный оператор forEach применит метод System.out.println ко всем элементам потока.

```
// вывести в консоль CAPS-ом все уникальные короткие слова из текста
String text = "Студенты очень любят посещать лекции по Java и Android "
        + "разработке. Всегда слушают очень внимательно и не спят.";
String[] words = text.split( regex: "\\PL+");
Stream stream = Arrays.stream( array: words);
stream.filter(x -> x.toString().length() <= 3)
      .map(t -> t.toString().toUpperCase())
      .distinct()
      .forEach(x -> System.out.println(x));
```

Рисунок 16

Запустите программу и убедитесь, что результат выводится корректно.

```
--- exec-maven-plugin:3.0.0:exec (de
ПО
И
НЕ
------------------------------------
BUILD SUCCESS
------------------------------------
```

Рисунок 17

«Раскройте» lambda-выражение в блоке filter (нажмите на желтую лампочку и выберете «Use anonymous inner class»):

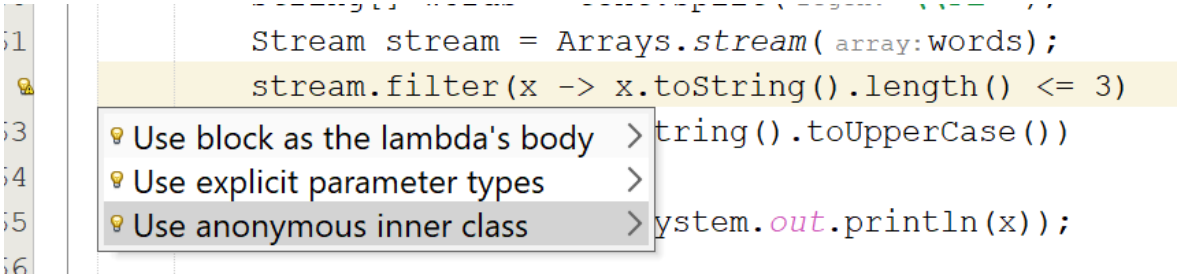

Рисунок 18

В результате лямбда будет преобразована к анонимному классу, реализующему функциональный интерфейс Predicate:

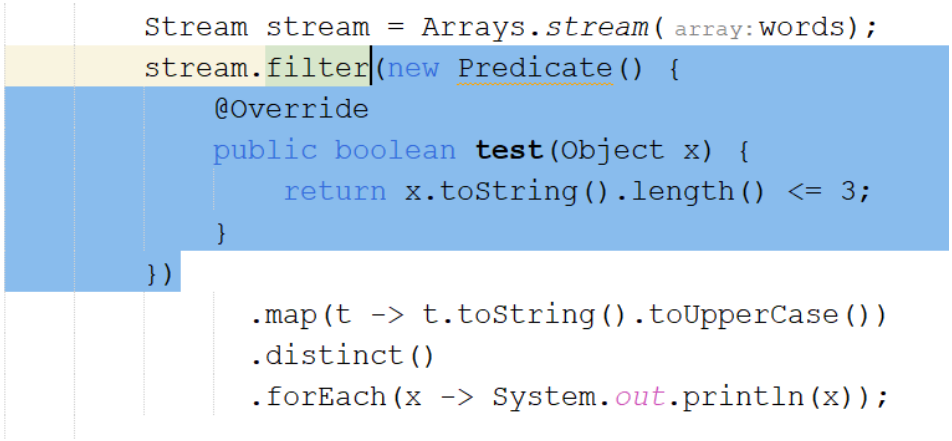

Рисунок 19

Создайте новый поток из коллекции чисел (Integer).

```
List<Integer> lst = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8});
Stream myIntegerStream = lst.stream();
```

Рисунок 20

Реализуйте фильтрацию и преобразование элементов (какие именно – по своему выбору) с помощью промежуточных операторов filter и map. Отобразите в консоль измененную коллекцию (используйте терминальный оператор collect).

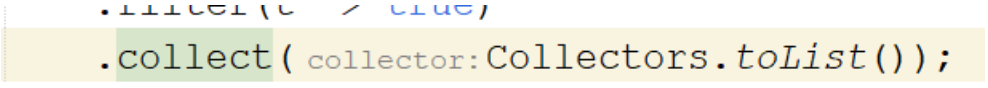

Рисунок 21

Индивидуальное задание

Требования:

1) На сдачу практического задания отводится **14 дней**.
2) Вариант определяется согласно порядковому номеру студента в журнале.
3) В случае дистанционного выполнения практического задания код программы необходимо выложить на Github и предоставить ссылку на него.
4) Итоговая программа должна компилироваться без ошибок, полностью выполнять требуемый функционал и при старте выводить в консоль номер варианта и ФИО студента.
5) Названия переменных и методов должны отражать суть и нести смысловую нагрузку.
6) Необходимо придерживаться стилистике по написанию Java-кода.
7) Необходимо предусмотреть защиту от ввода «аномальных» (ошибочных) значений.

Дополнительная информация:

1) Перед выполнением задания рекомендуется ознакомиться с Лекциями №1.11, №1.13.
2) Рекомендуется использовать обработку исключений.

Задание: разработать многопоточную консольную Java-программу в соответствии с вариантом.

Вариант 1. Разработать программу, которая выводит в консоль с интервалом 1 сек. имя текущего работающего потока строго по очереди. Обязательные требования к программе:

- Программа должна работать непрерывно (будет плюсом, если реализуете прерывание работы по нажатию на какую-нибудь клавишу).
- Для демонстрации работы необходимо создать **три** дополнительных потока.
- Программа должна использовать методы wait() и notify().

Результат работы программы:

Thread-0

Thread-1

Thread-2

Thread-0

Thread-1

Thread-2

Thread-0

Thread-1

Thread-2

И т.д.

Вариант 2. Разработать программу для моделирования ситуации в студенческом буфете, где остался свободным только один столик. В буфет пришли 7 студентов, свободный столик только один и одновременно за ним могут обедать только двое студентов. Необходимо с использованием средств многопоточного программирования накормить всех студентов. Очередность в данном случае не имеет значения. Обязательные требования к программе:

- В программе необходимо предусмотреть вывод в консоль (для наглядности) следующих состояний для студента: waiting (ожидание свободного места за столом), eating (прием пищи), exit (выход из-за стола).
- В рамках моделирования процедуры обеда заменить использовать 3-секундный сон потока.
- Одновременно могут обедать только два студента (то есть пока студент не выйдет из-за стола, следующий не может начать обедать).
- При выполнении задания рекомендуется использовать семафоры

В результате работы программы необходимо вывести в консоль демонстрацию поведения 7 студентов. Примерный результат может выглядеть следующим образом:

Student1 waiting

Student6 waiting

Student6 eating

Student7 waiting

Student4 waiting

Student5 waiting

Student3 waiting

Student2 waiting

Student1 eating

Student1 exit

Student7 eating

Student6 exit

Student4 eating

Student7 exit

Student5 eating

Student4 exit

Student3 eating

Student3 exit

Student5 exit

Student2 eating

Student2 exit

Вариант 3. Разработать программу для нахождения максимального и минимального значений из числовой последовательности. Обязательные требования к программе:

- Последовательность чисел вводит пользователь после старта программы с клавиатуры через запятую.
- Программа после ввода значений должна запускать два дополнительных потока. Один поток вычисляет максимальное значение, второй – минимальное.
- Результаты вычисления необходимо получить из метода **main**.

В результате работы программы необходимо вывести в консоль из метода main максимальное и минимальное значения из входной последовательности.

Вариант 4. Разработать программу, которая строго по очереди выводит в консоль имя текущего работающего потока. Обязательные требования к программе:

- Программа должна работать непрерывно (будет плюсом, если реализуете прерывание работы по нажатию на какую-нибудь клавишу).
- Для демонстрации работы необходимо создать **два** дополнительных потока.
- Программа должна использовать synchronized или wait/notify.

Результат работы программы:

Thread-0

Thread-1

Thread-0

Thread-1

Thread-0

Thread-1

И т.д.

Вариант 5. Разработать программу для генерации списка случайных целых чисел и последующей их сортировки. Обязательные требования к программе:

- Необходимо создать два дополнительных потока. Первый из них должен генерировать 100 случайных целых чисел (рекомендуется использовать класс **Random**) в диапазоне [0,1000). После этого второй поток должен отсортировать полученный список по возрастанию.
- Потоки должны быть созданы с использованием интерфейса Runnable

В результате работы программы необходимо вывести список сгенерированных чисел в порядке возрастания.

Например:

[1, 11, 18, 19, 31, 36, 38, 42, 45, 89, 91, 120, 133, 163, 165, 170, 177, 179, 185, 191, 191, 195, 199, 211, 217, 226, 237, 241, 247, 251, 295, 299, 302, 305, 317, 318, 324, 328, 337, 342, 354, 356, 368, 369, 435, 442, 453, 457, 457, 458, 471, 479, 486, 489, 489, 491, 494, 518, 523, 525, 546, 553, 554, 562, 564, 580, 592, 598, 647, 660, 676, 685, 703, 712, 716, 732, 738, 745, 747, 748, 757, 788, 790, 798, 800, 829, 835, 848, 852, 857, 876, 877, 922, 925, 938, 948, 964, 975, 986, 995]