1 Классическая проблема производитель - потребитель.
1.1.Синхронизация потоков выполнения.

Существует множество ситуаций, когда отдельные, но работающие одновременно потоки выполнения должны взаимодействовать друг с другом, чтобы получить доступ к различным общим ресурсам или динамически передавать результаты своей «работы». Наиболее красноречивый сценарий, в котором потоки выполнения должны взаимодействовать друг с другом, известен как проблема *производителя/потребителя,* в которой производитель генерирует поток данных, который принимается и обрабатывается потребителем. Рассмотрим, например, Java-приложение, в котором один поток выполнения (производитель) записывает данные в файл, а другой поток выполнения (потребитель) читает данные из того же файла для их обработки. Или предположим, что производитель генерирует числа и поочередно помещает их в буфер, а потребитель считывает числа из этого буфера для их интерпретации. В обоих случаях мы имеем дело с конкурирующими потоками выполнения, которые используют общий ресурс: файл, вектор, и они должны быть синхронизированы в зависимости от их деятельности. Чтобы лучше понять, как синхронизировать два потока выполнения, давайте фактически реализуем задачу типа «производитель/потребитель». Рассмотрим следующую ситуацию:

- Производитель генерирует целые числа от 1 до 10, каждое в случайном интервале от 0 до 100 миллисекунд. По мере их генерации он пытается поместить их в область памяти (целую переменную), откуда они будут считываться потребителем.
- Потребитель будет поочередно принимать числа, сгенерированные производителем, и отображать их значение на экране.

Чтобы быть доступной для обоих потоков выполнения, мы инкапсулируем переменную, которая будет содержать сгенерированные числа, в объект, описанный классом **`Buffer`**, который будет иметь два метода **`put`** (для помещения числа в буфер) и **`get`** (для получения числа из буфера). Без использования какого-либо механизма синхронизации класс Buffer выглядит следующим образом:

```
class Buffer {
       private int number = -1;

       public int get() {
         return number;
       }

       public void put(int number) {
              this.number = number;
       }
}
```

Теперь мы реализуем классы Producator и Consumator, которые будут описывать два потока выполнения. Оба будут иметь общую ссылку на объект типа `Buffer`, через который они обмениваются значениями.

```
class Производитель extends Thread {
   private Buffer buffer;
      public Производитель(Buffer b) {
        buffer = b;
      }
      public void run() {
```

```
        for (int i = 0; i < 10; i++) {
           buffer.put(i);
       System.out.println("Производитель поставил:\t"+ i);
            try {
              sleep((int)(Math.random() * 100));
              } catch (InterruptedException e) { }
       }
       }
}

class Потребитель extends Thread {
   private Buffer buffer;
   public Потребитель(Буфер b) {
      buffer = b;
   }
   public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
       value = buffer.get();
       System.out.println("Потребитель получил
                           получил:\t" + value);
      }
   }
}
//Основной класс
public class TestSincronizare1 {
       public static void main(String[] args) {
              Buffer b = new Buffer();
              Производитель p1 = new Производитель(b);
              Потребитель c1 = new Потребитель(b);
              p1.start();
              c1.start();
       }
}
```

Результат выполнения этой программы не решит предложенную нами задачу, поскольку между двумя потоками выполнения отсутствует какая-либо синхронизация. Точнее, результат будет примерно следующим:

```
Потребитель получил:     -1
Потребитель получил:     -1
Производитель поставил:  0
Потребитель получил:     0
Потребитель получил:     0
Потребитель получил:     0
Производитель поставил:  1
Производитель поставил:  2
Производитель поставил:  3
Производитель поставил:  4
Производитель поставил:  5
Производитель поставил:  6
```

```
Производитель поставил:   7
Производитель поставил:   8
Производитель поставил:   9
```

Оба потока выполнения обращаются к общему ресурсу, то есть объекту типа `Buffer`, хаотичным образом, и это происходит по двум причинам:

- потребитель не ждет, пока производитель сгенерирует число, и будет принимать одно и то же число несколько раз.
- производитель не ждет, пока потребитель получит сгенерированное число, прежде чем сгенерировать другое, таким образом потребитель обязательно «пропустит» некоторые числа (в нашем случае, почти все).

Проблема заключается в следующем: кто должен заниматься синхронизацией двух потоков выполнения: классы «Производитель» и «Потребитель» или общий ресурс «`Буфер`»? Ответ: общий ресурс «`Буфер`», поскольку он должен разрешать или запрещать доступ к своему содержимому, а не потоки выполнения, которые его используют. Таким образом, усилия по синхронизации переносятся с производителя/потребителя на более низкий уровень, уровень критического ресурса. Деятельность производителя и потребителя должна быть синхронизирована на уровне общего ресурса в двух аспектах:

1. Два потока выполнения не должны одновременно обращаться к буферу; это достигается путем блокировки объекта `Buffer`, когда к нему обращается один поток выполнения, так что ни один другой поток выполнения не может к нему обратиться.
2. Два потока выполнения должны координироваться, то есть производитель должен найти способ «сообщить» потребителю, что он поместил значение в буфер, а потребитель должен сообщить производителю, что он получил это значение, чтобы тот мог сгенерировать другое значение. Для осуществления этой коммуникации класс `Thread` предоставляет методы **`wait, notify, notifyAll`**.

Исходя из вышеизложенного, класс `Buffer` будет выглядеть следующим образом:

```
class Buffer {
	private int number = -1;
	private boolean available = false;
	public synchronized int get() {
		while (!доступно) {
			try {
				wait();
	//ожидание, пока производитель не установит значение
			} catch (InterruptedException e) { }
			}
			available = false;
			notifyAll();
			return number;
	}
	public synchronized void put(int number) {
		while (доступно) {
			try {
				wait();
	//ожидание, пока потребитель не получит значение
			} catch (InterruptedException e) { }
			}
			this.number = number;
```

```
            available = true;
            notifyAll();
        }
}
```

Результат выполнения:

```
            Производитель указал:     0
            Потребитель получил:      0
            Производитель вложил:     1
            Потребитель получил:      1
            .
            Производитель поставил:   9
```

2. Методы синхронизации потоков

2.1 Блокировка объекта (ключевое слово synchronized)

Определение: **Сегмент кода, который управляет ресурсом, общим для нескольких отдельных и конкурирующих потоков выполнения, называется *критическим участком*.** В Java критическим участком может быть блок инструкций или метод.

Контроль доступа в критическом участке осуществляется с помощью ключевого слова **`synchronized`**. Платформа Java связывает монитор с каждым объектом программы, содержащим критические участки, которые требуют синхронизации. Этот монитор будет указывать, доступен ли критический ресурс какому-либо потоку выполнения или он свободен, другими словами, «мониторит» критический ресурс. В случае доступа он «заблокирует» его, чтобы предотвратить доступ других потоков выполнения к нему. В момент освобождения ресурса «блокировка» будет снята, чтобы разрешить доступ другим потокам выполнения.

В приведенном выше примере типа «производитель/потребитель» критическими участками являются методы **`put`** и **`get,`** а общим критическим ресурсом является объект **`buffer`**. Потребитель не должен обращаться к буферу, когда производитель только что помещает в него значение, а производитель не должен изменять значение в буфере в момент, когда оно читается потребителем.

```
public synchronized int get() {
        ...
}
public synchronized void put(int number) {
        ...
}
```

Оба метода были объявлены с модификатором **`synchronized`**. Однако система связывает монитор с экземпляром класса **`Buffer`**, а не с конкретным методом. В момент получения монитора поток выполнения, который сделал вызов, заблокирует объект, к которому обращается метод, что означает, что другие потоки выполнения больше не смогут получить доступ к критическим ресурсам. Поскольку несколько критических секций (синхронных методов) объекта управляют одним критическим ресурсом. В нашем примере, когда производитель вызывает метод **`put`** для записи числа, он блокирует весь объект типа **`Buffer,`** поэтому поток выполнения потребителя не будет иметь доступа к другому синхронному методу **`get,`** и наоборот.

```
public synchronized void put(int number) {
```

```
        // буфер заблокирован производителем
        ...
        // буфер разблокирован производителем
    }
    public synchronized int get() {
        // буфер заблокирован потребителем
        ...
        // буфер разблокирован потребителем
    }
```

2.2. Методы `wait`, `notify` и `notifyAll`

Объект типа **`Buffer`** в приведенном выше примере имеет частную переменную member под названием **`number`**, в которой хранится число, сообщаемое производителем и принимаемое потребителем. Частная логическая переменная **`available`**, которая указывает состояние буфера: если она имеет значение **`true`**, это означает, что производитель поместил значение в буфер, а потребитель еще не получил его; если она имеет значение **`false`**, потребитель получил значение из буфера, но производитель не поместил другое значение в буфер. Методы класса **`Buffer`**:

```
    public synchronized int get() {
        if (available) {
            available = false;
            return number;
        }
    }
    public synchronized int put(int number) {
        if (!available) {
            доступно = true;
        this.number = number;
        }
    }
```

Реализация этих методов не даст правильного результата, поскольку потоки выполнения не синхронизируют доступ к буферу. Ситуации, в которых методы **`get`** и **`put`** ничего не делают, приведут к потере некоторых чисел потребителем или к двукратному получению одного и того же числа. Таким образом, два потока выполнения должны ждать друг друга.

```
    public synchronized int get() {
        while (!available) {
    //ничего — жду, пока переменная станет true
        }
        available = false;
        return number;
    }
    public synchronized int put(int number) {
        while (доступно) {
    //ничего - жду, пока переменная станет false
        }
      available = true;
      this.number = number;
    }
```

Программа некорректна, поскольку два метода «эгоистично» ожидают условия завершения. В результате корректность работы будет зависеть от операционной системы, что является ошибкой программирования. Правильное помещение потока выполнения в ожидание осуществляется с помощью метода `wait` класса `Thread`, который имеет три формы:

```
void wait( )
void wait( long timeout )
void wait( long timeout, long nanos ).
```

После вызова метода **`wait`** текущий поток выполнения освобождает монитор, связанный с соответствующим объектом, и ожидает выполнения одного из следующих условий:

- другой поток выполнения сообщает тем, кто «ожидает» на определенном мониторе, «проснуться»; это осуществляется путем вызова метода **`notifyAll`** или **`notify`**.
- указанный период ожидания истек.

Метод **`wait`** может генерировать исключения типа `InterruptedException`, когда поток выполнения находится в состоянии ожидания –`Not Runnable`, прерывается из ожидания и принудительно переводится в состояние `Runnable`. Метод `notifyAll` информирует все потоки выполнения, которые находятся в состоянии ожидания на мониторе текущего объекта, о выполнении условия, которого они ждали. Метод `notify` информирует только один поток выполнения. Правильный вариант методов `get` и put:

```
public synchronized int get() {
   while (!available) {
      try {
          wait();
//ожидает, пока производитель не установит значение
         } catch (InterruptedException e) { }
   }
   available = false;
   notifyAll();
   return number;
}

public synchronized void put(int number) {
   while (available) {
     try {
      wait();
//ожидает, пока потребитель не получит значение
         } catch (InterruptedException e) { }
    }
  this.number = number;
  available = true;
  notifyAll();
}}
```

2.3 Барьеры

В многопоточных приложениях необходимо, чтобы определенные потоки синхронизировались в определенный момент. Примером может служить параллельный вычисление в фазе, когда все потоки должны завершить фазу выполнения, прежде чем

одновременно перейти к следующей фазе. Барьер — это механизм, используемый для синхронизации нескольких потоков. Поток, который встречает барьер, автоматически входит в `wait()`. Когда последний поток «достигает» барьера, он сигнализирует (`notify()`) другим потокам, которые находятся в ожидании, что приводит к групповому «прохождению» барьера. Вот пример этого:

```
import java.util.*;
class Barrier { // Класс Barrier синхронизирует всех
//участников private
   int ParticipatingThreads;
   private int WaitingAtBarrier;
   public Barrier(int num){ //Конструктор
    ParticipatingThreads = num;
       WaitingAtBarrier=0;
   }
    public synchronized void Reached(){
       //Метод барьера
       WaitingAtBarrier++;
      if(ParticipatingThreads != WaitingAtBarrier){
       //Это означает, что поток не является последним
         try {
              wait(); //Поток останавливается до тех пор, пока
                      //не будет освобожден
             } catch (InterruptedException e) { }
      } else {  // Это был последний активный поток
          notifyAll();
          WaitingAtBarrier=0;//освобождает все
     }
  }
}
```

Достигнув барьера, все потоки, кроме последнего, ожидают внутри метода synchronized, пока последний поток не «разбудит» их. После «пробуждения» один поток выполнения берет на себя управление монитором. Остальные потоки продолжают соревноваться за монитор. Количество участвующих потоков должно быть известно. В противном случае используется механизм регистрации участников.

Лабораторная работа № 4

Тема работы: ***Синхронизация потоков в Java.***

Цель работы:

Основная цель данной работы — изучение и применение механизмов синхронизации потоков выполнения (thread-ов) путем реализации классической задачи «производитель–потребитель» с использованием различных методов и классов языка для управления конкурентным доступом к общим ресурсам.

Задачи работы:

1. Понимание концепции конкуренции и синхронизации в многопоточных приложениях.

2. Реализация общего ресурса (общего буфера) между производителями и потребителями.
3. Применение методов синхронизации, таких как `wait()`, `notify()`, `notifyAll()` и блоки `synchronized`.
4. Приобретение навыков проектирования и тестирования стабильных и масштабируемых многопоточных приложений.

Пример реализации:

```
import java.util.ArrayList;

public class Main {

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

        Store store=new Store();

        Producer p1 = new Producer(store);
        p1.setDaemon(true);
        p1.setName("Производитель №1");

        Producer p2 = new Producer(store);
        p2.setDaemon(true);
        p2.setName("Производитель №2");

        Производитель p3 = новый Производитель(хранилище);
        p3.setDaemon(true);
        p3.setName("Производитель №3");

        Потребитель c1 = новый Потребитель(магазин);
        c1.setName("Потребитель №1");

        Потребитель c2 = новый Потребитель(хранилище);
        c2.setName("Потребитель №2");

        Потребитель c3 = новый Потребитель(хранилище);
        c3.setName("Потребитель №3");

        Consumer c4 = new Consumer(store);
        c4.setName("Потребитель №4");

        p1.start();
        p2.start();
        p3.start();
        c1.start();
        c2.start();
        c3.start();
        c4.start();
        while(c1.isAlive() || c2.isAlive() || c3.isAlive() || c4.isAlive()){ }
        System.out.println("Все потоки завершены");
    }
}

class Store {

    ArrayList<Integer> stockList=new ArrayList<Integer>();
```

```
    public synchronized void get(String str) {
        while (stockList.size()<1) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        System.out.println(str + " взял со склада: "+
stockList.get(stockList.size()-1));
        stockList.remove(stockList.size()-1);
        if(stockList.size()!=0){
            System.out.print("На складе имеется " + stockList.size()+" единиц -> ");
            for(int нечетный : stockList){
                System.out.print(нечетный+ " ");
            }
            System.out.println(" ");}
        else{
            System.out.println("Склад пуст");
        }
        notifyAll();
    }

    public synchronized void put(String str,int a, int b) {
        while (stockList.size()>=5) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        System.out.print(str + " поместил в хранилище два числа: ");
        stockList.add(a);
        System.out.print(stockList.get(stockList.size()-1)+", ");
        stockList.add(b);
        System.out.println(stockList.get(stockList.size()-1));
        if(stockList.size()!=0){
            System.out.print("На складе имеется " + stockList.size()+" единиц -> ");
            for(int impar : stockList){
                System.out.print(нечетное+ " ");
            }
            System.out.println(" ");}
        else{
            System.out.println("Склад пуст");
        }
        notifyAll();
    }
}

class Producer extends Thread{

    Store s;

    public Producer(Store s) {
        this.s = s;
    }
    @Override
    public void run() {
        int[] нечетные = new int[]{1,3,5,7,9,11,13,15,17,19};
        while(true){
```

```
            s.put(getName(), нечетные[(int)(Math.random()*9)],
нечетные[(int)(Math.random()*9)]);
        }
    }
}

class Consumer extends Thread{

    Store s;
    public Consumer(Store s) {
        this.s = s;
    }
    @Override
    public void run() {
        int cons = 0;
        for(int i = 0; i < 2; i++){
            s.get(getName());
        }
        System.out.println(getName() + " взял 2 числа. Поток завершен");
    }
}
```

Задачи для выполнения:

Даны X производителей, которые случайным образом генерируют F объектов, которые потребляются Y потребителями. Отобразить информацию о производстве и потреблении объектов, сообщения о случаях, когда «склад пуст или полон». Все операции выполняются до тех пор, пока каждый потребитель не будет удовлетворен Z объектами.

Размер склада — D. Значения для X, Y, Z, D, F указаны в таблице 3.

Таблица 1 Варианты выполнения задачи

№	X	Y	Z	D	Тип Объекты
1	2	3	11	8	Четные числа
2	3	4	2	5	Нечетные числа
3	4	3	3	10	Вокал
4	3	2	12	11	Согласные
5	2	5	3	12	Четные числа
6	2	4	4	7	Нечетные числа
7	3	3	5	6	Вокал
8	4	2	4	5	Согласные
9	5	2	3	4	Четные числа
10	3	3	5	2	Нечетные числа
F - каждый производитель производит по 2 предмета за раз для всех вариантов					

Критерии оценки:

1. Создание и инициализация потоков для выполнения задач.
2. Выбор форм синхронизации потоков.
3. Синхронизация потоков с подходящими формами.
4. Создание графического интерфейса программы.

5. **Корректность кода** — проверка правильности кода , отсутствие ошибок в работе.

6. **Соблюдение инструкций и требований** — проверка правильности требований задачи, таких как количество потоков выполнения.

7. **Оптимизация кода** — оценка эффективности кода в использовании ресурсов и избегание кода.

8. **Соблюдение сроков сдачи** - оценка баллов по пунктуальности, сдана ли работа в установленный срок.

9. **Оценка знаний** — объяснения, данные о процессе выполнения работы, которые могут включать описание основных функций и использованной логики.

10.Использование ИИ студентом.

Для получения оценки 5-6 обязательны критерии 1, 2, 3, 5, 8, 9.

Для получения оценки 7-8 обязательны критерии 1, 2, 3, 4, 5, 6, 8, 9.

Для получения оценки 9-10 обязательны критерии 1-9.

Если был использован критерий 10, оценка снижается на 2 балла, только если студент разъяснил функционирование кода. В противном случае лабораторная работа не принимается.

***Контрольные вопросы*:**

1. Что такое поток (thread) и как он создается в Java?
2. Что такое синхронизация потоков и зачем она нужна?
3. Каковы основные методы синхронизации, предлагаемые классом `Object` в Java?
4. Объясните роль методов `wait()`, `notify()` и `notifyAll()`.
5. Что такое *гонка условий* и как ее можно предотвратить?
6. В каких ситуациях могут возникать *тупиковые ситуации* и *голодание*?
7. Как можно отслеживать и тестировать конкурентное поведение Java-программы?