ЛАБОРАТОРНАЯ РАБОТА №2.
ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»

1-40 05 01 «Информационные системы и технологии (в бизнес-менеджменте и промышленной безопасности)»

ТЕМА: «МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ»

ЦЕЛЬ РАБОТЫ: изучить особенности создания параллельных вычислений, назначение, состояния и свойства потоков исполнения, принципы и механизмы синхронизации, особенности использования процессов, потокобезопасные и небезопасные коллекции.

КЛАСС THREAD И ИНТЕРФЕЙС RUNNABLE

Многопоточная система в *Java* построена на основе класса *Thread*, его методах и дополняющем его интерфейсе *Runnable*. Класс *Thread* инкапсулирует поток исполнения. **Чтобы создать новый поток исполнения, следует расширить класс *Thread* или же реализовать интерфейс *Runnable*.**

В классе *Thread* определяется ряд методов, помогающих управлять потоками исполнения. Некоторые из тех методов перечислены в таблице 1.

Таблица 1 – Основные методы управления потоком исполнения класса *Thread*

Сигнатура метода	Назначение
public final String getName()	Возвращает имя потока исполнения
public final int getPriority()	Возвращает приоритет потока исполнения
public final boolean isAlive()	Определяет, выполняется ли поток
public final void join() throws InterruptedException *public final void join(long millis) throws InterruptedException* *public final void join(long millis, int nanos) throws InterruptedException {*	Ожидает завершения потока исполнения
@Override public void run()	Задает точку входа в поток исполнения
public static void sleep(long millis) throws InterruptedException *public static void sleep(long millis, int nanos) throws InterruptedException*	Приостанавливает выполнение потока на заданное время
Thread start(Runnable task); *public void start()* *void start(ThreadContainer container)*	Запускает поток исполнения, вызывая его метода *run()*

Во всех рассмотренных до сих пор темах и примерах программ использовался единственный поток исполнения, который запускался методом *main()*. Рассмотрим, как пользоваться классом *Thread* и интерфейсом *Runnable* для создания потоков исполнения и управления ими, начиная с главного потока, присутствующего в каждой программе на *Java*.

ГЛАВНЫЙ ПОТОК ИСПОЛНЕНИЯ ПРИЛОЖЕНИЯ

Когда программа на *Java* запускается на выполнение, сразу же начинает выполняться один поток исполнения. Он обычно называется **главным потоком**

программы, потому что он запускается вместе с программой. Главный поток исполнения важен по двум причинам.

– от этого потока исполнения порождаются все дочерние потоки.

– главный поток исполнения должен быть **последним** потоком, завершающим выполнение программы, поскольку в нем производятся различные завершающие действия.

Несмотря на то, что главный поток исполнения создается автоматически при запуске программы, им можно управлять через объект класса *Thread*. Для этого достаточно получить ссылку на главный поток, вызвав метод *currentThread()*, который объявляется как открытый и статический (*public static*) в классе *Thread*. Его общая форма выглядит следующим образом:

```
@IntrinsicCandidate
public static native Thread currentThread();
```

Этот метод возвращает ссылку на поток исполнения, из которого он вызван. Получив ссылку на главный поток, можно управлять им таким же образом, как и любым другим потоком исполнения.

Рассмотрим следующий пример программы.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
//Example №1. Управление главным потоком исполнения программы
public class CurrentThreadUsing {
    public static void main(String args[]) {
        Thread thread = Thread.currentThread();
        System.out.println("Текущий поток исполнения: " + thread);
        // изменение имени текущего потока исполнения
        thread.setName("My Thread");
        System.out.println("Новое имя потока:" + thread);
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
                System.out.println("Время: " +
DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                        .format(LocalDateTime.now()));
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток исполнения был
прерван: " + DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                    .format(LocalDateTime.now()));
        }   }}
```

Результат работы программы:

```
Текущий поток исполнения: Thread[#1,main,5,main]
Новое имя потока:Thread[#1,My Thread,5,main]
5
Время: 05 02 2023, 11:40:54 PM
4
Время: 05 02 2023, 11:40:55 PM
3
Время: 05 02 2023, 11:40:56 PM
2
Время: 05 02 2023, 11:40:57 PM
1
Время: 05 02 2023, 11:40:58 PM
```

В этом примере программы ссылка на текущий поток исполнения (в данном случае – главный поток) получается в результате вызова метода *currentThread()* и сохраняется в локальной переменной *thread*. Затем выводятся сведения о потоке исполнения (имя, приоритет и группа, к которой относится поток). Далее вызывается метод *setName()* для изменения имени потока исполнения. После этого сведения о потоке исполнения опять выводятся на консоль (имя, приоритет и группа, к которой относится поток). Далее в цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется с помощью вызова метода *sleep()*. Аргумент метода *sleep()* задает время задержки выполнения потока в миллисекундах.

Обратите внимание на блок операторов *try/catch*, в котором находится цикл вывода чисел на экран. Метод *sleep()* из класса *Thread* может сгенерировать исключение типа *InterruptedException*, если в каком-нибудь другом потоке исполнения потребуется прервать этот ожидающий поток. В данном примере просто выводится сообщение, если поток исполнения прерывается, а в реальных программах подобную ситуацию придется обрабатывать иначе, реагируя на произошедшее прерывание.

Обратите внимание на то, что при выводе данных о потоке с использованием метода *println()* используется переменная потока *thread*. Метод *toString()* для экземпляра потока возвращает имя потока исполнения, его приоритет и имя группы, к которой относится поток. По умолчанию главный поток исполнения имеет имя *main* и приоритет, равный 5. Именем *main* обозначается также группа потоков исполнения, к которой относится данный поток.

Потоки объединяются в группы потоков (*thread groups*) по соображениям улучшения управляемости и безопасности. Потоки, относящиеся к одной группе, могут управляться единовременно – можно прервать работу сразу всех потоков группы либо установить для них единое максимальное значение приоритета выполнения.

Метод *sleep()* вынуждает тот поток, для которого он вызывается, приостановить свое выполнение на указанное количество миллисекунд. Общая форма этого метода выглядит следующим образом:

```
public static void sleep(long millis) throws InterruptedException {
```

Количество миллисекунд, на которое нужно приостановить выполнение, задает аргумент *millis*. Метод *sleep()* может сгенерировать исключение типа *InterruptedException*. У него имеется и вторая форма, которая позволяет точнее задавать время ожидания в мили- и наносекундах.

```
public static void sleep(long millis, int nanos) throws
InterruptedException {
```

Вторая форма данного метода может применяться только в тех средах, где предусматривается задание промежутков времени в наносекундах.

В примере №1 показано, что установить имя потока исполнения можно с помощью метода *setName()*. А для того, чтобы получить имя потока исполнения, достаточно вызвать метод *getName()*. Оба эти метода являются элементами класса *Thread* и объявляются так, как показано ниже, где аргумент *name* обозначает имя потока исполнения.

```
public final synchronized void setName(String name);
public final String getName();
```

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА RUNNABLE

Можно создать отдельный поток исполнения, имплементировав интерфейс *Runnable*. Этот интерфейс предоставляет абстракцию единицы исполняемого кода. Поток исполнения можно создать из объекта любого класса, реализующего интерфейс *Runnable*. Для реализации интерфейса *Runnable* в классе должен быть объявлен единственный метод *run():*

```
public void run();
```

В теле метода *run()* определяется код, который представляет новый поток исполнения. В методе *run()* можно вызывать другие методы, создавать объекты, объявлять переменные таким же образом, как и в главном потоке исполнения. **Единственное отличие заключается в том, что в методе *run()* устанавливается точка входа в другой, параллельный поток исполнения в приложении.** Этот поток исполнения завершится, когда метод *run()* возвратит управление или завершит свое тело.

После создания класса, реализующего интерфейс *Runnable*, в этом классе следует получить экземпляр объекта типа *Thread*. Для этой цели в классе *Thread* определен ряд конструкторов. Тот конструктор, который должен использоваться в данном случае, выглядит в общей форме следующим образом:

```
public Thread(Runnable task, String name)
```

В этом конструкторе параметр *task* обозначает экземпляр класса, реализующего интерфейс *Runnable*. Этим определяется место, где начинается выполнение потока. Имя нового потока исполнения передается данному конструктору в качестве параметра *name*.

После того как новый поток исполнения будет создан, он не запускается до тех пор, пока не будет вызван метод *start()*, объявленный в классе *Thread*. По существу, в методе *start()* вызывается метод *run()*. Ниже показано, каким образом объявляется метод *start()*.

```
public void start();
```

Рассмотрим следующий пример, демонстрирующий создание и запуск нового потока на выполнение.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
//Example №2. Создание второго потока исполнения
public class NewThread implements Runnable {
    Thread thread;
    NewThread() {
        // создать новый поток исполнения
        thread = new Thread(this, "Демонстрационный поток");
        System.out.println("Дочерний поток создан:" + thread);
        thread.start(); // запустить поток исполнения
    }
    // Точка входа во второй поток исполнения
    @Override
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток:" + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Дочерний поток прерван." +
DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                    .format(LocalDateTime.now()));
        }
        System.out.println("Дочерний поток завершен."+
DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                .format(LocalDateTime.now()));
    }}
class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Главный поток:" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
```

```
            System.out.println("Главный поток прерван." +
DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                    .format(LocalDateTime.now()));
        }
        System.out.println("Главный поток завершен." +
DateTimeFormatter.ofPattern("dd MM yyyy, hh:mm:ss a")
                .format(LocalDateTime.now()));
    }}
```

Новый объект класса *Thread* создается в следующем операторе в конструкторе *NewThread():*

```
thread = new Thread(this, "Демонстрационный поток");
```

Далее в для него вызывается метод *start(),* в результате чего поток исполнения запускается, начиная с метода *run().* Это, в свою очередь, приводит к началу цикла *for* в дочернем потоке исполнения. После вызова метода *start()* конструктор *NewThread()* возвращает управление методу *main().* Возобновляя свое исполнение, главный поток входит в свой цикл *for*. Далее потоки выполняются параллельно, совместно используя ресурсы процессора, вплоть до завершения своих циклов.

Результат работы программы (он может оказаться иным в зависимости от конкретной исполняющей среды):

```
Дочерний поток создан:Thread[#23,Демонстрационный поток,5,main]
Главный поток:5
Дочерний поток:5
Дочерний поток:4
Дочерний поток:3
Главный поток:4
Дочерний поток:2
Дочерний поток:1
Главный поток:3
Дочерний поток завершен.07 02 2023, 03:06:38 PM
Главный поток:2
Главный поток:1
Главный поток завершен.07 02 2023, 03:06:40 PM
```

В многопоточной программе главный поток исполнения должен завершаться последним. На самом же деле, если главный поток исполнения завершается раньше дочерних потоков, то исполняющая система *Java* может "зависнуть", что характерно для некоторых старых виртуальных машин *JVM*. В приведенном примере программы гарантируется, что главный поток исполнения завершится последним, поскольку главный поток исполнения находится в состоянии ожидания в течение 1000 миллисекунд в промежутках между последовательными шагами цикла, а дочерний поток исполнения – только 500

миллисекунд. Это заставляет дочерний поток исполнения завершиться раньше главного потока.

РАСШИРЕНИЕ КЛАССА THREAD

Еще один способ создать поток исполнения состоит в том, чтобы сначала объявить класс, расширяющий класс *Thread*, а затем получить экземпляр этого класса. В расширяющем классе должен быть переопределен метод *run()*, который является точкой входа в новый поток исполнения. Кроме того, в этом классе должен быть вызван метод *start()* для запуска нового потока на исполнение. Приведена версия программы из предыдущего примера реализует создание потока с использованием наследования класса *Thread*.

```
//Example №3. Создать поток исполнения, расширив класс Thread
class NewThread extends Thread {
    NewThread() {
// создать новый поток исполнения
        super("Демонстрационный поток");
        System.out.println("Дочерний поток:" + this);
        start(); // запустить поток на исполнение
    }
    // Точка входа в другой поток исполнения
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток:" + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Дочерний поток прерван.");
        }
        System.out.println("Дочерний поток завершен.");
    }}
class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток исполнения
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Главный поток:" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван. ");
        }
        System.out.println("Главный поток завершен. ");
    }}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. Дочерний поток исполнения создается при создании объекта класса *NewThread*, наследующего класс *Thread*. Обратите внимание на метод *super()* в классе *NewThread*. Он вызывает конструктор *Thread()* базового класса *Thread*, общая форма которого приведена ниже, где параметр *name* обозначает имя создаваемого потока исполнения.

```
public Thread(String name);
```

СОЗДАНИЕ НЕСКОЛЬКИХ ПОТОКОВ ИСПОЛНЕНИЯ

В приложении может быть создано сколько угодно потоков исполнения. Например, в следующей программе создаются три дочерних потока исполнения:

```
//Example №4. Создание несколько потоков исполнения
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread thread;
    NewThread(String threadName) {
        name = threadName;
        thread = new Thread(this, name);
        System.out.println("Новый поток: " + thread);
        thread.start(); // запустить поток на исполнение
    }
    // Точка входа в поток исполнения
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ":" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван");
        }
        System.out.println(name + " завершен.");
    }}
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("The first"); // создать поток
        new NewThread("The second");
        new NewThread("The third");
        try {
// ожидать завершения других потоков исполнения
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }
        System.out.println("Главный поток завершен.");
    }}
```

Результат работы программы:

```
Новый поток: Thread[#23,The first,5,main]
Новый поток: Thread[#24,The second,5,main]
Новый поток: Thread[#25,The third,5,main]
The first:5
The second:5
The third:5
The first:4
The second:4
The third:4
The first:3
The third:3
The second:3
The first:2
The second:2
The third:2
The first:1
The second:1
The third:1
The first завершен.
The third завершен.
The second завершен.
Главный поток завершен.
```

После запуска на исполнение все три дочерних потока совместно используют общие ресурсы центрального процессора. Обратите внимание на вызов метода *sleep (10000)* в методе *main()*. Это вынуждает главный поток перейти в состояние ожидания на 10 секунд и гарантирует его завершение последним.

ПРИМЕНЕНИЕ МЕТОДОВ ISALIVE() И JOIN()

Часто требуется, чтобы главный поток исполнения завершался последним. С этой целью метод *sleep()* вызывался в предыдущих примерах из метода *main()* с достаточной задержкой, чтобы все дочерние потоки исполнения завершились раньше главного. **В классе *Thread* предоставляются инструменты, позволяющее определить** был ли поток исполнения завершен. Для этого можно вызвать метод *isAlive()*, определенный в классе *Thread*. Ниже приведена общая форма этого метода.

```
public final boolean isAlive();
```

Метод *isAlive()* возвращает значение *true*, если поток, для которого он вызван, еще исполняется. В противном случае он возвращает значение *false*.

В классе *Thread* имеется метод *join()*, который применяется для того, чтобы дождаться завершения другого потока исполнения. Ниже приведены формы этого метода.

```
public final void join() throws InterruptedException;
public final void join(long millis, int nanos) throws
InterruptedException;
public final void join(long millis) throws InterruptedException
public final boolean join(Duration duration) throws
InterruptedException;
```

Метод *join()* ожидает завершения того потока исполнения, для которого он вызван. Имя этого метода отражает следующий принцип: вызывающий поток ожидает, когда указанный поток присоединится к нему. Дополнительные формы метода *join()* позволяют указывать максимальный промежуток времени, в течение которого требуется ожидать завершения указанного потока исполнения.

Ниже приведена версия программы из предыдущего примера, где с помощью метода *join()* гарантируется, что главный поток завершится последним. В данном примере демонстрируется также применение метода *isAlive()*.

```
//Example №5. Применить метод join(), чтобы ожидать завершения
потоков исполнения
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread thread;
    NewThread(String threadName) {
        name = threadName;
        thread = new Thread(this, name);
        System.out.println("Новый поток:" + thread);
        thread.start(); // запустить поток исполнения
    }
    // Точка входа в поток исполнения
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ":" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }}
class DemoJoin {
    public static void main(String args[]) {
        NewThread thread1 = new NewThread("The first");
        NewThread thread2 = new NewThread("The second");
        NewThread thread3 = new NewThread("The third");
        System.out.println("Поток Один запущен:" +
thread1.thread.isAlive());
        System.out.println("Поток Два запущен:" +
thread2.thread.isAlive());
        System.out.println("Поток Три запущен: " +
thread3.thread.isAlive());
```

```
// ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков. ");
            thread1.thread.join();
            thread2.thread.join();
            thread3.thread.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван ");
        }
        System.out.println("Поток Один запущен:" +
thread1.thread.isAlive());
        System.out.println("Поток Два запущен:" +
thread2.thread.isAlive());
        System.out.println("Поток Три запущен:" +
thread3.thread.isAlive());
        System.out.println("Главный поток завершен. ");
    }}
```

Результат работы программы (у вас он может оказаться иным в зависимости от конкретной исполняющей среды):

```
Новый поток:Thread[#23,The first,5,main]
Новый поток:Thread[#24,The second,5,main]
Новый поток:Thread[#25,The third,5,main]
Поток Один запущен:true
Поток Два запущен:true
Поток Три запущен: true
Ожидание завершения потоков.
The third:5
The second:5
The first:5
The second:4
The first:4
The third:4
The third:3
The second:3
The first:3
The third:2
The second:2
The first:2
The second:1
The first:1
The third:1
The second завершен.
The first завершен.
The third завершен.
Поток Один запущен:false
Поток Два запущен:false
Поток Три запущен:false
Главный поток завершен.
```

Потоки прекращают исполнение после того, как вызовы метода *join()* возвращают управление.

ПРИОСТАНОВКА, ВОЗОБНОВЛЕНИЕ И ОСТАНОВКА ПОТОКОВ ИСПОЛНЕНИЯ

Иногда возникает потребность в **приостановке** исполнения потоков. Например, отдельный поток исполнения может служить для отображения времени. Если пользователю не требуется отображение текущего времени, то этот поток исполнения можно приостановить. Выполнение приостановленного потока может быть **возобновлено**. Механизм временной или окончательной остановки потока исполнения, а также его возобновления отличался в ранних версиях *Java*. До версии *Java* 2 методы *suspend()* и *resume()*, определенные в классе *Thread*, использовались в программах для приостановки и возобновления потоков исполнения. На первый взгляд применение этих методов кажется вполне правильным подходом к управлению выполнением потоков. Тем не менее пользоваться им в новых программах на *Java* не рекомендуется, так как метод *suspend()* из класса *Thread* был объявлен не рекомендованным к употреблению. Это было сделано потому, что иногда он способен порождать серьезные системные сбои.

```
@Deprecated(since="1.2", forRemoval=true)
public final void suspend();
```

Допустим, что поток исполнения заблокировал важные структуры данных. **Если в этот момент приостановить исполнение данного потока, то блокировки структур данных не будут сняты** и другие потоки исполнения, ожидающие эти ресурсы, могут оказаться также заблокированными.

Метод *resume()* также не рекомендован к употреблению. И хотя его применение не вызовет особых осложнений, тем не менее им нельзя пользоваться без метода *suspend()*, который его дополняет.

```
@Deprecated(since="1.2", forRemoval=true)
public final void resume();
```

Метод *stop()* из класса *Thread* также объявлен устаревшим с версии *Java* 2. Это было сделано потому, что он может иногда послужить причиной серьезных системных сбоев.

```
@Deprecated(since="1.2", forRemoval=true)
public final void stop();
```

Предположим, что поток выполняет запись в структуру данных и успел выполнить запись лишь частично. **Если его остановить в этот момент, то структура данных может оказаться в поврежденном состоянии.** Дело в том, что метод *stop()* вызывает снятие любой блокировки, устанавливаемой вызывающим потоком исполнения. Следовательно, поврежденные данные могут быть использованы в другом потоке исполнения.

Вместо использования методов *suspend()*, *resume()*, *stop()* код управления выполнением потока в текущей версии *Java* должен быть составлен таким образом, чтобы метод *run()* периодически проверял, должно ли исполнение потока быть приостановлено, возобновлено или прервано. Обычно для этой цели служит переменная-флаг, обозначающая состояние потока исполнения. До тех пор, пока эта переменная-флаг содержит признак "выполняется", метод *run()* должен продолжать выполнение.

Если же эта переменная содержит признак "приостановить", то поток исполнения должен быть приостановлен. А если переменная-флаг получает признак "остановить", то поток исполнения должен завершиться.

В приведенном ниже примере программы демонстрируется применение методов *wait()* и *notify()*, унаследованных от класса *Object*, для управления выполнением потока.

```
//Example №6. Использование методов wait() и notify()
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread thread;
    boolean isSuspended;
    NewThread(String threadname) {
        name = threadname;
        thread = new Thread(this, name);
        System.out.println("Новый поток:" + thread);
        isSuspended = false;
        thread.start(); // запустить поток исполнения
    }
    public void run() {
        try {
            for (int i = 8; i > 0; i--) {
                System.out.println(name + ":" + i);
                Thread.sleep(200);
                synchronized (this) {
                    while (isSuspended) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }
    synchronized void suspend() {
        isSuspended = true;
    }
    synchronized void resume() {
        isSuspended = false;
        notify();
    }
}
public class SuspendResume {
```

```
    public static void main(String args[]) {
        NewThread thread1 = new NewThread("Один");
        NewThread thread2 = new NewThread("Два");
        try {
            Thread.sleep(1000);
            thread1.suspend();
            System.out.println("Приостановка потока Один");
            Thread.sleep(1000);
            thread1.resume();
            System.out.println("Возобновление потока Один");
            thread2.suspend();
            System.out.println("Приостановка потока Два");
            Thread.sleep(1000);
            thread2.resume();
            System.out.println("Возобновление потока Два");
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }
// ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков.");
            thread1.thread.join();
            thread2.thread.join();
        } catch (InterruptedException ex) {
            System.out.println("Главный поток прерван.");
        }
        System.out.println("Главный поток завершен");
    }
}
```

Класс *NewThread* содержит переменную экземпляра *isSuspended* типа *boolean*, используемую для управления выполнением потока. В конструкторе этого класса она инициализируется логическим значением *false*. Метод *run()* содержит блок оператора *synchronized*, где проверяется состояние переменной *suspendFlag*. Если она принимает логическое значение *true*, то вызывается метод *wait()* для приостановки выполнения потока.

В методе *suspend()* устанавливается логическое значение *true* переменной *isSuspended*, а в методе *resume()* – логическое значение *false* этой переменной и вызывается метод *notify()*, чтобы активизировать поток исполнения. В методе *main()* вызываются методы *suspend()* и *resume()*. В результате запуска программы, видно, как исполнение потоков приостанавливается и возобновляется. Результат работы программы:

```
Новый поток:Thread[#23,Один,5,main]
Новый поток:Thread[#24,Два,5,main]
Один:8
Два:8
Два:7
Один:7
Два:6
Один:6
Два:5
Один:5
Один:4
Два:4
Приостановка потока Один
Два:3
Два:2
Два:1
Два завершен.
Возобновление потока Один
Приостановка потока Два
Один:3
Один:2
Один:1
Один завершен.
Возобновление потока Два
Ожидание завершения потоков.
Главный поток завершен
```

ПОЛУЧЕНИЕ СОСТОЯНИЯ ПОТОКА ИСПОЛНЕНИЯ

Поток исполнения может находиться в нескольких состояниях. Для того чтобы получить текущее состояние потока исполнения, достаточно вызвать метод *getState()*, определенный в классе *Thread*, следующим образом:

```
public State getState()
```

Этот метод возвращает значение типа *Thread.State*, обозначающее состояние потока исполнения на момент вызова. Перечисление *State* определено в классе *Thread*. Значения, которые может возвратить метод *getState()*, приведены на экране.

Таблица – Значения, возвращаемые методом *getState()*

Значение	Состояние
NEW	Состояние потока потока, который еще не запущен.
RUNNABLE	Поток выполняется или ожидает другие ресурсы от операционной системы, например, процессорнное время.

BLOCKED	Поток заблокирован в ожидании монитора объекта. Поток ожидает освобождение монитора, чтобы войти в синхронизированный блок или метод
WAITING	Состояние ожидания. Поток находится в состоянии ожидания из-за вызова одного из следующих методов: *Object.wait()* без указания времени *Thread.join()* без указания времени *LockSupport.park()* Поток в состоянии ожидания ожидает, пока другой поток выполнит определенное действие. Например, поток, который вызвал *Object.wait()* для объекта, ожидает, что другой поток вызовет *Object.notify()* или *Object.notifyAll()* для этого объекта. Поток, вызвавший Thread.join(), ожидает завершения указанного потока.
TIMED_WAITING	Состояние ожидания потока с заданным временем ожидания. Поток находится в состоянии ожидания по времени из-за вызова одного из следующих методов с заданным положительным временем ожидания: *Thread.sleep()* *Object.wait()* без указания времени *Thread.join()* без указания времени *LockSupport.parkNanos()* *LockSupport.parkUntil()*
TERMINATED	Поток завершил выполнение

Имея в своем распоряжении экземпляр класса *Thread*, можно вызвать метод *getState()*, чтобы получить состояние потока исполнения. Например, в следующем фрагменте кода определяется, находится ли поток исполнения *thread* в состоянии *RUNNABLE* во время вызова метода *getState():*

```
NewThread thread2 = new NewThread("Два");
if (thread2.thread.getState() == Thread.State.RUNNABLE) //...
```

Состояние потока исполнения может измениться после вызова метода *getState()*. **Состояние, полученное при вызове метода *getState()*, мгновение спустя может уже не отражать фактическое состояние потока исполнения.** По этой и другим причинам метод *getState()* **не предназначен** для синхронизации потоков исполнения. Он служит прежде всего для отладки во время выполнения программы.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Назовите виды многозадачности в операционных системах.
2. Что такое "процесс"? Что такое "поток" ("нить", "thread")?
3. Назовите возможные состояния потоков исполнения.
4. Что такое приоритет выполнения потока и для чего он предназначен? Как получить и как установить значение приоритета выполнения потока?
5. Что такое переключение контекста?
6. Что такое монитор объекта?
7. Как создать отдельный поток исполнения?
8. Дайте определение понятию “синхронизация потоков”.
9. В каких случаях целесообразно создавать несколько потоков?

10. Какой метод запускает поток на выполнение?

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.
2. Ответить на контрольные вопросы лабораторной работы.
3. Разработать алгоритм программы по индивидуальному заданию.
4. Написать, отладить и проверить корректность работы созданной программы.
5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. диаграмму созданных классов в нотации UML
8. выводы по лабораторной работе

ВО ВСЕХ ЗАДАНИЯХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. ВСЕ РЕШАЕМЫЕ ЗАДАЧИ ДОЛЖНЫ БЫТЬ РЕАЛИЗОВАНЫ, ИСПОЛЬЗУЯ КЛАССЫ И ОБЪЕКТЫ.

В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО:

Необходимо разработать многопоточное консольное приложение на языке программирования *Java*. Смоделировать работу согласно варианту задания. Все параметры заданий необходимо вводить с клавиатуры (например, 1.5 т., 30 мин и т.п.). Рекомендуется время в минутах и часах в задачах заменить на пропорциональные единицы в миллисекундах.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Имеется склад с тремя воротами: для маленьких (3 тонны), средних (5 тонн) и больших (15 тонн) грузовиков. Всего надо вывести 186 т. груза. Отгрузка из одних ворот склада осуществляется так, что как только один грузовик загружен полностью, переходят к погрузке следующего грузовика. Время погрузки одной тонны груза составляет 3 мин. Также требуется время, чтобы предыдущий грузовик уехал, а следующий заехал (2 мин, 2 мин и 5 мин соответственно). В больших воротах после обслуживания каждых 10 машин требуется тех обслуживание в течении 1 мин. Какое время потребуется для погрузки всего груза и сколько груза погрузят через каждые ворота. Результаты выполненных действий выводятся на экран и в файл.

2. На склад подъезжают грузовики для погрузки груза. Всего надо вывезти 86 т. груза. Всего у фирмы 3 грузовика. У каждого грузовика своя грузовместимость (1,5 т., 3,4 т. и 5.1 т., соответственно), также каждый грузовик характеризуется своим временем, затрачиваемым на доставку груза со склада (30 мин., 45 мин., 40 мин.). Время погрузки одной тонны груза составляет 3 мин. Ворота не могут простаивать, то есть после окончания погрузки на один грузовик сразу же переходят к погрузке другого грузовика. Какое время потребуется для перевозки всего груза и количество рейсов, которое сделает каждый грузовик. Результаты выполненных действий выводятся на экран и в файл.

3. Для обслуживания клиентов в банке работает электронная очередь. Всего четыре кассира. Клиенты подходят к тому кассиру, который освободился. У каждого кассира своя средняя скорость обслуживания одного клиента (2мин., 2,5 мин., 2,4 мин. и 3 мин., соответственно). Всего кассирам надо обслужить 150 клиентов. Какое время потребуется для обслуживания всех клиентов и количество клиентов, которое обслужит каждый кассир. Результаты выполненных действий выводятся на экран и в файл.

4. Для обслуживания покупателей в магазине работает три кассы. У каждого кассира на кассе своя средняя скорость обслуживания одного покупателя (3 мин., 2,5 мин. и 2,7 мин., соответственно). Покупатели выстраиваются в очереди к кассам таким образом, что количество покупателей в каждую кассу одинаковое. Приложение должно выводить количество покупателей, которое обслужат все кассиры за 60 мин. и выводить количество покупателей, которое обслужит каждый кассир за 60 мин. Результаты выполненных действий выводятся на экран и в файл.

5. В порту одновременно работает три погрузчика. Им надо загрузить контейнеры на корабли. Всего надо погрузить 3000 контейнеров. Грузоподъемность каждого из них 15, 13 и 10 контейнеров, а время погрузки составляет 12 мин., 11 мин. и 11 мин. Чтобы встать на место погрузки у каждого из погрузчиков кораблю требуется 5 мин., 6 мин. и 5 мин. соответственно. Приложение должно рассчитать и вывести время, за которое все контейнеры будут погружены, и количество контейнеров, которое погрузит каждый погрузчик. Результаты выполненных действий выводятся на экран и в файл.

6. Надо перевезти 3000 контейнеров на кораблях. Контейнеры перевозят четыре корабля. Вместимость каждого из них 15, 14, 13 и 10 контейнеров, а время погрузки одного контейнера составляет 2 мин. Время доставки погруженных контейнеров к месту назначения для каждого корабля одинаковое и составляет 40 мин. Приложение должно рассчитать и вывести время, за которое все контейнеры будут вывезены и количество рейсов, которое сделает каждый корабль. Результаты выполненных действий выводятся на экран и в файл.

7. На бензозаправку приезжают автомобили, чтобы заправиться топливом. Всего на заправке четыре колонки. Среднее время заправки одной машины 6 мин. Наблюдается такое количество машин, желающих заправиться, что колонки не простаивают. Через каждые 45 мин. работы колонки закрываются на техническое обслуживание: первая– на 10 мин, вторая – на 15 мин., третья – на 5 мин., четвертая– на 13 мин. Приложение должно выводить количество автомобилей,

которое будет обслужено всеми колонками за 4 часа и выводить количество автомобилей, которое обслужит каждая колонка за 4 часа. Результаты выполненных действий выводятся на экран и в файл.

8. Диспетчерская такси принимает заказы на перевозку пассажиров. Всего работает 10 автомобилей такси. Водители такси через каждые три часа работы делают 15 минутный перерыв. Время доставки одного пассажира случайно в пределах от 5 до 30 мин. для каждого такси. Приложение должно выводить количество вызовов, которое было обслужено всеми такси за 7 ч и одним такси за 6 ч. Результаты выполненных действий выводятся на экран и в файл.

9. В поликлинике в регистратуре выдаются талоны на прием к врачу. Всего работает два регистратора. Всего надо выдать 100 талонов к разным специалистам. Первый регистратор обслуживает одного посетителя за 2 мин., второй за 1,5 мин. Приложение должно выводить время, за которое были обслужены все посетители и количество талонов, которые были выданы каждым регистратором. Результаты выполненных действий выводятся на экран и в файл.

10. В театральной кассе продаются билеты. Работает три кассы. Первый кассир обслуживает одного покупателя за 2 мин., второй – за 2,5 мин., третий – за 2,8 мин. Всего надо обслужить 250 человек. Через каждые 45 мин. кассиры закрывают кассу на 10 минутный технический перерыв. Приложение должно выводить время, за которое были обслужены все покупатели и количество покупателей, обслуженные каждым кассиром. Результаты выполненных действий выводятся на экран и в файл.

11. В гардеробе работает 3 работника. Первый работник обслуживает одного посетителя за 1 мин., второй за 0,5 мин. и третий за 0,8 мин. Всего надо обслужить 450 человек. Приложение должно выводить время, за которое были обслужены все посетители и количество посетителей, которое было обслужено каждым работником. Результаты выполненных действий выводятся на экран и в файл.

12. На склад с некоторой фирмы подъезжают грузовики для погрузки груза. Всего надо вывести 145 т. груза. У склада только одни ворота. Всего у фирмы 3 грузовика. У каждого грузовика своя грузовместимость (1,5 т., 3,4 т. и 5.1 т., соответственно), также каждый грузовик характеризуется своим временем, затрачиваемым на доставку груза со склада (30 мин., 45 мин., 40 мин.). Отгрузка из ворот склада осуществляется так, что пока один грузовик не загружен полностью, не переходят к погрузке следующего грузовика. Время погрузки одной тонны груза составляет 3 мин. Приложение должно выводить общее время, за которое весь груз будет вывезен со склада и количество рейсов, которое сделает каждый грузовик. Результаты выполненных действий выводятся на экран и в файл.

13. В порту работает погрузчик. Ему надо загрузить контейнеры на корабли. Всего надо погрузить 3000 контейнеров. Контейнеры перевозят четыре кораблями, вместимостью 15, 14, 13 и 10 контейнеров. Время погрузки одного контейнера составляет 2 мин. Погрузчик может грузить контейнер только на один корабль (сразу на несколько кораблей не может). Время доставки погруженных контейнеров к месту назначения для каждого корабля одинаковое

и составляет 40 мин. Приложение должно выводить время, за которое все контейнеры будут вывезены и количество рейсов, которое сделает каждый корабль. Результаты выполненных действий выводятся на экран и в файл.

14. Пользователь вводит с клавиатуры значение в массив целых чисел. После чего запускаются три потока исполнения. Первый поток находит максимум в массиве, второй — минимум, третий – второй по величине элемент в массиве. Результаты вычислений возвращаются в метод *main()* и выводятся в файл.

15. Пользователь вводит с клавиатуры текст. После чего запускаются три потока исполнения. Первый поток находит тексте количество вхождения указанного слова, второй — реверсирует содержимое массива, третий – считает количество символов в массиве. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

16. Пользователь вводит значения полей для объектов типа *File*. После чего запускаются три потока исполнения. Первый поток находит наборе файлов наибольший по размеру, второй — все файлы с казанным расширением, третий – все файлы, дата создания которых позже указанной. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

17. Создать класс *Matrix* (матрица), в котором реализовать методы для работы с матрицами: перемножение матриц, вычитание матриц. Пользователь вводит значения полей для объектов класса. После чего запускаются три потока исполнения. Первый поток находит в матрице наибольший элемент, второй — матрицу с наибольшим количеством столбцов, третий – матрицу с наибольшим количеством отрицательных элементов.

18. Создать класс *Blog* (блог). Пользователь вводит значения полей для объектов типа *Blog*. После чего запускаются три потока исполнения. Первый поток находит в наборе блогов все блоги с указанной тематикой, второй — все блоги, дата создания которых позже указанной, третий – блог, с наибольшим количеством подписчиков. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

19. Создать класс *MobileApp* (мобильное приложение). Пользователь вводит значения полей для объектов типа *MobileApp*. После чего запускаются три потока исполнения. Первый поток находит в наборе мобильных приложений все приложения с указанной тематикой, второй — все приложения, дата создания которых позже указанной, третий – приложение, с наибольшим размером установочного файла. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

20. Создать класс *PetStore* (зоомагазин). Пользователь вводит значения полей для объектов типа *PetStore*. После чего запускаются три потока исполнения. Первый поток находит в наборе зоомагазинов все магазины с указанной районом расположения, второй — все магазины, дата открытия которых позже указанной, третий – магазины, с наличием указанного товара. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

21. Создать класс *Bot* (бот). Пользователь вводит значения полей для объектов типа *Bot*. После чего запускаются три потока исполнения. Первый поток находит в наборе ботов все приложения с указанной тематикой, второй — все боты, дата создания которых позже указанной, третий – боты, с наибольшим количеством пользователей. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

22. Создать класс *Control* (элемент управления). Пользователь вводит значения полей для объектов типа *Control*. После чего запускаются три потока исполнения. Первый поток находит в наборе элементов управления все элементы с указанной характеристикой, второй — все элементы, у которых нет обработчиков, третий – элементы управления указанного типа. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

23. Создать класс *Configuration* (конфигурация компьютера). Пользователь вводит значения полей для объектов типа *Configuration*. После чего запускаются три потока исполнения. Первый поток находит в наборе конфигураций все конфигурации с указанной характеристикой, второй — все элементы, у которых нет всех необходимых данных, третий – элементы указанного типа. Результаты выполненных действий возвращаются в метод *main()* и выводятся в файл.

24.