ГЛАВА 1

1. Онлайн-платформа для веб-хостинга GIT.

Git — это бесплатная система контроля версий с открытым исходным кодом, разработанная для быстрой и эффективной работы с любыми проектами — от небольших до очень крупных. Git прост в освоении, занимает мало места и отличается высокой производительностью. Он превосходит такие SCM (Supply Chain Management — управление цепочкой поставок) инструменты, как Subversion, CVS, Perforce и ClearCase, благодаря таким характеристикам, как локальные недорогие ветвления, удобные хранилища и множественные рабочие процессы.

1.1 Ветвление и слияние

Особенность Git, которая действительно выделяет его среди почти всех других SCM, — это модель ветвления. Git позволяет и поощряет создание нескольких локальных веток, которые могут быть полностью независимыми друг от друга. Создание, объединение и удаление этих веток занимает всего несколько секунд.

Это означает, что вы можете делать такие вещи, как:

• Бесконтактное переключение контекста. Создайте ветвь, на которой можно опробовать любую идею, сделайте несколько коммитов, вернитесь к месту, где была создана ветвь, примените исправление, вернитесь к месту, где были проведены эксперименты, и объедините их.

• Линии кода для ролевых игр. У вас есть ветка, которая всегда содержит только то, что отправлено в производство, другая, в которой работа объединяется для тестирования, и несколько небольших веток для повседневной работы.

• Рабочий процесс, основанный на функциях. Создавайте новые ветки для каждой новой функции, над которой вы работаете, чтобы можно было легко переключаться между ними, а затем удаляйте каждую ветку, когда функция будет объединена с основной веткой.

• Уникальные эксперименты. Создайте ветку для экспериментов, поймите, что она не работает, и удалите ее, уйдите с работы, и никто больше ее не увидит (даже если за это время вы продвинули другие ветки).

1.2 Распределение

Одной из самых полезных функций любого распределенного SCM, включая Git, является его распределение. Это означает, что вместо того, чтобы выполнять «проверку» вершины текущего исходного кода, вы выполняете «клонирование» всего репозитория.

1.3 Множественные резервные копии

Это означает, что даже если вы используете централизованный рабочий процесс, каждый пользователь фактически имеет полную резервную копию основного сервера.

Каждая из этих копий может быть использована для замены основного сервера в случае сбоя или повреждения. По сути, в Git нет единой точки отказа, если только не существует единственной копии репозитория.

1.4 Рабочий процесс

Благодаря распределенному характеру Git и отличной системе ветвления можно относительно легко реализовать практически бесконечное количество рабочих процессов.

1.5 Рабочий процесс в стиле Subversion

Централизованный рабочий процесс очень распространен, особенно среди пользователей, которые переходят с централизованной системы. Git не позволит вам выполнить push, если кто-то выполнил push после последнего извлечения, поэтому централизованная модель, в которой все разработчики выполняют push на один и тот же сервер, работает просто великолепно.

1.6 Установка Git

Скачайте программу установки (https://git-scm.com/downloads), запустите программу установки, выберите нужные опции (или оставьте все как есть, если не очень хорошо разбираетесь), дождитесь завершения процесса установки — и все готово.

1.7 Использование Git

В рабочей папке, где находятся исходные файлы приложения/проекта, через терминал или командную строку используйте команду git init (официальная документация - https://git-scm.com/docs/git-init) для инициализации локального репозитория, после чего Git будет отслеживать изменения в исходных файлах и фиксировать их — используйте команду git commit (официальная документация — https://git scm.com/docs/git-commit).

В будущем, если вам нужно синхронизировать изменения в общем репозитории для всех репозиториев, используйте команду git push

2. Потоки выполнения

2.1. Что такое ветка?

Потоки выполнения обеспечивают переход от последовательного программирования к параллельному программированию. Последовательная программа представляет собой классическую модель программы: она имеет начало, последовательность выполнения своих инструкций и конец. Другими словами, в данный момент времени программа имеет только одну точку выполнения. Выполняемая программа называется *процессом*. Однозадачная операционная система (MS-DOS) выполняет только один процесс в данный момент времени, в то время как многозадачная операционная система (UNIX, Windows) может запускать множество процессов одновременно (конкурентно), периодически выделяя каждому процессу часть рабочего времени процессора. Поскольку понятие потока выполнения имеет смысл только в рамках многозадачной операционной системы. Поток выполнения похож на последовательный процесс — он имеет начало, последовательность выполнения и конец. Разница между потоком выполнения и процессом заключается в том, что поток выполнения не может работать независимо, а должен работать в рамках процесса.

а) б)

Рисунок 1. Потоки выполнения в рамках программы

- Программа с одним потоком выполнения;

На рис. 1 показано расположение потоков выполнения в рамках программы.

Определение : *Поток выполнения* — это последовательность инструкций, выполняемых в рамках процесса.

В программе может быть определено несколько потоков выполнения, что означает, что в рамках одного процесса могут одновременно выполняться несколько потоков выполнения, что позволяет одновременное выполнение независимых задач этой программы. Поток выполнения можно сравнить с уменьшенной версией процесса, оба из которых работают одновременно и независимо на последовательной структуре выполнения своих инструкций. Кроме того, одновременное выполнение потоков выполнения в рамках одного процесса сходно с параллельным выполнением процессов: операционная система будет циклически выделять доли времени процессора каждому потоку выполнения до их завершения. По этой причине потоки выполнения еще называют *легкими процессами.* Основное отличие между потоком выполнения и процессом заключается в том, что потоки выполнения могут работать в рамках одного процесса. Другое отличие заключается в том, что каждый процесс имеет свою собственную память (свой собственный адресный пространство), а при создании нового процесса (fork) создается точная

копия родительского процесса: код + данные; при создании потока выполнения копируется только код родительского процесса; все потоки выполнения имеют доступ к одним и тем же данным исходного процесса. Таким образом, поток выполнения можно рассматривать и как *контекст выполнения* в рамках родительского процесса. Потоки выполнения полезны во многих отношениях, но обычно они используются для выполнения трудоемких операций без блокирования основного процесса: математических вычислений, ожидания освобождения ресурса, которые обычно выполняются в фоновом режиме.

2.2. Конкуренция потоков

Исходя из особенностей аппаратного обеспечения, следует отметить, что процессоры могут выполнять только одну инструкцию за раз. На многопроцессорной машине потоки выполняются на разных процессорах в одно и то же физическо , даже если процессоры находятся на разных компьютерах, подключенных к сети. Но и на машине с одним процессором потоки могут «разделять» один и тот же процессор, выполняясь в переплетенном режиме, а конкуренция за время CPU создает иллюзию, что они выполняются одновременно. Эта иллюзия кажется реальной, когда 30 отдельных изображений в секунду, улавливаемых человеческим глазом, воспринимаются как непрерывный поток изображений. Переключение между потоками также имеет свою «цену»: оно потребляет время CPU, необходимое для замораживания состояния одного потока и размораживания состояния другого потока (смена контекста). Если конкурирующие потоки выполняются на одном процессоре и все выполняют вычисления, то общее время выполнения не будет превышать время выполнения последовательной программы, выполняющей то же самое. Увеличение скорости системы достигается за счет пересечения различных фаз различных потоков. Многие задачи могут быть логически разделены на типы фаз: фаза вычислений и фаза ввода-вывода.

Фаза вычислений требует максимального внимания со стороны ЦП за счет использования различных методов вычислений. Фаза ввода-вывода (I/O) требует максимального внимания со стороны периферийных устройств (принтера, жесткого диска, сетевой карты и т. д.), и в этих ситуациях процессор, как правило, свободен, ожидая, пока периферийное устройство завершит свою задачу. Увеличение скорости достигается путем пересечения фаз. В то время как один поток находится в фазе ввода-вывода, ожидая загрузки последовательности данных с жесткого диска, поток с фазой вычисления может занять процессор, и когда он достигает фазы ввода-вывода, другой поток (который только что завершил свою фазу ввода-вывода) может начать использовать CPU.

2.3. Создание потоков выполнения

Как и любой другой объект в Java, поток выполнения является экземпляром класса. Потоки выполнения, определенные классом, будут иметь одинаковый код и, следовательно, одинаковую последовательность инструкций. Создание класса, определяющего потоки выполнения, может быть осуществлено двумя способами:

- **путем расширения класса Thread**
- **путем реализации интерфейса Runnable**

Любой класс, экземпляры которого будут выполняться в потоке выполнения, должен быть объявлен как Runnable. Это интерфейс, содержащий единственный метод, а именно метод «run». Таким образом, любой класс, описывающий потоки выполнения, будет содержать метод **run()**, в котором реализован код, который будет выполняться потоком выполнения. Интерфейс Runnable разработан как общий протокол для объектов, которые хотят выполнять код в течение своего существования (которые представляют собой потоки выполнения). Наиболее важным классом, реализующим интерфейс Runnable, является класс Thread. Класс Thread реализует общий поток выполнения, который по умолчанию ничего не делает. Другими словами, метод run не содержит никакого кода. Любой поток выполнения является экземпляром класса Thread или его подкласса.

2.3.1 Расширение класса Thread

Самый простой способ создать поток выполнения, который что-то делает, — это расширить класс Thread и перегрузить его метод run. Общий формат такого класса:

```
public class SimpleThread extends Thread {
   public SimpleThread(String имя) {
     super(имя);
    //вызываю конструктор суперкласса Thread
   }

   public void run() {
   //код, выполняемый потоком выполнения
   }
}
```

Первый метод класса — это конструктор, который получает в качестве аргумента строку, которая будет представлять имя потока выполнения, созданного в момент вызова конструктора.

```
SimpleThread t = new SimpleThread("Java");
//создает поток выполнения с именем Java
```

Если мы не хотим давать имена создаваемым потокам выполнения, мы можем отказаться от определения этого конструктора и оставить только конструктор по умолчанию без аргументов, который создает поток выполнения без имени. Впоследствии его можно назвать с помощью метода setName(String).

Можно определить и другие конструкторы, которые будут полезны, когда мы захотим отправить различные параметры потоку выполнения.

Второй метод — это метод run, «сердце» любого потока выполнения, в котором мы фактически пишем код, который должен выполнять поток выполнения. Созданный поток выполнения не запускается автоматически, его запуск осуществляется с помощью метода start, также определенного в классе Thread.

```
SimpleThread t = new SimpleThread("Java");
t.start();
//создает и запускает поток выполнения
```

Далее рассмотрим пример, в котором мы определяем поток выполнения, отображающий целые числа из интервала с определенным шагом. Поток выполнения реализуется классом Counter.

```
class Counter extends Thread {
 //класс, определяющий поток выполнения
 private int from, to, step;

 public Counter(int from, int to, int step) {
       this.from = from;
       this.to = to;
       this.step = step;
 }

 public void run() {
       for(int i = from; i &lt;= to; i += step)
             System.out.print(i + " " );
 }
}

public class TestCounter { //главный класс
 public static void main(String args[]) {
       Counter cnt1, cnt2;
```

```
        cnt1 = new Counter(0, 10, 2);
        //считает от 0 до 100 с шагом 5

        cnt2 = new Counter(100, 200, 10);
        //считает от 100 до 200 с шагом 10

        cnt1.start();
        cnt2.start();
//запускаем потоки выполнения
//они будут автоматически уничтожены по завершении
  }
}
```

Последовательное выполнение этой программы сначала отобразит числа от 0 до 100 с шагом 5, а затем числа от 100 до 200 с шагом 10, поскольку первый вызов направлен на счетчик cnt1, поэтому результат, отображаемый на экране, должен быть следующим: 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 100 110 120 130 140 150 160 170 180 190 200.

Однако в реальности полученный результат будет представлять собой интерполяцию значений, произведенных двумя нитями выполнения, работающими одновременно. При разных прогонах могут быть получены разные результаты, поскольку время, выделенное каждой нити выполнения, может быть разным, так как оно контролируется процессором «кажущимся» случайным образом: 0 100 5 110 10 120 15 130 20 140 25 150 160 170 180 190 200 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100…

2.3.2. Реализация интерфейса Runnable

Но что делать, если мы хотим создать класс, который инстанциирует потоки выполнения, а у него уже есть суперкласс, зная, что в Java не допускается множественное наследование?

```
class FirExecuţie extends Părinte, Thread //незаконно!
```

В этом случае мы не можем расширить класс Thread, а должны напрямую реализовать интерфейс Runnable в нашем классе. Класс Thread реализует интерфейс Runnable, и по этой причине при его расширении мы получаем имплицитную реализацию интерфейса. Таким образом, интерфейс Runnable позволяет классу быть активным, не расширяя класс Thread.

Интерфейс Runnable находится в пакете java.lang и определяется следующим образом:

```
public interface Runnable {
        pulic abstract void run( );
}
```

Следовательно, класс, который создает экземпляры потоков выполнения путем реализации интерфейса Runnable, должен обязательно реализовывать метод run. Общий формат класса, реализующего интерфейс Runnable, следующий:

```
public class SimpleThread implements Runnable {

  private Thread simpleThread = null;

  public SimpleThread() {
        if (simpleThread == null) {
        simpleThread = new Thread(this);
              simpleThread.start();
        }
        public void run() {
        //код, выполняемый потоком выполнения
  }
}
```

В отличие от предыдущего подхода, теряется вся поддержка, предоставляемая классом Thread для создания потока выполнения. Простое создание экземпляра класса, реализующего интерфейс Runnable, не создает никакого потока выполнения. По этой причине создание потоков выполнения путем создания экземпляра такого класса должно производиться явно. Это делается путем объявления объекта типа Thread в качестве переменной-члена соответствующего класса. Этот объект будет представлять собой собственно поток выполнения, код которого находится в классе:

```
private Thread simpleThread = null;
```

Следующим шагом является инстанциирование и инициализация потока выполнения. Это осуществляется с помощью инструкции new, за которой следует вызов конструктора, который принимает в качестве аргумента экземпляр класса. После создания поток выполнения может быть запущен вызовом метода start. (Эти операции обычно записываются в конструкторе класса, чтобы выполняться при инициализации экземпляра, но могут быть записаны в любом месте в теле класса или даже вне его)

```
simpleThread = new Thread( this );
simpleThread.start();
```

Указание аргумента **this** в конструкторе класса Thread приводит к созданию потока выполнения, который при запуске будет искать в нашем классе метод run и выполнять его. Этот конструктор принимает в качестве аргумента любой экземпляр класса «Runnable». Таким образом, метод run не должен вызываться явно, что происходит автоматически при вызове метода **start()**.

Явный вызов метода run не вызовет никаких ошибок, но он будет выполняться как любой другой метод, то есть не в отдельном потоке. Давайте перепишем предыдущий пример (отображение целых чисел из диапазона с определенным шагом) с использованием интерфейса Runnable. Обратите внимание, что реализация интерфейса Runnable обеспечивает большую гибкость при работе с потоками выполнения.

Вариант 1 (стандартный)

Создание потока выполнения осуществляется в конструкторе класса Counter:

```
class Counter implements Runnable {
  private Thread counterThread = null;
  private int from, to, step;
  public Counter(int from, int to, int step) {
      this.from = from;
      this.to = to;
      this.step = step;
      if (counterThread == null) {
         counterThread = new Thread(this);
         counterThread.start();
      }
  }
  public void run() {
     for(int i = from; i &lt;= to; i += step)
        System.out.print(i + " " );
  }
}
public class TestThread2 {
  public static void main(String args[]) {
        Counter cnt1, cnt2;
```

```
//запускаю первый поток выполнения (через конструктор)
        cnt1 = new Counter(0, 100, 5);
//запускаю второй поток выполнения (через конструктор)
        cnt2 = new Counter(100, 200, 10);
  }
}
```

Вариант 2

```
Создание потока выполнения осуществляется вне класса Counter:
class Counter implements Runnable {
  private int from, to, step;
  public Counter(int from, int to, int step) {
        this.from = from;
        this.to = to;
        this.step = step;
  }
  public void run() {
        for(int i = from; i &lt;= to; i += step)
              System.out.print(i + " " );
  }
}
public class TestThread2 {
  public static void main(String args[]) {
        Counter cnt1, cnt2;
        cnt1 = new Counter(0, 100, 5);
        cnt2 = new Counter(100, 200, 10);
        new Thread( cnt1 ).start();
        //запускаю первый поток выполнения
        new Thread( cnt2 ).start();
        //запускаю второй поток выполнения
}}
```

2.4. Жизненный цикл потока выполнения

Каждый поток выполнения имеет свой собственный жизненный цикл: он создается, становится активным при запуске и в определенный момент заканчивается. Далее мы более подробно опишем состояния, в которых может находиться поток выполнения. Диаграмма, которая в общем виде иллюстрирует эти состояния, а также методы, вызывающие переход из одного состояния в другое, представлена на рисунке 2:

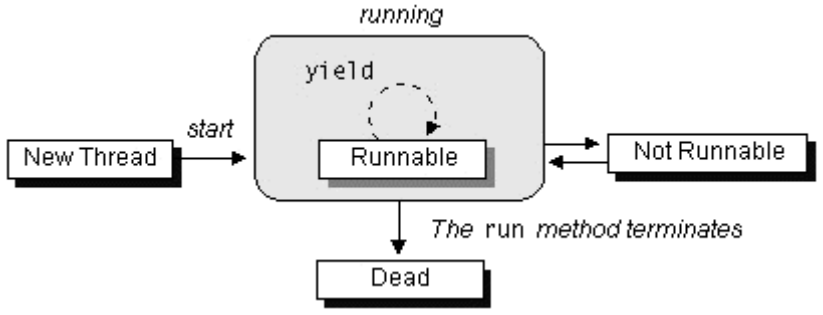

Рисунок 2. Состояния потока выполнения

Таким образом, поток выполнения может находиться в одном из следующих четырех состояний:

1. Состояние «New Thread»

Поток выполнения находится в этом состоянии сразу после его создания, то есть после инстанциирования объекта из класса Thread или его подкласса.

```
Thread counterThread = new Thread ( this );
//counterThread находится в состоянии New Thread
```

2. Состояние «Runnable»

После вызова метода start поток выполнения переходит в состояние «Runnable», то есть находится в процессе выполнения.

```
counterThread.start();
//counterThread находится в состоянии Runnable
```

Метод start выполняет следующие операции, необходимые для запуска потока выполнения: выделяет необходимые системные ресурсы, помещает поток выполнения в очередь ожидания CPU для запуска, вызывает метод run объекта, представленного потоком выполнения.

3. Состояние «Not Runnable»

Поток выполнения попадает в это состояние в одной из следующих ситуаций: он «засыпает» при вызове метода **sleep;** он вызвал метод **wait**, ожидая выполнения определенного условия; он заблокирован в операции ввода/вывода.

«Усыпление» потока выполнения

Метод **sleep** — это статический метод класса Thread, который вызывает паузу во время выполнения текущего потока, то есть «усыпляет» его на определенное время. Длина этой паузы указывается в миллисекундах и даже наносекундах.

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

Поскольку вызов этого метода может вызвать исключения типа InterruptedException, он выполняется в блоке типа try-catch:

```
try {
      Thread.sleep(1000);
      //делает паузу в одну секунду
    } catch (InterruptedException e) {
            . . .
    }.
```

4. Состояние «Dead»

Это состояние, в которое попадает поток выполнения по окончании своего выполнения. Поток выполнения не может быть остановлен из программы с помощью определенного метода, он должен завершиться естественным образом по окончании метода run, который он выполняет. Завершение потока выполнения с помощью переменной завершения:

```
import java.io.*;
public class TestThread {
   public static void main(String args[]) throws
        IOException {
      WaitKey thread = new WaitKey();
      thread.start();
      System.in.read();//ожидание нажатия клавиши Enter
      thread.running = false;
      System.out.println("Прошло " + thread.sec
                                      + " секунд");
  }
}
class WaitKey extends Thread {
   public int sec = 0;
   public boolean running = true;
```

```
   //переменная завершения
public void run() {
  while ( running ) {
        try {
           Thread.sleep(1000);
           sec ++;
           } catch(InterruptedException e){
 }}}}
```

Лабораторная работа 1

Тема работы: Создание потоков

Цели работы:

- Понимание концепции потока и разницы между процессами и потоками выполнения.
- Ознакомление с методами создания и управления потоками в используемом языке программирования.
- Отработка техник взаимодействия между потоками.
- Оценка преимуществ и недостатков многопоточного программирования.
- Формирование практических навыков написания параллельных и конкурентных программ.

Цель работы:

- Понимание принципов конкурентного программирования.
- Практическое освоение способов создания и управления потоками в Java.
- Формирование практических навыков для разработки эффективных и безопасных многопоточных приложений.

Пример реализации:

```
import java.util.*;
class Counter1 extends Thread
{
private int from, to, step;
        private  int[] массив;

public Counter1(int from, int to, int step, int[] tablou) {
     this.from = from;
     this.to = to;
     this.step = шаг;
          this.массив = массив;
}

public void run() {
               int s1=0, s2=0, s=0;
               int i=from;

                    while(i!=to){
                        if(tablou[i]&lt;=50){
                         s1=i;
                         i+=шаг;
                         do{
                          if(tablou[i]&lt;=50){
                          s2=i;
                          s=s1+s2;
      System.out.println(getName()+"  "  +  s1+  "  "  +  s2  +"  "+s+  "  "+
tablou[s1]+" "+tablou[s2] );
                           break;
```

```
                            }
                            i+=step;
                           }while(true);

                          // i+=step;
                           }
                            i+=step;
                      }
                   }
                }

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

                Counter1 cnt1, cnt2;

                int[] массив = new int[101];

                for(int i=0; i&lt;100; i++){
  tablou[i] = (int)(Math.random()*99);
                    System.out.print(массив[i]+" ");
                }
                System.out.println(" ");

cnt1 = new Counter1(0, 99, 1, tablou);

cnt2 = new Counter1(99, 0, -1, tablou);

          cnt1.start();
                  cnt1.setName("Один");
        cnt2.start();
                 cnt2.setName("Два");

}}
```

Результат выполнения:

65 90 74 92 40 41 53 27 88 34 52 70 37 17 0 75 42 56 90 46 37 11 50 77 81 20 49 19 86 65 36 17 46 31 14 87 34 72 29 93 91 84 1 47 98 22 45 41 57 84 70 44 23 12 47 46 6 4 56 81 2 48 11 42 80 54 51 86 16 67 69 36 59 90 9 84 94 69 41 88 27 57 45 33 84 36 10 60 68 70 54 18 26 75 18 81 68 81 80 22

Два 99 94 193 22 18
Один 4 5 9 40 41
Два 92 91 183 26 18
Один 7 9 16 27 34
Два 86 85 171 10 36
Один 12 13 25 37 17
Два 83 82 165 33 45
Один 14 16 30 0 42
Два 80 78 158 27 41
Один 19 20 39 46 37
Два 74 71 145 9 36
Один 21 22 43 11 50
Два 68 63 131 16 42
Один 25 26 51 20 49
Один 27 30 57 19 36
Один 31 32 63 17 46
Два 62 61 123 11 48
Один 33 34 67 31 14
Два 60 57 117 2 4
Один 36 38 74 34 29
Два 56 55 111 6 46
Один 42 43 85 1 47
Два 54 53 107 47 12
Один 45 46 91 22 45
Два 52 51 103 23 44
Один 47 51 98 41 44
Два 47 46 93 41 45
Один 52 53 105 23 12
Один 54 55 109 47 46
Два 45 43 88 22 47
Один 56 57 113 6 4
Два 42 38 80 1 29
Один 60 61 121 2 48
Два 36 34 70 34 14

Один 62 63 125 11 42
Два 33 32 65 31 46
Один 68 71 139 16 36
Два 31 30 61 17 36
Один 74 78 152 9 41
Два 27 26 53 19 49
Один 80 82 162 27 45
Два 25 22 47 20 50
Два 21 20 41 11 37
Один 83 85 168 33 36
Два 19 16 35 46 42
Один 86 91 177 10 18
Два 14 13 27 0 17
Один 92 94 186 26 18
Два 12 9 21 37 34
Два 7 5 12 27 41
BUILD SUCCESSFUL (общее время: 1 секунда)

Этапы выполнения работы:

1. Студенты группы делятся на команды по два человека. Практическая работа будет выполняться в команде.
2. Создайте учетную запись на GitHub (https://github.com/) (если у вас ее еще нет) и отправьте имя пользователя по ссылке в профиле.
3. Установите среду Git на вашем устройстве, подготовьте папку для приложения и инициализируйте локальный репозиторий в этой папке (команда git init).
4. Реализуйте простое приложение (из ***задания***). В команде решите, кто из вас будет реализовывать какие части приложения.
5. После получения инструкций от преподавателя внесите изменения в свою отдельную ветку на GitHub. Зафиксируйте изменения в коде приложения (команда git commit).
6. По завершении работы составьте отчет, который должен содержать: имя, фамилию, группу, задание и ваш вариант выполнения задания, краткое описание, ссылку на исходный код на GitHub. Сохраните отчет в формате PDF или WORD и отправьте на ELSE.

Задание работы:

Задача:

Каждый член команды создает по одному классу с двумя потоками выполнения. Первый поток Th1 отобразит: Условие 1 из таблицы 1. Второй поток Th2 отобразит: Условие 2 из таблицы 1. Все четыре потока выполнения будут считывать данные из одной и той же матрицы данных mas[] типа integer, сгенерированной случайным образом с размером 100 и содержащей значения от 1 до 100. Затем, после завершения обоих потоков выполнения, главный поток отобразит информацию о студентах, выполнивших данную лабораторную работу, буквы текста будут появляться на экране с интервалом 100 миллисекунд.

Таблица 1 Условия для выполнения задачи проблемы в соответствии с вариантами

№/вар	Условие 1	Условие 2
1	Суммы четных чисел по два, начиная поиск и суммирование с первого элемента	Суммы четных чисел по два, начиная поиск и суммирование с последнего элемента
2	Сумма позиций четных чисел по два, начиная поиск и суммирование с первого элемента	Сумма позиций четных чисел по два, начиная поиск и суммирование с последнего элемента
3	Суммы нечетных чисел по два, начиная поиск и суммирование с первого элемента	Суммы нечетных чисел по два, начиная поиск и суммирование с последнего элемента
4	Суммы позиций нечетных чисел по два, начиная поиск и суммирование с первого элемента	Суммы позиций нечетных чисел по два, начиная поиск и суммирование с последнего элемента
5	Суммы произведений чисел по позициям по два, начиная поиск и суммирование с первого элемента	Суммы произведений чисел четных позиций по два, начиная поиск и суммирование с последнего элемента.
6	Суммы произведений чисел на нечетных позициях по два, начиная с поиска и сложения с первым элементом	Суммы произведений чисел на нечетных позициях по два, начиная с поиск и суммирование с последним элементом
7	Суммы четных чисел по два, начиная поиск и суммирование с первого элемента	Суммы произведений четных чисел по два, начиная поиск и суммирование с последнего элемента
8	Суммы произведений нечетных чисел по два, начиная поиск и суммирование с первого элемента	Суммы произведений нечетных чисел по два, начиная поиск и суммирование с последнего элемента
9	Разница произведений четных чисел по два, начиная поиск и суммирование с первого элемента	Разница между числами в позициях появляется по два, начиная с поиска и сложения с последним элементом
10	Разница произведений чисел по нечетным позициям по два, начиная с поиска и сложения с первым элементом	Разница произведений чисел на нечетных позициях по два, начиная с поиска и сложения с последним элементом
11	Разница произведений четных чисел по два, начиная поиск и суммирование с первого элемента	Суммы произведений четных чисел по два, начиная поиск и сложение с последнего элемента
12	Суммы произведений нечетных чисел по два, начиная с , поиск и сложение с первым элементом	Суммы произведений нечетных чисел по два, начиная с , поиск и сложение с последним элементом

Критерии оценки:

1. Регистрация на GitHub и создание локального GIT на хост-компьютере.
2. Создание и инициализация потоков.
3. Создание потоков через интерфейс Runnable.
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 - оценка баллов в зависимости от пунктуальности, если работа была сдана в установленный срок.

9. Оценка знаний - объяснения, данные о процессе выполнения работы, что может включать описание основных функций и использованной логики.

10.Использование студентом ИИ.

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

Для получения оценки 7-8 обязательны критерии 1, 2, 3, 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 работу кода. В противном случае лабораторная работа не принимается.

***Контрольные вопросы*:**

1. Определите нить выполнения?
2. Назовите способы создания потоков выполнения?
3. Как можно определить и реализовать интерфейс Runnable?
4. Назовите состояния потока выполнения?
5. Что такое поток и чем он отличается от программы?
6. Объясните, что такое критический участок.

Список рекомендуемой литературы :

1. Schildt, H. – *Java: Полное руководство*. Издательство Teora, Бухарест, 2012.
2. Deitel, P. J., Deitel, H. M. – *Java – Как программировать*. Издательство Teora, Бухарест, 2008.
3. Pătrașcu, V. – *Конкурентное и параллельное программирование*. Издательство Университета Бухареста, 2015.
4. Станкулеску, А. – *Структуры данных и алгоритмы в Java*. Издательство Polirom, Яссы, 2016.
5. Таненбаум, А. С. – *Современные операционные системы*. Перевод на румынский язык, Издательство Polirom, 2010.
6. Шилдт, Г. – *Java. Полное руководство*. Издательство Вильямс, Москва, 2021.
7. Гослинг, Дж., Арнольд, К., Холмс, Д. – *Язык программирования Java.* СПб: Питер, 2018.
8. Дейтел, П., Дейтел, Х. – *Java. Как программировать*. Москва: Вильямс, 2016.
9. Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Дж. – *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. СПб: Питер, 2019.