|  |
| --- |
| https://lh6.googleusercontent.com/QcftzNtI05T0Y6fjdSh1Rr2rt8oqZ1IvnLvbn1jLJ7CCyteVir3k-xBLv4SL1wAgWJsRhmmJSR0UW-RP63_GQenE4vVWv05BRoZTsmIcBccVTnfxwmsnNMvjg599x9SqZd8E3dkd |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **«МИРЭА - Российский технологический университет»**  **РТУ МИРЭА** |

Институт информационных технологий (ИТ)

Кафедра инструментального и прикладного программного обеспечения (ИиППО)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ** | | | | | |
| **по дисциплине «Разработка клиент-серверных приложений»** | | | | | |
|  | | | | | |
|  | | | | | |
| Выполнил студент группы ИКБО-20-19 | | | | Московка А.А. | |
|  | | | |  | |
| Принял  *Ассистент* | | Зарипов Е.А. | | |
| Практические работы выполнены | «\_\_\_»\_\_\_\_\_\_\_2022 г. | | (подпись студента) | | |
| «Зачтено» | «\_\_\_»\_\_\_\_\_\_\_2022 г. | | (подпись руководителя) | | |
|  |  | |  | | |

Москва 2022

**Практическая работа №1: Примитивы синхронизации и их использование при создании клиент серверных приложений**

**Ход работы:** для решения первой практической работы были написаны класс Main (Рисунок 1.1), три класса для функций (Рисунки 1.2-1.5), классы для реализации с помощью многопоточности (Рисунок 1.6) и с использованием метода ForkJoin (Рисунок 1.7), класс с используемыми инструментами Tools (Рисунок 1.8).

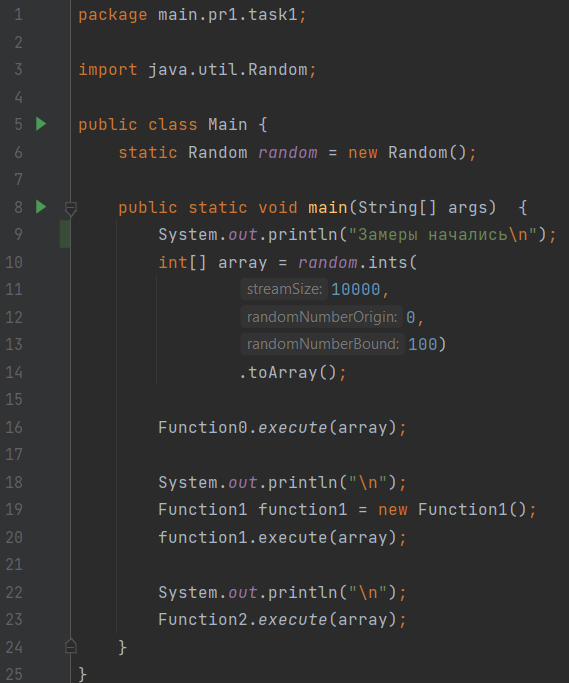


Рисунок 1.1 – Скриншот кода класса Main

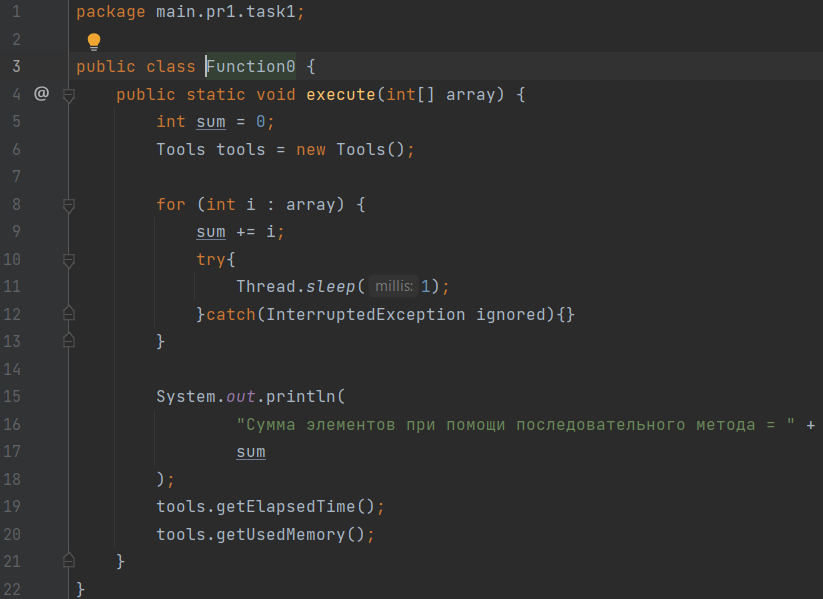


Рисунок 1.2 – Скриншот кода класса императивного решения задачи

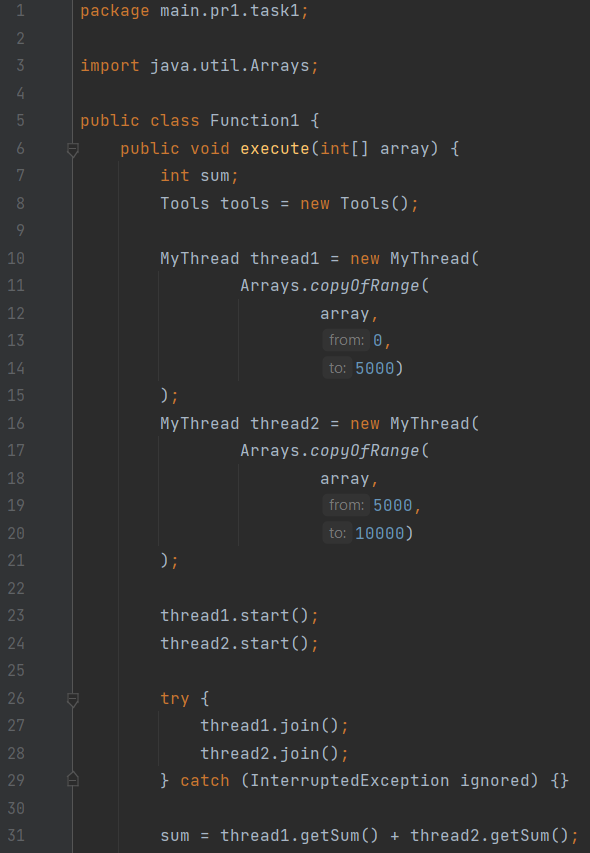


Рисунок 1.3 – Скриншот кода класса для реализации метода многопоточности (Часть 1)

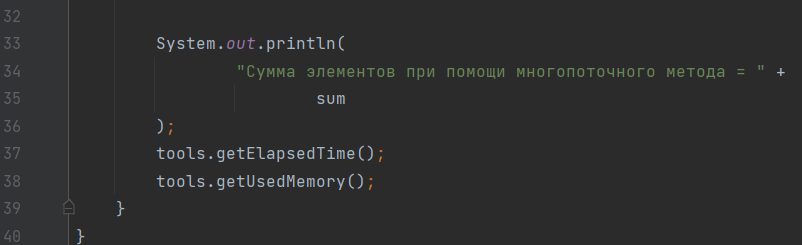


Рисунок 1.4 – Скриншот кода класса для реализации метода многопоточности (Часть 2)

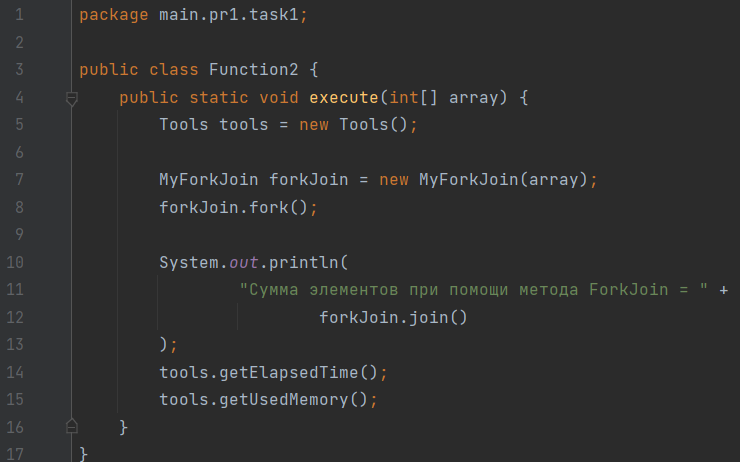


Рисунок 1.5 – Скриншот кода класса для реализации метода ForkJoin



Рисунок 1.6 – Скриншот кода класса наследника Thread

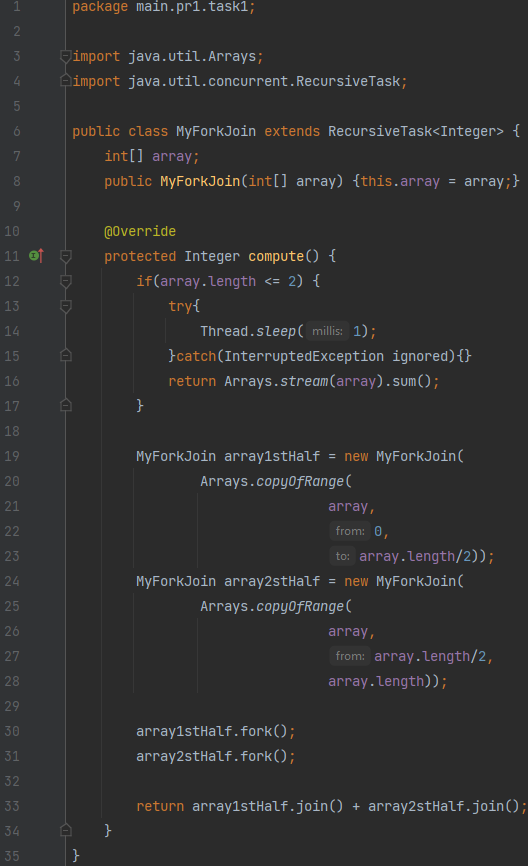


Рисунок 1.7 – Скриншот кода класса наследника для реализации метода ForkJoin



Рисунок 1.8 – Скриншот кода класса вспомогательных методов

После запуска первой программы и ожидания около одной минуты для всех методов выводятся результаты вычислений, что можно увидеть на следующем рисунке (Рисунок 1.9):

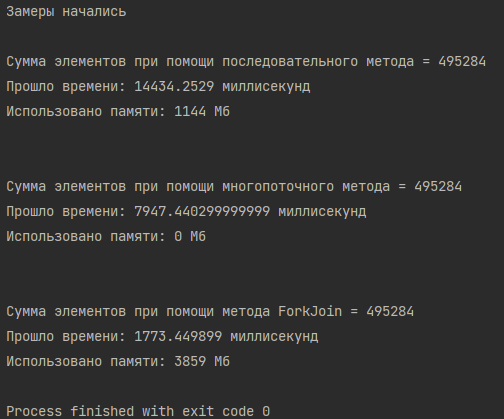


Рисунок 1.9 – Скриншот результата выполнения первой задачи

Затем производился запуск второй программы, в которой вводятся последовательно числа до получения результата вычислений, поскольку вычисления производятся с желаемой задержкой (1-5 секунд), результат выполнения задания представлен на рисунке 1.10.

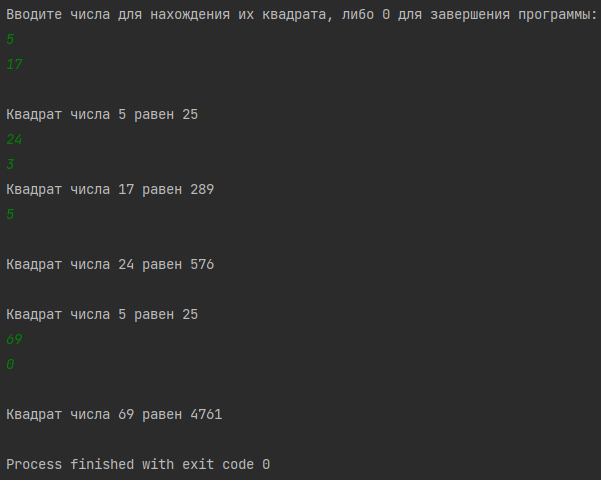


Рисунок 1.10 – Скриншот результата выполнения второй задачи

Наконец, запуск третьей программы производился с целью генерации и дальнейшей обработки ранее сгенерированных файлов в разных потоках, поскольку записей с новых строк больше, чем умещается на скриншоте, результат выполнения третьей задачи представлен на рисунках 1.11-1.12.

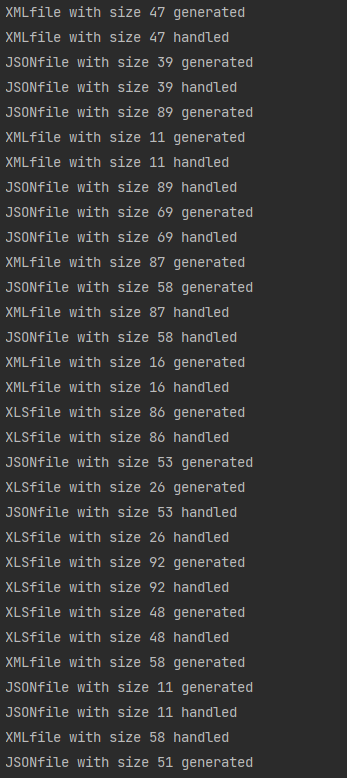


Рисунок 1.11 – Скриншот результата выполнения третьей задачи (Часть 1)

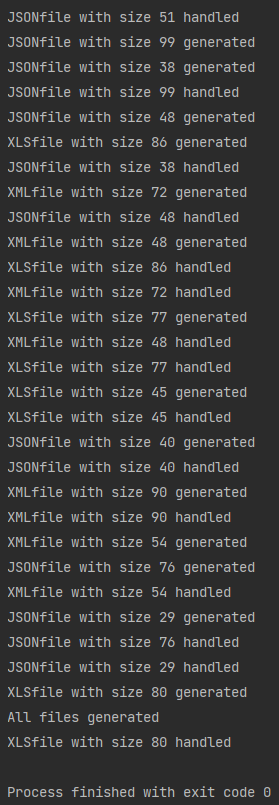


Рисунок 1.12 – Скриншот результата выполнения третьей задачи (Часть 2)

**Практическая работа №2: Java NIO**

**Ход работы:** для решения второй практической работы были написаны класс Main (Рисунок 2.1), четыре класса для функций (Рисунки 2.2-2.7), классы для реализации различных методов копирования во втором задании и вспомогательные интерфейс Example и Timer для подсчета времени (Рисунки 2.8-2.13).

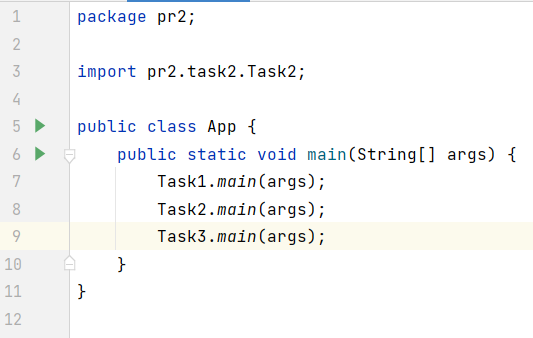


Рисунок 2.1 – Скриншот кода класса Main



Рисунок 2.2 – Скриншот кода класса для решения первого задания



Рисунок 2.3 – Скриншот кода класса для решения второго задания

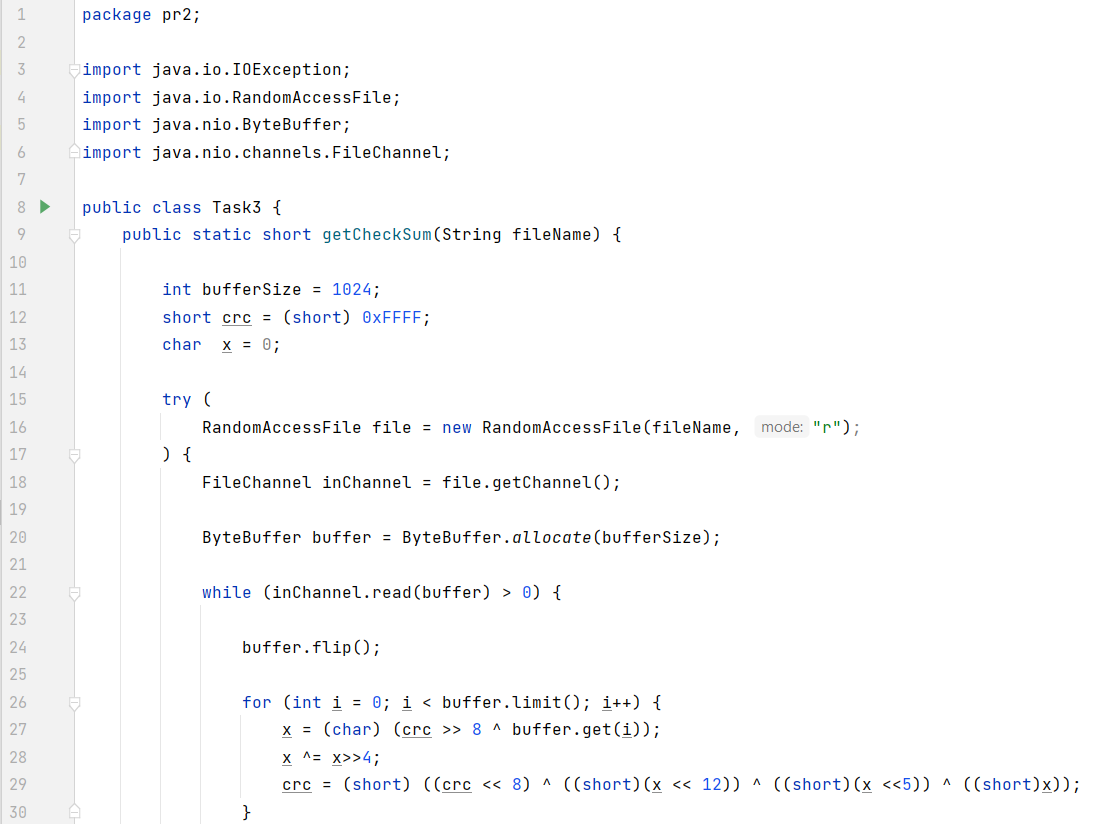


Рисунок 2.4 – Скриншот кода класса для реализации третьей задачи (Часть 1)

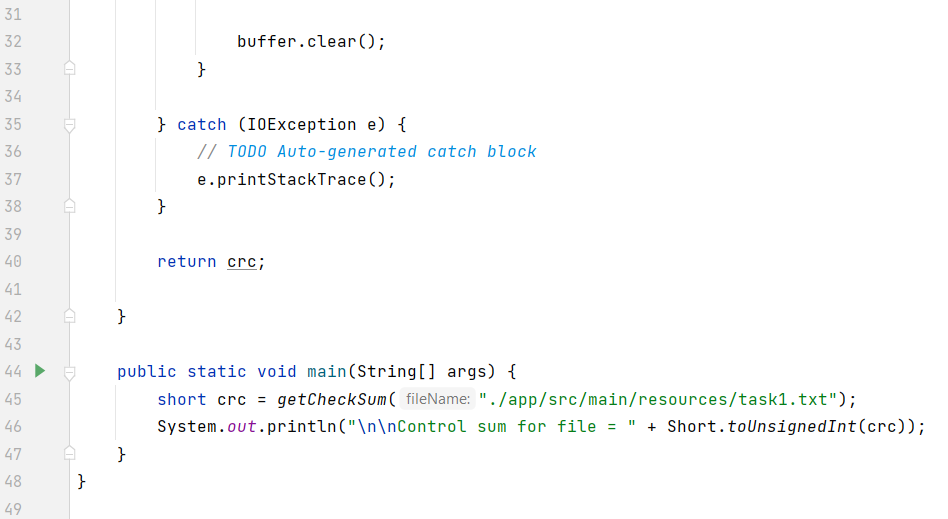


Рисунок 2.5 – Скриншот кода класса для реализации третьей задачи (Часть 2)



Рисунок 2.6 – Скриншот кода класса реализации четвертой задачи (Часть 1)

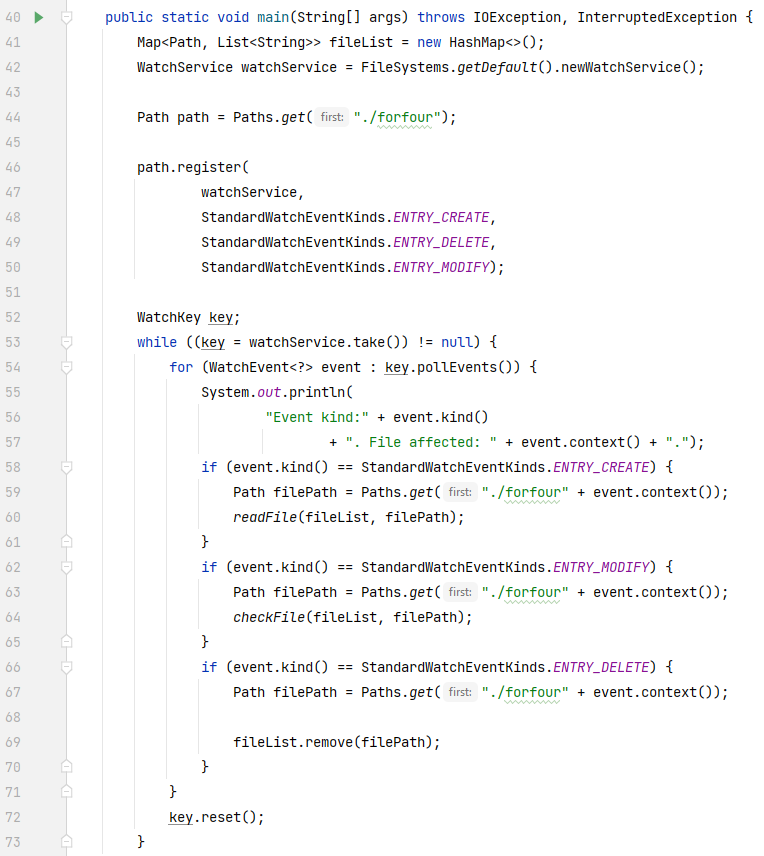


Рисунок 2.7– Скриншот кода класса реализации четвертой задачи (Часть 2)



Рисунок 2.8 – Скриншот кода класса реализации метода Apache Commons IO

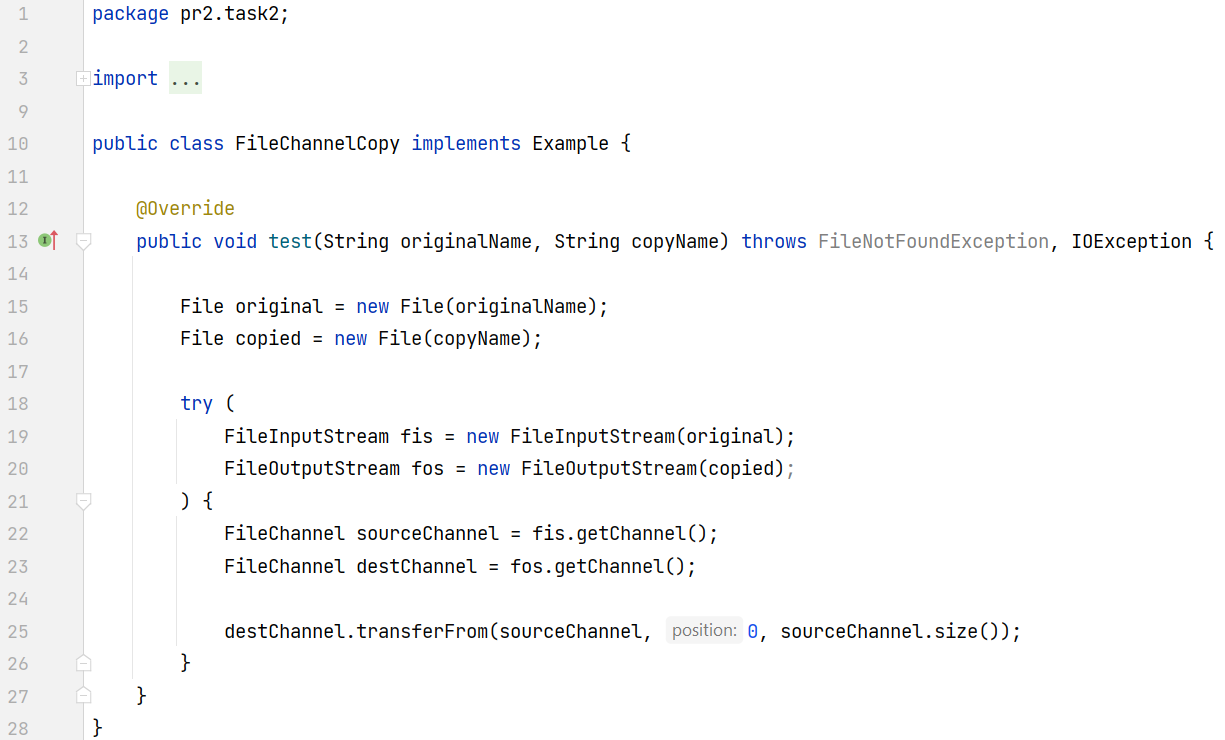


Рисунок 2.9 – Скриншот кода класса реализации метода FileChannel



Рисунок 2.10 – Скриншот кода класса реализации метода File



Рисунок 2.11 – Скриншот кода класса реализации метода FileInputStream/FileOutputStream

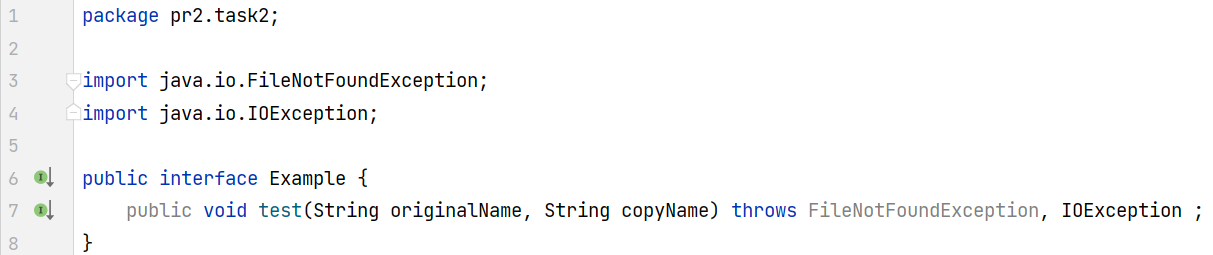


Рисунок 2.12 – Скриншот кода интерфейса Example

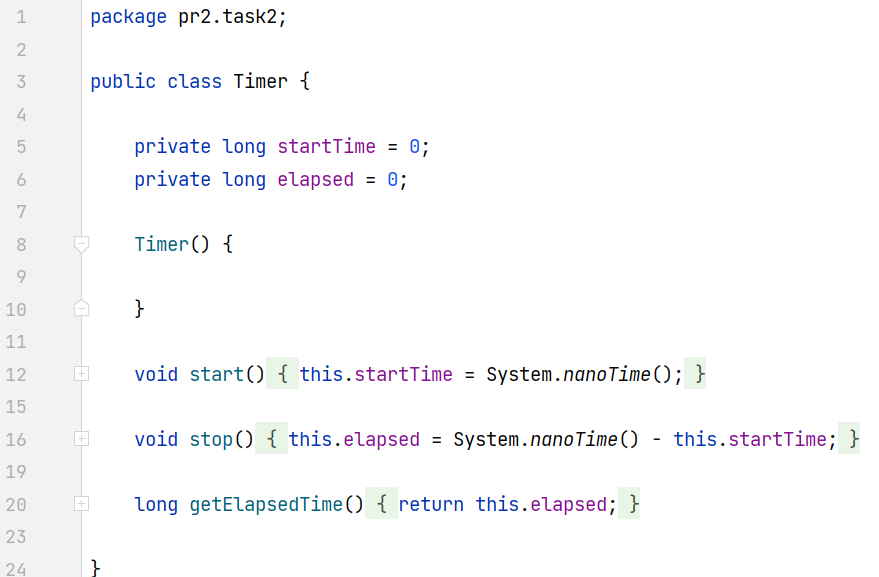


Рисунок 2.13 – Скриншот кода вспомогательного класса Timer

После запуска метода main класса App.java происходит выполнение первого, второго и третьего заданий, что изображено на рисунках 2.14-2.15.

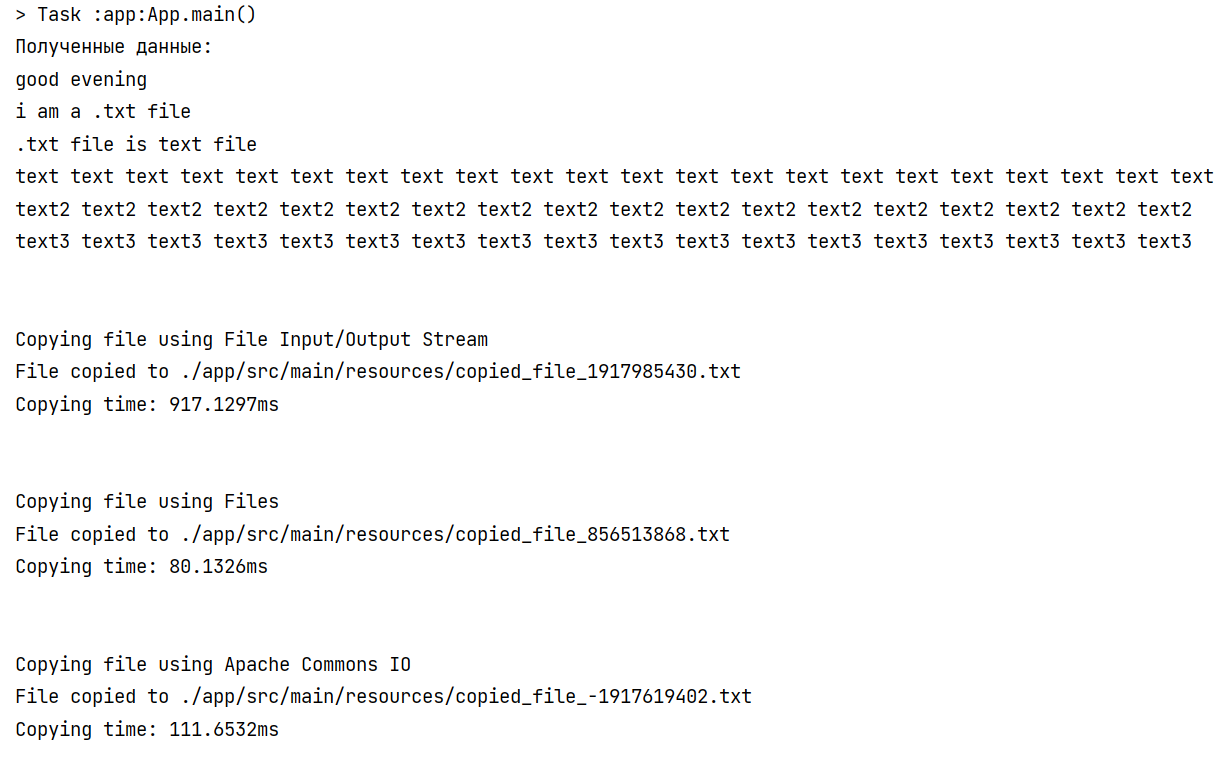


Рисунок 2.14 – Скриншот успешного запуска первого, второго и третьего заданий (Часть 1)

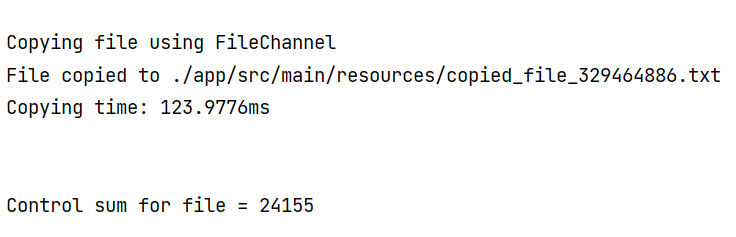


Рисунок 2.15 – Скриншот успешного запуска первого, второго и третьего заданий (Часть 2)

Наконец, после запуска четвертой задачи был создан текстовый документ в соответствующей папке, переименован, изменен и затем удален – все это отражено в консоли, что можно видеть на рисунке 2.16.

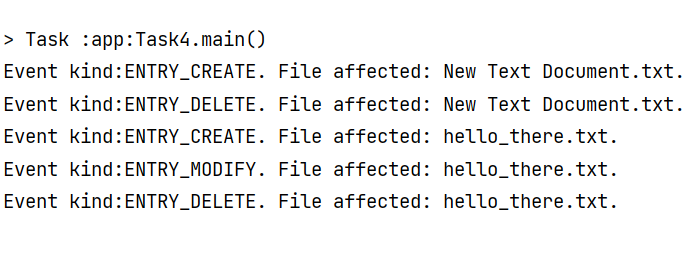


Рисунок 2.16 – Скриншот успешного запуска и работы четвертого задания

**Практическая работа №3. RxJava**

**Ход работы:** В первом задании была использована RxJava, что можно видеть на листинге 3.1.

Листинг 3.1 – Код первой задачи

package org.example;  
  
import io.reactivex.rxjava3.core.Observable;  
import io.reactivex.rxjava3.schedulers.Schedulers;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.TimeUnit;  
  
public class taskOne {  
 public static int randomNumber(int bot, int top) {  
  
 return (int)Math.*round*(bot + (Math.*random*() \* (top-bot)));  
 }  
 public static String checkResults(int[] array) {  
 if (array[0]>25 && array[1]>70)  
 return "Houston, we have a problem!";  
 if (array[0]>25)  
 return "Bad temperature stats: "+array[0];  
 if (array[1]>70)  
 return "Bad C02 stats: "+array[1];  
 return "We good B^)";  
 }  
 public static void main(String[] args) throws InterruptedException {  
 ExecutorService executor1 = Executors.*newSingleThreadExecutor*();  
 ExecutorService executor2 = Executors.*newSingleThreadExecutor*();  
 Observable<Integer> temp = Observable.*interval*(1000, TimeUnit.*MILLISECONDS*)  
 .subscribeOn(Schedulers.*from*(executor1)).map(a -> *randomNumber*(15,30));  
  
 Observable<Integer> co2 = Observable.*interval*(1000, TimeUnit.*MILLISECONDS*)  
 .subscribeOn(Schedulers.*from*(executor2)).map( a -> *randomNumber*(30, 100));  
  
 Observable.*zip*(temp, co2,  
 (observable1, observable2) -> new int[]{observable1, observable2})  
 .subscribe(item -> System.*out*.println(*checkResults*(item)));  
 }  
}

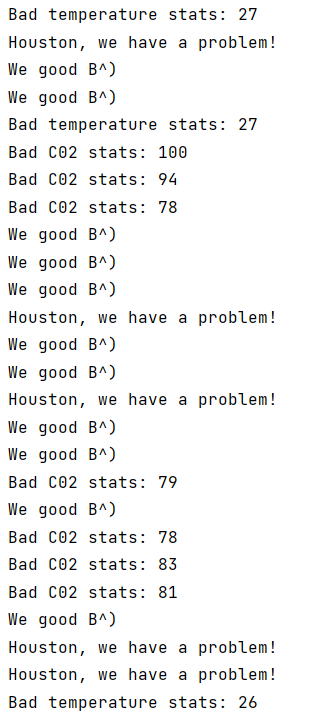


Рисунок 3.1 – Результат работы предыдущего листинга

Для второй практической работы в первом задании была выбрана задача - Преобразовать поток из 1000 случайных чисел от 0 до 1000 в поток, содержащий только числа больше 500, листинг программы представлен ниже.

Листинг 3.2 – Код задания 2.1.2

package org.example;  
import io.reactivex.rxjava3.core.Flowable;  
  
import java.util.Arrays;  
  
public class taskTwoOne {  
 public static Integer[] fillArray() {  
 Integer[] array = new Integer[1000];  
 for (int i = 0; i < array.length; i++) {  
 array[i] = (int) Math.*round*((Math.*random*() \* 1000));  
 }  
 return array;  
 }  
  
 public static void main(String[] args) {  
 Integer[] array = *fillArray*();  
 System.*out*.println(Arrays.*toString*(array));  
 Flowable<Integer> flow = Flowable.*fromArray*(array);  
 flow.subscribe(num -> System.*out*.print(num+" "));  
 System.*out*.print("\n");  
 Flowable<Integer> next = flow.filter(item -> item > 500).map(item -> item\*item);  
 next.subscribe(num -> System.*out*.print(num+" "));  
 }  
}

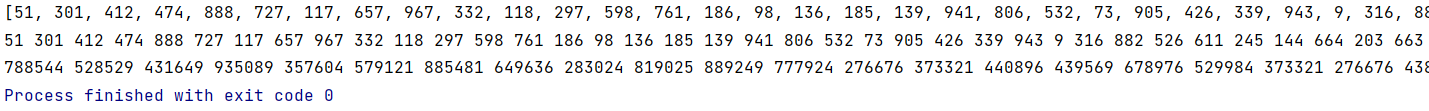


Рисунок 3.2 – Результат работы предыдущего листинга

Для второй практической работы во втором задании была выбрана задача - Даны два потока по 1000 элементов: первый содержит случайную букву, второй — случайную цифру. Сформировать поток, каждый элемент которого объединяет элементы из обоих потоков. Например, при входных потоках (A, B, C) и (1, 2, 3) выходной поток — (A1, B2, B3), листинг программы представлен ниже.

Листинг 3.3 – Код задания 2.2.1

package org.example;  
import io.reactivex.rxjava3.core.Flowable;  
  
import java.util.Random;  
  
public class taskTwoTwo {  
 public static Character[] randomFiller(String digits, Random random) {  
 Character[] array = new Character[1000];  
 for (int i = 0; i < array.length; i++) {  
 array[i] = digits.charAt(random.nextInt(digits.length()));  
 }  
 return array;  
 }  
  
  
 public static void main(String[] args) {  
 Random random = new Random();  
 final String letters = "ABCDEFGHIGKLM";  
 final String digits2 = "3567893200123";  
 Flowable<Character> firstFlow = Flowable.*fromArray*(*randomFiller*(letters, random));  
 Flowable<Character> secondFlow = Flowable.*fromArray*(*randomFiller*(digits2, random));  
 Flowable<String> finalFlow = Flowable.*zip*(firstFlow, secondFlow,  
 (char1, char2) -> String.*valueOf*(char1)+char2);  
 finalFlow.subscribe(item -> System.*out*.print(item+" "));  
 }  
}

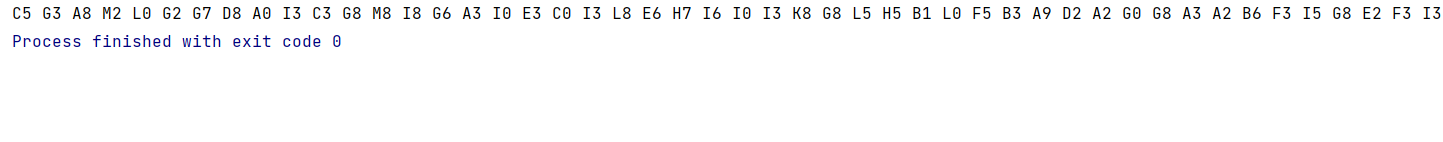


Рисунок 3.3 – Результат работы предыдущего листинга

Для второй практической работы во третьем задании была выбрана задача - Дан поток из 10 случайных чисел. Сформировать поток, содержащий только первые 5 чисел, листинг программы представлен ниже.

Листинг 3.4 – Код задания 2.3.2

package org.example;  
  
  
import io.reactivex.rxjava3.core.Flowable;  
import rx.Observable;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class taskTwoThree {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
 Observable.*from*(numbers)  
 .takeWhile(number -> number < 6)  
 .subscribe(System.*out*::println);  
 }  
}

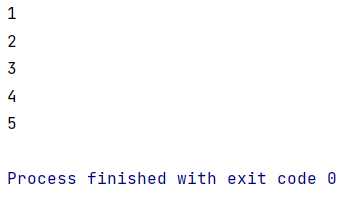


Рисунок 3.4 – Результат работы предыдущего листинга

В третьем задании был реализован класс UserFriend и поля — int userId, friendId, листинг предоставлен ниже.

Листинг 3.5 – Код задания 3

package org.example;  
  
  
import io.reactivex.rxjava3.core.Observable;  
  
public class taskThree {  
 public static int randomNumber(int bot, int top) {  
 return (int)Math.*round*(bot + (Math.*random*() \* (top-bot)));  
 }  
 public static UserFriend[] fillArray() {  
 UserFriend[] array = new UserFriend[1000];  
 for (int i = 0; i < array.length; i++) {  
 int userId = (int) Math.*round*((Math.*random*() \* 50));  
 int friendId = (int) Math.*round*((Math.*random*() \* 50));  
 array[i] = new UserFriend(userId,friendId);  
 }  
 return array;  
 }  
 public static class UserFriend {  
 private final int userId;  
 private final int friendId;  
 public UserFriend(int userId, int friendId) { this.userId = userId; this.friendId = friendId; }  
 public int getUserId() { return userId; }  
 @Override  
 public String toString() { return "UserFriend{" + "userId=" + userId + ", friendId=" + friendId + "}\n"; }  
 }  
 public static Observable<UserFriend> getFriends(int userId, Observable<UserFriend> users) {  
 return users.filter(x -> x.getUserId() == userId);  
 }  
  
 public static void main(String[] args) {  
 UserFriend[] userFriends = *fillArray*();  
 Integer[] randIds = new Integer[15];  
 for (int i=0; i<randIds.length; i++)  
 randIds[i] = *randomNumber*(0,50);  
 Observable<UserFriend> userFriendObservable = Observable.*fromArray*(userFriends);  
 Observable<UserFriend> filtered = Observable.*empty*();  
 for (Integer randId : randIds) {  
 filtered = filtered.concatWith(*getFriends*(randId, userFriendObservable));  
 }  
 filtered.subscribe(item -> System.*out*.print(item.toString()+" "));  
 }  
}



Рисунок 3.5 – Результат работы предыдущего листинга

В четвертом задании был реализована система:

Файл. Имеет следующие характеристики:

0. Тип файла (например XML, JSON, XLS)

1. Размер файла — целочисленное значение от 10 до 100.

Генератор файлов -- генерирует файлы с задержкой от 100 до 1000 мс.

Код задания представлен на листинге 3.6.

Листинг 3.6 – Код задания 4

package org.example;  
  
import rx.Observable;  
import rx.Observer;  
  
public class Task4 {  
 private int count = 0;  
  
 public class Handler implements Runnable{  
 Task4File file;  
 Handler(Task4File file){  
 this.file = file;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(file.getSize()\* 70L);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 count--;  
 System.*out*.println(file.getType() + " file with size " + file.getSize() + " bytes processed");  
 }  
 }  
  
 Observer<Task4File> getQueue() {  
 return new Observer<>() {  
 @Override  
 public void onCompleted() {  
 }  
  
 @Override  
 public void onError(Throwable throwable) {  
 }  
  
 @Override  
 public void onNext(Task4File file) {  
 Handler handler = new Handler(file);  
 new Thread(handler).start();  
 }  
 };  
 }

Продолжение листинга 3.6 представлено ниже:

public Observable<Task4File> getGenerator(){  
 String[] types = new String[]{"xml", "json", "xls"};  
 return Observable.*create*(subscriber -> {  
 Runnable r = () -> {  
 while (true){  
 String type = types[(int)(Math.*random*()\*3)];  
 int size = (int)(Math.*random*()\*90)+10;  
 try {  
 Thread.*sleep*((int)(Math.*random*()\*900)+100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 Task4File file = new Task4File(type, size);  
 System.*out*.println(type + " file with size " + size + " bytes generated");  
 while (count == 4){  
 try {  
 Thread.*sleep*(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(file.getType() + " file with size " +  
 file.getSize() + " bytes added to queue");  
 count++;  
 subscriber.onNext(file);  
 }  
 };  
 new Thread(r).start();  
 }  
 );  
 }  
  
 public static void main(String[] args){  
 Task4 task4 = new Task4();  
 Observer<Task4File> queue = task4.getQueue();  
 Observable<Task4File> generator = task4.getGenerator();  
 generator.subscribe(queue);  
 }  
}

Листинг 3.7 – Код файла Task4File

package org.example;  
  
public record Task4File(String type, int size) {  
 public String getType() {  
 return type;  
 }  
  
 public int getSize() {  
 return size;  
 }  
  
}



Рисунок 3.6 – Результат работы предыдущего листинга

**Практическая работа №4**

**Ход работы:**

Была реализована клиент-серверная система, основные элементы которой представлены на следующих скриншотах.

В первую очередь была разработана база данных на базе PosgreSQL. На рисунках 4.1-4.2 представлены скриншоты содержимого баз данных.

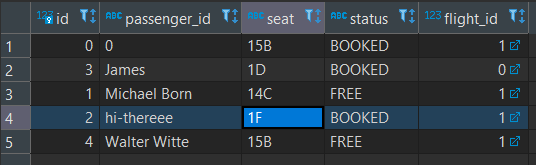


Рисунок 4.1 – Скриншот содержимого базы данных с билетами

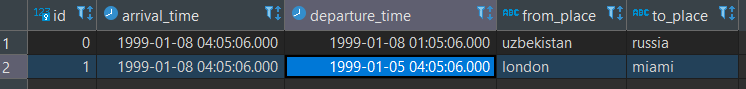


Рисунок 4.2 – Скриншот содержимого базы данных с полетами

Развертывание базы данных происходило в контейнере, docker-compose файл которого представлен на рисунке 4.3.

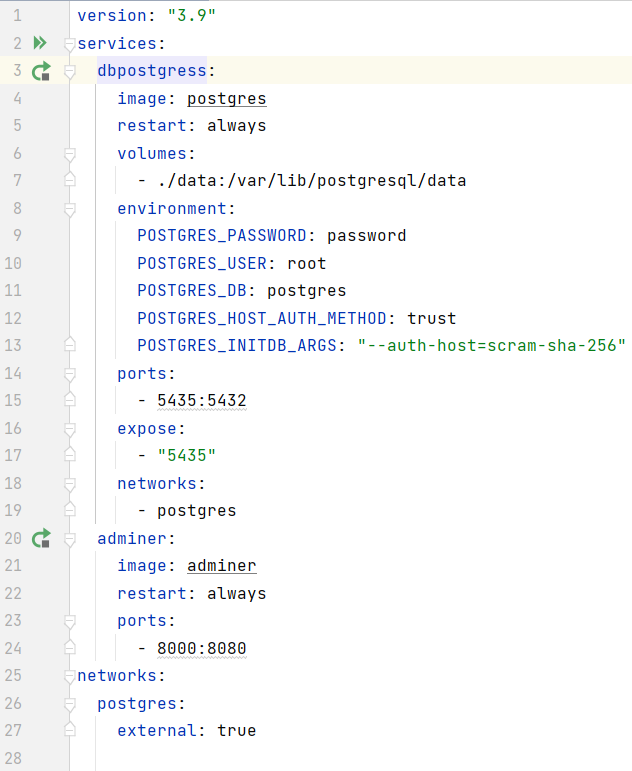


Рисунок 4.3 – Скриншот содержимого файла docker-compose.yaml

Для успешной сборки проекта использовался сборщик maven, на рисунках 4.4-4.5 представлены зависимости в проекте.

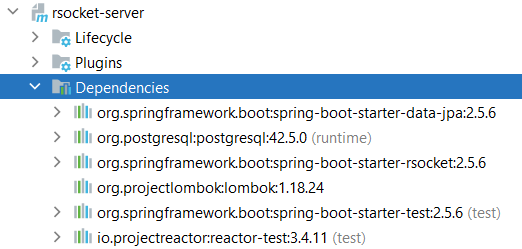


Рисунок 4.4 – Скриншот зависимостей серверной части

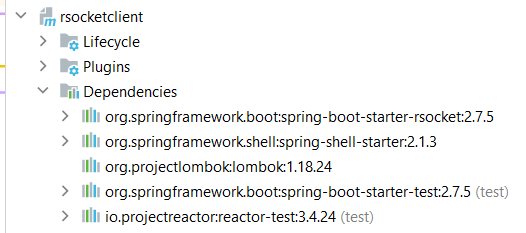


Рисунок 4.5 – Скриншот зависимостей клиентской части

В результате сборки и запуска приложения, были реализованы следующие методы, требуемые в постановке задачи:

Request-Response реализован с помощью команды *book --ticket-id --passenger-id*, благодаря ей происходит бронь места. Пример работы представлен на рисунках 4.6-4.7.

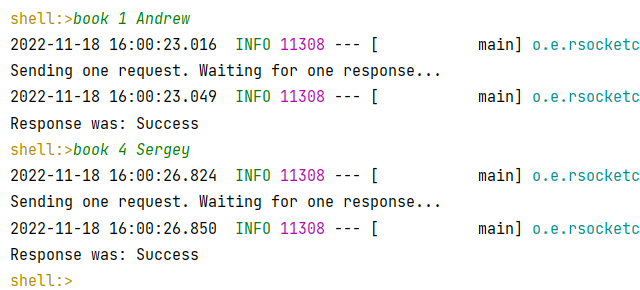


Рисунок 4.6 – Скриншот реализации метода Request-Response

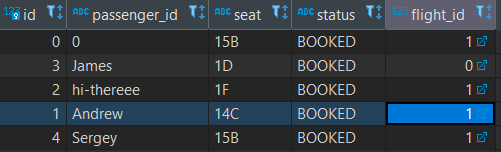


Рисунок 4.7 – Скриншот результата использования метода

Request-Stream метод реализован посредством команды *flights*, благодаря которой происходит вывод всех рейсов. На рисунке 4.8 представлены скриншоты действия метода.

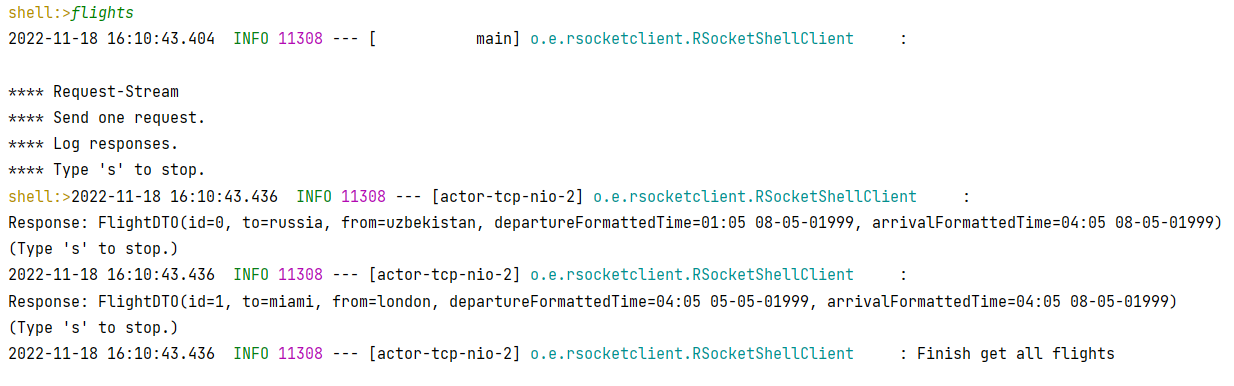


Рисунок 4.8 – Скриншот реализации метода Request-Stream

Channel метод реализован с помощью команды *tickets --flight-ids*, который производит вывод броней рейсов по идентификатору полета. На рисунке 4.9 изображены скриншоты работы метода.

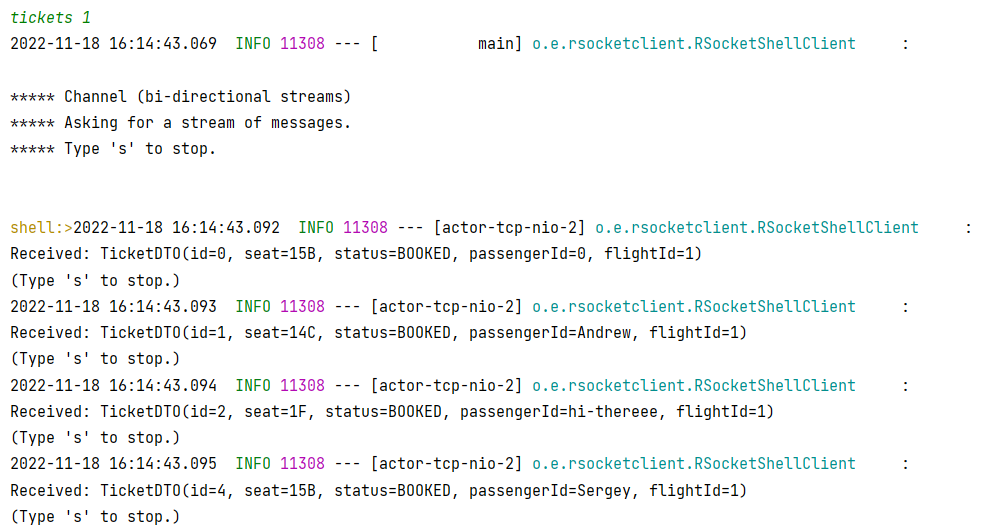


Рисунок 4.9 – Скриншот реализации и использования метода Channel

Fire-and-Forget метод реализован посредством команды *delete-reservation --ticket-id*. С ее помощью происходит удаление брони на место. На следующих двух рисунках (4.11-4.13) изображены скриншоты работы метода.

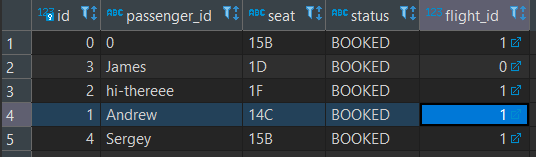


Рисунок 4.11 – Скриншот содержимого базы данных до выполнения метода



Рисунок 4.12 – Скриншот реализации метода Fire-and-Forget

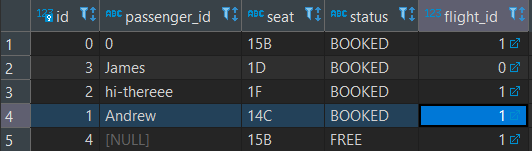


Рисунок 4.13 – Скриншот содержимого базы данных после выполнения метода

**Вывод:** в ходе выполнения практических работ по дисциплине «Разработка клиент-серверных приложений» были получены полезные теоретические, а главное практические навыки создания приложений, использования многопоточности разными методами (наследованием классов Thread и ForkJoin), создания и обработки файлов, создания и обработки файлов, использования пакета java.nio, а также реализаций копирования методами FileInputStream/FileOutputStream, FileChannel, Apache Commons IO, Files class. В третьей практической работе были использованы классы Observer и Observable из библиотеки RxJava. В четвертой практической работе была разработана клиент-серверная система с использованием протокола RSocket, продемонстрированы 4 типа взаимодействия в рамках протокола (Request-Response, Request-Stream, Channel, Fire-and-Forget).