|  |  |  |
| --- | --- | --- |
|  |  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |

Институт Информационных технологий

Кафедра Инструментального и прикладного программного обеспечения

**ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ 1-8**

по дисциплине «Разработка клиент-серверных приложений»

**Студент группы** ИКБО-20-19 Анваржонов Ж. Т

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись студента)

**Руководитель практической работы** преподаватель Зарипов Е. А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Работа представлена «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ 2022 г.

Допущен к работе «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2022 г.

Москва 2022

Оглавление

# **Практическая работа №1**

**Задание 1**

Дан массив из 10000 элементов. Необходимо написать несколько реализаций некоторой функции F в зависимости от варианта. Функция должна быть реализована следующими способами:

1. Последовательно
2. С использованием многопоточности (Thread, Future, и т. д.)
3. С использованием ForkJoin.

После каждой операции с элементом массива (сравнение, сложение) добавить задержку в 1 мс при помощи Thread.sleep(1);

Провести сравнительный анализ затрат по времени и памяти при запуске каждого из вариантов реализации.

Варианты функций (выбор варианта осуществляется по формуле «Номер в

списке группы % 3»)

1. Поиск суммы элементов массива.
2. Поиск максимального элемента в массиве.
3. Поиск минимального элемента в массиве.

**Выполнение работы**

Номер в группе – 2.

Покажем реализацию функции поиска суммы элементов в массиве.

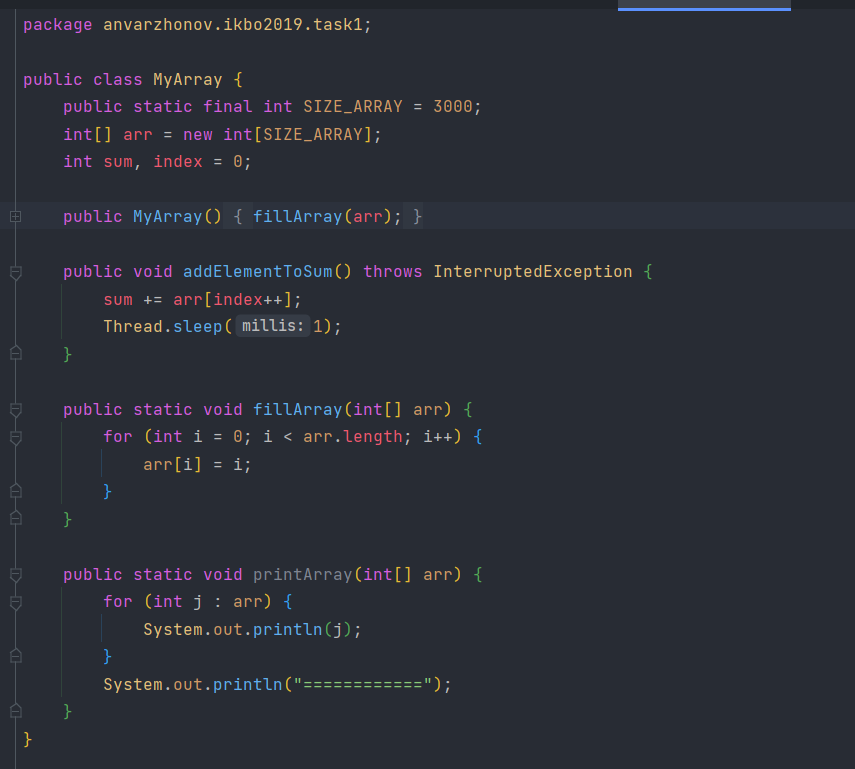


Рисунок 1.1 – Фрагмент кода с созданием массива и методом добавления элемента к сумме «sum»

Реализуем функцию поиска суммы элементов массива последовательным способом. На рисунке 1.2 представлена реализация.

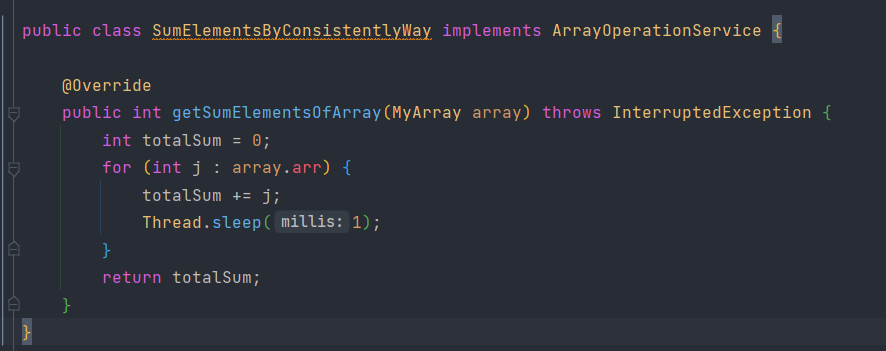


Рисунок 1.2 – Фрагмент кода поиска сумму элементов последовательным способом

Реализуем функцию поиска суммы элементов массива с использованием многопоточности. На рисунке 1.3 представлена реализация.



Рисунок 1.3 – Фрагмент кода поиска сумму элементов c использованием многопоточности

Реализуем функцию поиска суммы элементов массива с использованием ForkJoin. Реализация будет идентичная с предыдущим способом за исключением метода фабричного метода, который создает Work Stealing Pool. На рисунке 1.4 представлена реализация.



Рисунок 1.4 – Фрагмент кода поиска сумму элементов с использованием ForkJoin

На рисунке 1.5 покажем результаты выполнения алгоритма поиска суммы элементов массива разными способами.

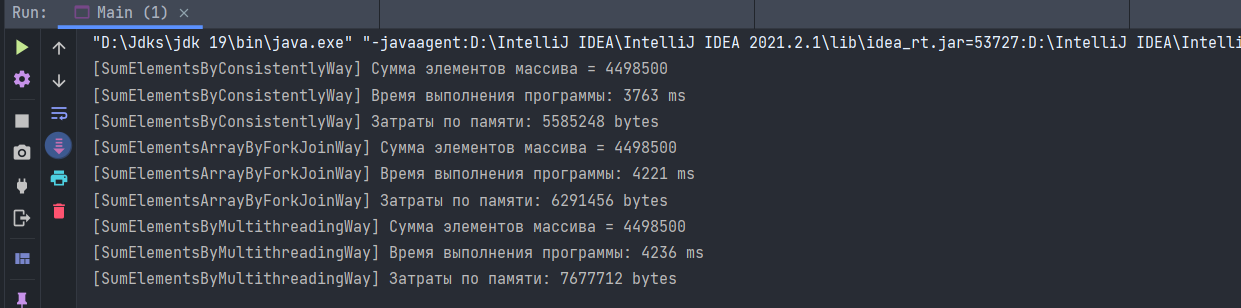


Рисунок 1.5 – Сравнительный анализ затрат по времени и памяти при запуске каждого из вариантов способов

**Задание 2**

Программа запрашивает у пользователя на вход число. Программа имитирует обработку запроса пользователя в виде задержки от 1 до 5 секунд выводит результат: число, возведенное в квадрат. В момент выполнения запроса пользователь имеет возможность отправить новый запрос. Реализовать с использованием Future.

**Выполнение работы**

На рисунке 1.6 покажем класс, функционалом которого будет имитация запроса пользователя.



Рисунок 1.6 – Сервис обработки запроса пользователя

На рисунке 1.7 представлен главный класс, который будет взаимодействовать с сервисом по обработки запроса пользователя

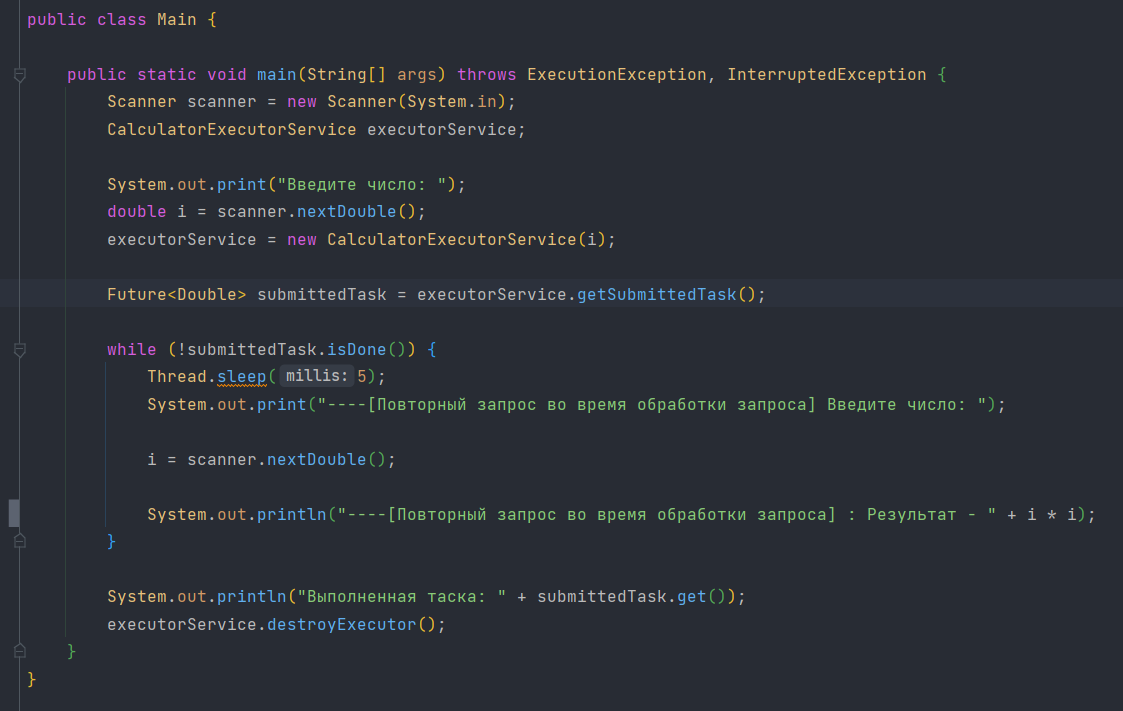


Рисунок 1.7 – Главный класс программы

На рисунке 1.8 покажем результат работы программы.

**Задание 3**

Реализовать следующую многопоточную систему.

Файл. Имеет следующие характеристики:

1. Тип файла (например XML, JSON, XLS)
2. Размер файла — целочисленное значение от 10 до 100.

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

Очередь — получает файлы из генератора.

Вместимость очереди — 5 файлов.

Обработчик файлов — получает файл из очереди. Каждый обработчик имеет параметр — тип файла, который он может обработать. Время обработки файла: «Размер файла\*7мс»

Система должна удовлетворять следующими условиям:

0. Должна быть обеспечена потокобезопасность.

1. Работа генератора не должна зависеть от работы обработчиков, и наоборот.

2. Если нет задач, то потоки не должны быть активны.

3. Если нет задач, то потоки не должны блокировать другие потоки.

4. Должна быть сохранена целостность данных

# Практическая № 3

1. Алла

Заведем два Observable для датчиков температуры и co2. Данные для датчиков будем генерировать рандомно с помощью RandomUtils.

Следующим шагом Восспользуемся Observable.zip чтобы из первых двух потоков создать новый объект Containter, который будет хранить в себе значения этих датчиков.

Когда подпишемся на текущего Observable, вызываем функцию get

