

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ**

Ордена Трудового Красного Знамени

**Федеральное государственное бюджетное образовательное
учреждение высшего образования**

«Московский технический университет связи и информатики»

Лабораторная работа № 8

«Аннотации»

Выполнила: Студентка группы БВТ2306

Максимова Мария

Москва 2024

Задание:

Вам необходимо разработать приложение, которое считывает данные из исходного источника (например, файл, база данных или сетевой ресурс), применяет к ним различные операции с использованием Stream API, и сохраняет результаты в новый источник данных.

Аннотации — это метаданные, которые могут быть присоединены к классам, методам, полям и другим элементам программы в Java. Аннотации позволяют добавлять дополнительную информацию к коду и могут использоваться для анализа и автоматизации процесса компиляции

```
/*@Target
Retention: RUNTIME;
Target: ANNOTATION_TYPE;
Данная аннотация задает тип объекта над которым может указываться создаваемая
нами аннотация.

@Retention
Retention: RUNTIME;
Target: ANNOTATION_TYPE;
Данная аннотация задает "тип хранения" аннотации над которой она указана
```

1. Аннотация @DataProcessor

Этот файл создает аннотацию @DataProcessor, которую можно использовать для методов.

- **Назначение:** помечать методы, которые должны участвовать в обработке данных.
- **Особенности:**
 - Аннотация работает во время выполнения программы (@Retention(RetentionPolicy.RUNTIME)).
 - Применяется только к методам (@Target(ElementType.METHOD)).

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface DataProcessor {
}
```

Рисунок 1

2. Класс DataManager

DataManager реализует управление процессами обработки данных, от загрузки до сохранения.

Основные поля:

- `processors`: список объектов, содержащих методы для обработки данных.
- `data`: список исходных строк данных.
- `processedData`: список обработанных строк.
- `executor`: пул потоков для асинхронной обработки данных.

Ключевые методы:

1. `registerDataProcessor`:

Добавляет объект с методами, помеченными аннотацией `@DataProcessor`, в список процессоров.

2. `loadData`:

Читает данные из указанного текстового файла и сохраняет их в `data`.

3. `processData`:

Организует параллельную обработку данных:

- Находит методы с аннотацией `@DataProcessor` в зарегистрированных процессорах.
- Выполняет их асинхронно, собирая результаты в `processedData`.
- Использует `synchronized`, чтобы обеспечить безопасное добавление данных в многопоточной среде.
- После завершения обработки заменяет содержимое `data` на результат.

4. `saveData`:

Записывает обработанные данные в указанный текстовый файл.

5. `shutdown`:

Завершает работу пула потоков.

```

public class DataManager { 2 usages
    private final List<Object> processors = new ArrayList<>(); 2 usages
    private List<String> data = new ArrayList<>(); 5 usages
    private final List<String> processedData = new ArrayList<>(); // обработанные данные 3 usages
    private final ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4); 2 usages

    public void registerDataProcessor(Object processor) { 1 usage
        processors.add(processor);
    }

    public void loadData(String source) { 1 usage
        try {
            data.addAll(Files.readAllLines(Paths.get(source)));
        } catch (IOException e) {
            throw new RuntimeException("Failed to load data from file: " + source , e);
        }
    }

    public void processData() { 1 usage
        System.out.println("Обработка данных...");
        List<CompletableFuture<Void>> futures = new ArrayList<>();

        for (Object processor : processors) {
            for (Method method : processor.getClass().getDeclaredMethods()) {
                if (method.isAnnotationPresent(DataProcessor.class)) {
                    futures.add(CompletableFuture.runAsync(() -> {

```

Рисунок 2

```

                        try {
                            List<String> result = (List<String>) method.invoke(processor, data);
                            synchronized (processedData) {
                                processedData.addAll(result);
                            }
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }, executor));
                }
            }
        }

        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        data.clear();
        data.addAll(processedData);
    }

    public void saveData(String destination) { 1 usage
        try {
            Files.write(Paths.get(destination), data);
        } catch (IOException e) {
            throw new RuntimeException("Failed to save data to file: " + destination, e);
        }
    }
}

```

Рисунок 3

3. Класс FilterProcessor

FilterProcessor — это пример процессора, содержащего три метода обработки данных, помеченных аннотацией @DataProcessor:

1. **filterData:**

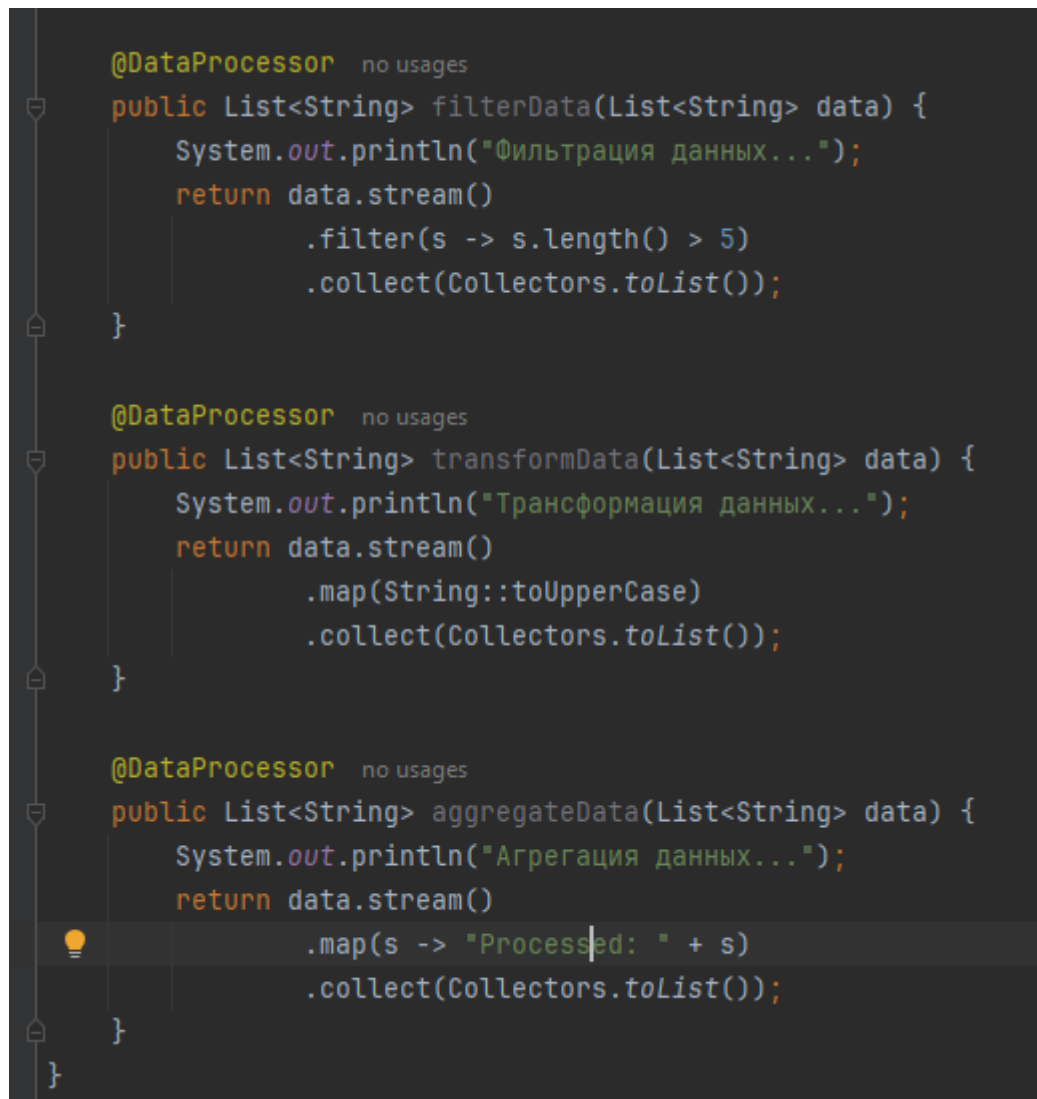
Оставляет строки длиной больше 5 символов.

2. **transformData:**

Преобразует строки в верхний регистр.

3. **aggregateData:**

Добавляет к каждой строке префикс "Processed: ".

A screenshot of a code editor showing the implementation of the FilterProcessor class. The code is written in Java and includes three methods: filterData, transformData, and aggregateData, each annotated with @DataProcessor. The filterData method filters strings by length, transformData converts strings to uppercase, and aggregateData adds a prefix to each string. The code is color-coded, and there are some UI elements like a lightbulb icon and a cursor visible in the editor.

```
@DataProcessor no usages
public List<String> filterData(List<String> data) {
    System.out.println("Фильтрация данных...");
    return data.stream()
        .filter(s -> s.length() > 5)
        .collect(Collectors.toList());
}

@DataProcessor no usages
public List<String> transformData(List<String> data) {
    System.out.println("Трансформация данных...");
    return data.stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());
}

@DataProcessor no usages
public List<String> aggregateData(List<String> data) {
    System.out.println("Агрегация данных...");
    return data.stream()
        .map(s -> "Processed: " + s)
        .collect(Collectors.toList());
}
```

Рисунок 4

4. Класс Main

Класс Main демонстрирует использование системы обработки данных:

1. Создает экземпляр DataManager.
2. Регистрирует процессор FilterProcessor.
3. Загружает данные из файла input.txt.
4. Выполняет обработку данных с использованием методов из FilterProcessor.
5. Сохраняет обработанные данные в файл output.txt.
6. Завершает работу пула потоков.

```
public static void main(String[] args) {  
    DataManager manager = new DataManager();  
    FilterProcessor filters = new FilterProcessor();  
  
    manager.registerDataProcessor(filters);  
  
    manager.loadData( source: "src/input.txt");  
    manager.processData();  
    manager.saveData( destination: "src/output.txt");  
  
    manager.shutdown();  
}
```

Рисунок 5