Министерство науки высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

(Университет ИТМО)

Факультет программной инженерии и компьютерной техники

ЛАБОРАТОРНАЯ РАБОТА №1

По курсу «Реактивная Java»

Предметная область «Матчи, турниры, команды»

Выполнили:

Голоскок Дмитрий Сергеевич P4119

Симовин Кирилл Константинович P4116

Преподаватель:

Гаврилов Антон Валерьевич

Санкт-Петербург

2024

Задание

1. Написать для согласованной предметной области как минимум 3 базовых класса и генераторы объектов. Генератор должен уметь создавать указанное количество различных объектов сответствующего класса со случайными (но при этом валидными) характеристиками. Класс, представляющий собой массовый объект должен обязательно содержать поля следующих типов:

– один из примитивов (int, long, double);

– String;

– дата/время (LocalDate, LocalTime, ...);

– Enum;

– Record;

– массив или коллекция.

Остальные поля – произвольные, какие нужны для предметной области.

Два оставшихся класса должны представлять собой дополнительные атрибуты и характеристики массового класса.

2. С помощью генератора создать коллекцию объектов.

3. Написать код, реализующий расчет согласованных агрегированных статических данных тремя способами:

– итерационным циклом по коллекции;

– конвейером с помощью Stream API на базе коллекторов из стандартной библиотеки;

– конвейером с помощью собственного коллектора.

4. Для каждого варианта измерить время выполнения, зафиксировав моменты начала и окончания расчета для количества элементов в коллекции – 5000, 50000 и 250000. Время измерять с помощью методов класса System или Instant.

Ход выполнения работы

В ходе исследования времени работы программ здесь и далее проводилось усреднение по 12 запускам.

Программа, производящая расчет значений в цикле, представляла собой набор методов, каждый из которых отвечал за расчет определенного статистического параметра итерационным путем, используя цикл for.

Программа, производящая расчет значений с использованием стандартного коллектора, также представляла собой набор методов, каждый из которых отвечал за расчет определенного статистического параметра. Однако в данном случае использовался стандартный коллектор. Для расчета необходимых параметров использовался метод .filter(Predicate predicate), а отфильтрованный результат в дальнейшем приводился к списку, путем использования метода .toList().

Программа, производящая расчет значений с использованием собственного коллектора аналогична программе с использованием стандартного коллектора, однако отфильтрованный результат приводился к ArrayList.

Для возможности работы собственного коллектора со всеми реализованными классами был реализован интерфейс IReactive, унифицирующий все разработанные классы.

В таблице 1 представлено время выполнения программ в зависимости от размера обрабатываемых коллекций.

Таблица 1 – Сравнительная таблица методов обработки

Число элементов	Время выполнения программ, с		
	Цикл	Стандартный коллектор	Собственный коллектор
5000	0,01167	0,00525	0,00392
50000	0,03458	0,03575	0,03725
250000	0,06642	0,14467	0,14467

Как можно заметить, программа с использованием цикла работает быстрее в двух из трех случаев – не считая теста на 5000 тысячах элементах.

Вероятно, это обусловлено тем, что на малых значениях коллекторам приходилось собирать в коллекцию меньше элементов, удовлетворяющих условию, чем в случае с большими размерами коллекций.

В случаях с 50000 и 250000 элементов коллекторы проигрывают итеративному подходу ввиду того, что цикл for при работе с массивами или списками более легковесны и требуют меньше ресурсов ЦПУ и кучи.

Однако, стоит отметить, что среди программ, реализованных с использованием коллектора лучший результат был продемонстрирован в рамках программы с использованием собственного коллектора.

На рисунке 1 приведен график зависимости времени работы программ от размера коллекций.

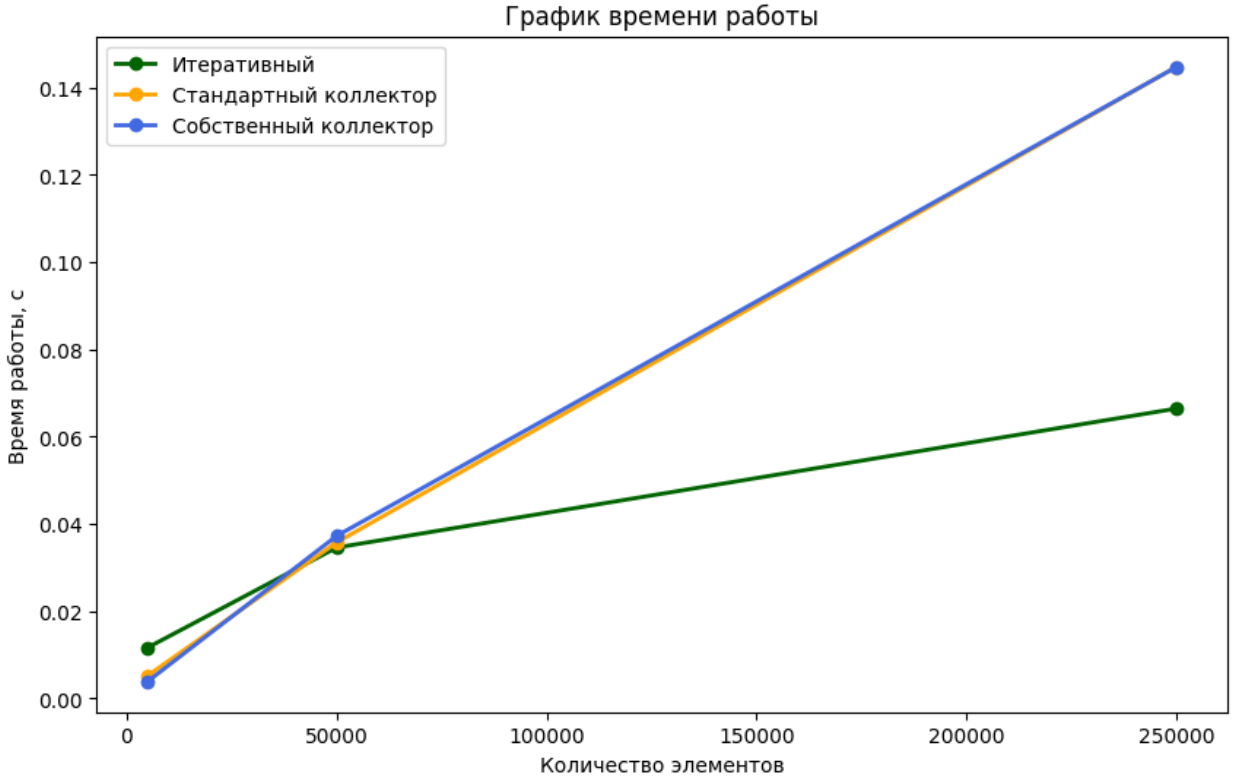

Рисунок 1 – График времени работы программ

Заключение

Цель лабораторной работы – сравнить скорость работы трех решений для расчета статистических данных. Показано, что на предоставленном размере коллекций итеративный способ является более предпочтительным.

Однако, для 5000 элементов итеративный способ является более медленным, по сравнению с коллекторами: 0,01167 секунд против 0,00525 секунд для стандартного коллектора и 0,00392 секунд с использованием собственного коллектора.

Для 50000 элементов решения с коллекторами уступили итеративному подходу: 0,03458 секунд для итеративного против 0,03575 секунд для решения с использованием стандартного коллектора и 0,03725 секунд с использованием собственного.

Для 250000 элементов итеративный подход оказался более, чем в два раза быстрее: 0,06642 секунды против 0,14467 секунд для обоих решений с коллектором.

В ходе выполнения лабораторной работы были получены навыки по работе с коллекторами и реализации собственных.

ПРИЛОЖЕНИЕ А

Код Record-класса

```
import org.example.interfaces.IReactive;
import org.jetbrains.annotations.Contract;
import org.jetbrains.annotations.NotNull;

import java.time.LocalDate;


public record Tournament(String name, String place, LocalDate startDate,
LocalDate endDate) implements IReactive {
    @Contract(pure = true)
    @Override
    public @NotNull String toString() {
        return "Tournament{name=%s, place=%s, startDate=%s, endDate=%s}"
                .formatted(this.name, this.place, this.startDate,
this.endDate);
    }
}
```

ПРИЛОЖЕНИЕ Б

Код класса с коллекцией

```
import lombok.AllArgsConstructor;
import lombok.Data;
import org.example.interfaces.IReactive;

import java.util.List;


@Data
@AllArgsConstructor
public class Team implements IReactive {
    private String name;

    private List<String> members;

    private int wins;

    private int loses;
}
```

ПРИЛОЖЕНИЕ В

Код главного класса

```
import lombok.AllArgsConstructor;
import lombok.Data;
import org.example.enums.MatchType;
import org.example.interfaces.IReactive;

import java.time.LocalDateTime;


@Data
@AllArgsConstructor
public class Match implements IReactive {

    private Team team1;

    private Team team2;

    private LocalDateTime startDateTime;

    private LocalDateTime endDateTime;

    private Tournament tournament;

    private int scoreTeam1;

    private int scoreTeam2;

    private MatchType matchType;

    private String map;
}
```

ПРИЛОЖЕНИЕ Г

Код перечисления

```
import lombok.AllArgsConstructor;
import lombok.Getter;


@Getter
@AllArgsConstructor
public enum MatchType {


    DEATHMATCH("Deathmatch"),


    FLAG_CAPTURE("Flag Capture"),


    TEAM_BATTLE("Team Battle");

    private final String type;
}
```

ПРИЛОЖЕНИЕ Д

Код коллектора

```
import org.example.interfaces.IReactive;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;


public class ReactiveCollector implements Collector<IReactive,
List<IReactive>, List<IReactive>> {
    @Override
    public Supplier<List<IReactive>> supplier() {
        return ArrayList::new;
    }


    @Override
    public BiConsumer<List<IReactive>, IReactive> accumulator() {
        return List::add;
    }


    @Override
    public BinaryOperator<List<IReactive>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }


    @Override
    public Function<List<IReactive>, List<IReactive>> finisher() {
        return Function.identity();


    @Override
    public Set<Characteristics> characteristics() {
        return Set.of();
    }
}
```