МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

ОТЧЕТ ПО
ЛАБОРАТОРНОЙ РАБОТЕ № 2

«Разработка набора классов для работы с табулированными функциями»

по курсу
Объектно-ориентированное программирование

Выполнил: Волынец Никита,
группа 6203-010302D

Оглавление

Задание №1

Первым шагом я создал пакет functions, который будет содержать все классы, связанные с реализацией табулированных функций.

Задание №2

В созданном пакете functions я разработал класс FunctionPoint, который представляет точку табулированной функции. Класс инкапсулирует координаты точки по осям X и Y. Для обеспечения правильной работы с данными были реализованы три конструктора: конструктор с параметрами для создания точки с заданными координатами, конструктор копирования для создания новой точки на основе существующей, и конструктор по умолчанию, создающий точку в начале координат (0, 0). Также были добавлены геттеры и сеттеры для доступа к координатам точки, что соответствует принципам инкапсуляции.

```java
package  functions;

public class FunctionPoint{  18 usages
    private double x;  6 usages
    private double y;  6 usages

    // конструктор с параметрами
    public FunctionPoint(double x, double y){  26 usages
        this.x = x;
        this.y = y;
    }
    // конструктор копирования
    public FunctionPoint(FunctionPoint point){  22 usages
        this.x = point.x;
        this.y = point.y;
    }
    // конструктор по умолчанию
    public FunctionPoint(){  20 usages
        this.x = 0;
        this.y = 0;
    }
    // получить x
    public double getX(){  19 usages
        return x;
    }
    // получить y
    public double getY(){  5 usages
        return y;
    }
    // установить x
    public void setX(double x){  1 usage
        this.x = x;
    }
    // установить y
    public void setY(double y){  1 usage
        this.y = y;
    }
}
```

Задание №3

Следующим этапом я создал класс TabulatedFunction для представления табулированной функции. В качестве внутренней структуры данных используется массив объектов FunctionPoint, причем точки всегда поддерживаются в упорядоченном состоянии по координате X. Были реализованы два конструктора: первый создает функцию с равномерно распределенными точками на заданном интервале с начальными значениями Y равными 0, второй принимает массив значений Y и создает точки с равномерным распределением по X. Оба конструктора обеспечивают правильное начальное состояние объекта.

```java
package functions;

public class TabulatedFunction {  3 usages
    private FunctionPoint[] point_mass;  37 usages
    private int point_count;  28 usages

    // конструктор создающий функцию с равномерно распределенными точками
    public TabulatedFunction(double leftX, double rightX, int pointsCount){  17 usages
        // создаем массив для хранения точек
        this.point_mass = new FunctionPoint[pointsCount];
        this.point_count = pointsCount;
        // вычисляем шаг между точками
        double step = (rightX-leftX)/(pointsCount-1);

        for(int i =0; i < pointsCount; i++){
            // вычисляем x координату для каждой точки
            double x = leftX + i * step;
            // создаем точку с вычисленным x и y=0
            point_mass[i] = new FunctionPoint(x,  y: 0);
        }
    }
    // конструктор создающий функцию с заданными значениями y
    public TabulatedFunction(double leftX, double rightX, double[] values){  16 usages
        // создаем массив точек размером с массив значений
        this.point_mass = new FunctionPoint[values.length];
        this.point_count = values.length;

        // вычисляем шаг между точками
        double step = (rightX-leftX)/(point_count-1);

        for(int i =0; i < point_count; i++){
            // вычисляем x координату
            double x = leftX + i * step;
            // создаем точку с вычисленным x и заданным y
            point_mass[i] = new FunctionPoint(x, values[i]);
        }
    }
```

Задание №4

Для класса TabulatedFunction я реализовал методы доступа к характеристикам функции: getLeftDomainBorder() и getRightDomainBorder() возвращают границы области определения функции, а getFunctionValue(double x) вычисляет значение функции в произвольной точке X с использованием линейной интерполяции. Метод корректно обрабатывает точки внутри области определения, выполняя интерполяцию между соседними точками, и возвращает Double.NaN для точек вне области определения.

```
    public double getLeftDomainBorder(){  1 usage
        return this.point_mass[0].getX();
    }
    // получить правую границу области определения функции
    public double getRightDomainBorder(){  1 usage
        return this.point_mass[point_count-1].getX();
    }
    // вычислить значение функции в заданной точке x
    public double getFunctionValue(double x){  3 usages
        // если точек нет возвращаем не число
        if(point_count == 0) return Double.NaN;

        // проверяем что x находится в области определения
        if(x >= point_mass[0].getX() && x <= point_mass[point_count-1].getX()){
            // ищем точку с точно таким же x
            for(int i = 0; i < point_count; i++){
                if(point_mass[i].getX() == x){
                    // если нашли возвращаем соответствующий y
                    return point_mass[i].getY();
                }
            }
            // если точного совпадения нет ищем интервал для интерполяции
            for(int i = 0; i < point_count-1; i++){
                double x1 = point_mass[i].getX();
                double x2 = point_mass[i+1].getX();

                // проверяем попадает ли x в текущий интервал
                if(x >= x1 && x <= x2){
                    double y1 = point_mass[i].getY();
                    double y2 = point_mass[i+1].getY();

                    // вычисляем значение линейной интерполяции
                    return y1+(y2-y1)*(x-x1)/(x2-x1);
                }
            }
        }
        // если x вне области определения возвращаем не число
        return Double.NaN;
    }
```

Задание №5

Были добавлены методы для работы с отдельными точками функции. Метод getPointsCount() возвращает количество точек, getPoint(int index) возвращает копию точки по указанному индексу. Методы setPoint(int index, FunctionPoint point), setPointX(int index, double x) и setPointY(int index, double y) позволяют изменять координаты точек с проверкой корректности , гарантируя, что изменения не нарушают упорядоченность точек по X. Все методы обеспечивают защиту внутренних данных через возврат копий объектов.

```
// получить общее количество точек в функции
    public int getPointsCount(){
        return point_count;
    }
    // получить копию точки по указанному индексу
    public FunctionPoint getPoint(int index){
        // возвращаем копию чтобы защитить исходные данные
        return new FunctionPoint(point_mass[index].getX(),
point_mass[index].getY());
    }
```

```
    // заменить точку по указанному индексу
    public void setPoint(int index, FunctionPoint point) {
        // проверяем валидность индекса
        if (index < 0 || index >= point_count) {
            return;
        }
        // проверяем что новый x не нарушает порядок точек
        // слева от текущей точки x должен быть меньше
        // справа от текущей точки x должен быть больше
        if ((index > 0 && point.getX() <= point_mass[index -
1].getX()) || (index < point_count - 1 && point.getX() >=
point_mass[index + 1].getX())){
            return;
        }
        // создаем новую точку чтобы избежать ссылочной зависимости
        point_mass[index] = new FunctionPoint(point);
    }
    // получить координату x точки по индексу
    public double getPointX(int index){
        return point_mass[index].getX();
    }
    // установить новую координату x для точки по индексу
    public void setPointX(int index, double x) {
        // проверяем валидность индекса
        if (index < 0 || index >= point_count) {
            return;
        }
        // проверяем что новый x сохраняет порядок точек
        if ((index > 0 && x <= point_mass[index - 1].getX()) || (index
< point_count - 1 && x >= point_mass[index + 1].getX())){
            return;
        }
        point_mass[index].setX(x);
    }
    // получить координату y точки по индексу
    public double getPointY(int index){
        return point_mass[index].getY();
    }
    // установить новую координату y для точки по индексу
    public void setPointY(int index, double y) {
        // проверяем валидность индекса
        if (index < 0 || index >= point_count) {
            return;
        }
        point_mass[index].setY(y);
    }
```

Задание №6

Для динамического изменения количества точек я реализовал методы deletePoint(int index) и addPoint(FunctionPoint point). Метод deletePoint удаляет точку по указанному индексу, сохраняя минимальное количество точек (не менее 2). Метод addPoint добавляет новую точку в правильную позицию для поддержания упорядоченности по X. Оба метода эффективно работают с массивом точек, используя System.arraycopy() для перемещения элементов, и динамически расширяют массив при необходимости, удваивая его размер когда достигается предельная емкость.

```
// удалить точку по указанному индексу
    public void deletePoint(int index){
        // проверяем можно ли удалить точку
        // должно быть минимум 2 точки и индекс должен быть валидным
        if(index < 0 || index >= point_count || point_count <= 2){
```

```
            return;
        }
        // сдвигаем все точки после удаляемой влево
        if (index < point_count - 1) {
            System.arraycopy(point_mass, index + 1, point_mass, index,
point_count - index - 1);
        }
        // уменьшаем счетчик точек
        point_count--;
        // очищаем последний элемент
        point_mass[point_count] = null;
    }
    // добавить новую точку в функцию
    public void addPoint(FunctionPoint point) {
        int ins_index= 0;
        // ищем позицию для вставки чтобы сохранить возрастающий
порядок по x
        while (ins_index < point_count && point_mass[ins_index].getX()
< point.getX()) {
            ins_index++;
        }
        // если точка с таким x уже существует выходим
        if (ins_index < point_count && point_mass[ins_index].getX() ==
point.getX()) {
            return;
        }
        // проверяем нужно ли увеличивать массив
        if (point_count == point_mass.length) {
            // удваиваем размер массива
            int newcap = point_mass.length * 2;
            // если массив был пустой устанавливаем размер 1
            if (newcap == 0) newcap = 1;

            // создаем новый массив большего размера
            FunctionPoint[] newArray = new FunctionPoint[newcap];
            // копируем старые точки в новый массив
            System.arraycopy(point_mass, 0, newArray, 0, point_count);
            point_mass = newArray;
        }
        // сдвигаем точки чтобы освободить место для новой
        if (ins_index < point_count) {
            System.arraycopy(point_mass, ins_index, point_mass,
ins_index + 1, point_count - ins_index);
        }

        // вставляем новую точку
        point_mass[ins_index] = new FunctionPoint(point);
        // увеличиваем счетчик точек
        point_count++;
    }
```

Задание №7

Для проверки работоспособности всех реализованных классов я создал класс Main с методом main(). В тестовой программе была создана табулированная функция квадратичной зависимости $y = x^2$ на интервале [0, 10] с 5 точками. Затем последовательно выполнялись операции: установка значений Y, добавление новых точек, удаление точки, изменение координат существующих точек. Проверялась интерполяция в различных точках, в том числе внутри интервалов и на границах области определения. Также тестировалось поведение функции вне области определения. Все операции

демонстрируют корректную работу реализованных методов - точки сохраняют упорядоченность, интерполяция работает правильно, граничные случаи обрабатываются адекватно.

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
Квадратичная функция y = x^2:
(0,0, 0,0) (2,5, 6,3) (5,0, 25,0) (7,5, 56,3) (10,0, 100,0)
После добавления точек:
(0,0, 0,0) (1,5, 2,3) (2,5, 6,3) (3,5, 12,3) (5,0, 25,0) (7,5, 56,3) (10,0, 100,0)
После удаления точки:
(0,0, 0,0) (1,5, 2,3) (2,5, 6,3) (5,0, 25,0) (7,5, 56,3) (10,0, 100,0)
После изменения точки:
(0,0, 0,0) (1,5, 2,3) (3,0, 4,0) (5,0, 25,0) (7,5, 56,3) (10,0, 100,0)
Интерполяция:
f(0,5) = 0,75
f(1,2) = 1,80
f(2,5) = 3,42
f(3,2) = 6,10
f(6,8) = 47,50
Область определения: [0,0, 10,0]
Вне области определения:
f(-1) = NaN
f(11) = NaN
Финальное состояние:
(0,0, 0,0) (1,5, 2,3) (3,0, 4,0) (5,0, 25,0) (7,5, 56,3) (10,0, 100,0)

Process finished with exit code 0
```