Лабораторная работа №2

Выполнил работу

Гапанюк Антон Андреевич

Группа: 6204-010302D

Самара, 2025 г.

Цель работы: разработать набор классов для работы с функциями одной переменной, заданными в табличной форме.

Ход работы

Задание 1

Создаем пакет functions. В нем будем создавать классы для программы

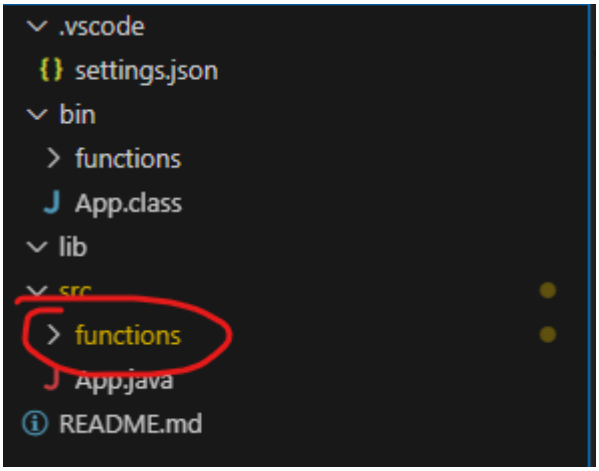

Рис. 1 – Создание пакета functions

Задание 2

В пакете functions создадим класс FunctionPoint, объект которого должен описывать одну точку табулированной функции.

```
public class FunctionPoint {
    // Переменные для хранения координат точкии
    private double x;
    private double y;
    // Конструктор создания объекта с заданными координатами
    FunctionPoint(double x, double y){
        this.x = x; // this обращается к полю текущего объекта
        this.y = y;
    }
    // Конструктор копирования существующей точки
    FunctionPoint(FunctionPoint point){
        this.x = point.x;
        this.y = point.y;
    }
    // Конструктор создания точки с координатами (0 , 0)
    FunctionPoint(){
        this.x = 0;
        this.y = 0;
    }
```

Не забудем про инкапсуляцию, создадим геттеры и сеттеры.

Задание 3

В пакете functions создадим класс TabulatedFunction, объект которого должен описывать табулированную функцию.

```
private FunctionPoint[] points;

// Конструктор создания объекта табулированной функции по области определения и количеству точек
TabulatedFunction(double leftX, double rightX, int pointsCount){

    // Провека на область определения
    if (rightX < leftX){
        throw new IllegalArgumentException(s:"Правая граница области определения меньше левой");
    }

    this.points = new FunctionPoint[pointsCount]; // Выдялем память для точек функции
    double step = ((rightX - leftX)/(pointsCount - 1)); // -1 -- Для корректности. Пример: левая точка: 1,
                                                        //  кол-во точек 5, промежуток между ними 4
    for(int i = 0; i < pointsCount; ++i){
        double x = leftX + i * step; // Рассчитываем x, учитывая "шаг"
        this.points[i] = new FunctionPoint(x, y:0); // Фиксируем x для каждой точкии
    }
}
```

Рис. 2 – Первый конструктор TabulatedFunction

Массив **points** будет хранить наши точки. В конструкторе сначала будем проверять, что область определения корректна, после выделяем память для точек функции. Рассчитываем шаг на основе информации о количестве точек.

В цикле **for** присваиваем каждой точке аргумент x.

```
// Конструктор создания объекта таб. функции по области определения и массиву значений функции
TabulatedFunction(double leftX, double rightX, double[] values){
    // Провека на область определения
    if (rightX < leftX){
        throw new IllegalArgumentException(s:"Правая граница области определения меньше левой");
    }

    int len = values.length; // Найдем длину массива и для удобства длину запишем в отдельную переменную
    this.points = new FunctionPoint[len];
    double step = ((rightX - leftX)/(len - 1));

    for(int i = 0; i < len; ++i){
        double x = leftX + i * step;
        this.points[i] = new FunctionPoint(x, values[i]);
    }
}
```

Рис. 3 – Второй конструктор TabulatedFunction

Так же проверяем область определения, но теперь в конструктор мы передаем массив значений функции, поэтому ищем длину и выделяем память на основе информации о длине массива.

В цикле **for** присваиваем точкам значения x и y.

Обратим внимание, что точки создаются через равные интервалы по x.

Задание 4

Опишем методы, необходимые для работы с функцией:

```
// Методы получения крайних значений области определения
    public double getLeftDomainBorder(){
        return points[0].getX();
    }
    public double getRightDomainBorder(){
        return points[points.length - 1].getX();
    }
```

Метод **getLeftDomainBorder()** возвращает левую границу области определения. **getRightDomainBorder()** – правую границу.

```
// Метод нахождения значения функции по аргументу x
public double getFunctionValue(double x){
    // Проверям, что x находится в области определения
    if (x >= getLeftDomainBorder() || x <= getRightDomainBorder()){
        for (int i = 0; i < points.length - 1; ++i){
```

Рис. 4 – Начало метода нахождения значения функции по x

Метод **getFunctionValue** будет находить значение функции по аргументу x. Сначала обязательно проверяем, что x находится в области определения, если да, то будем искать значение функции по x.

```
// Фиксируем точки для определения нахождения x
double x1 = points[i].getX();
double x2 = points[i+1].getX();

// Проверяем где находится x
if (x1 == x)
    return points[i].getY();
if (x2 == x)
    return points[i+1].getY();
if (x > x1 || x < x2){
    return linearInterpolation(x, x1, x2, points[i].getY(), points[i+1].getY());
}
```

Рис. 5 – Продолжение метода

Будем использовать линейную интерполяцию, то есть разбивать нашу функцию на ломаную. Значение функции будет находиться на одном из отрезков ломаной. Поэтому будем фиксировать крайние аргументы отрезка и проверять, не равен ли **x** этим аргументам, если нет, то воспользуемся уравнением прямой (рис. 6) по двум точкам и аргументом **x**. Найдем **y** по формуле:

```
// Приватный метод линейной интерполяции
private double linearInterpolation(double x, double x1, double x2, double y1, double y2){
    return y1 + (y2 - y1)/(x2 - x1)*(x - x1); // уравнение прямой по двум точкам
}
```

Рис. 6 – Нахождение y с помощью метода линейной интерполяции

Задание 5

Добавим методы, необходимые для работы с точками табулированной функции:

```
// Метод, возвращающий кол-во точек
    public int getPointsCount(){
        return points.length;
    }
// Метод создания копии точки по индексу
    public FunctionPoint getPoint(int index){
        return new FunctionPoint(points[index]);
    }
```

И еще методы…

```
// Функция замены указанной точки на переданную
public void setPoint(int index, FunctionPoint point){
    // Проверяем, лежит ли координата x вне интервала, если да, то ничего не делаем
    if (index > 0 && point.getX() <= points[index - 1].getX()){
        return;
    }
    if (index < points.length - 1 && point.getX() >= points[index + 1].getX()){
        return;
    }
    points[index] = new FunctionPoint(point);
}
```

Рис. 7 – Функция замены указанной точки на переданную

Логика метода такова, что нам важно, чтобы аргумент переданной точки не превышал аргумента соседней точки справа и был не меньше аргумента соседней точки слева, и только тогда меняем точку на переданную.

Добавим сеттеры и геттеры для работы с координатами точки указанного номера:

```java
// Метод возвращения абсциссы указанной точки
public double getPointX(int index){
    return points[index].getX();
}

// Метод установки нового значения абсциссы у конкретной точки
public void setPointX(int index, double x){
    if (index > 0 && x <= points[index - 1].getX()){
        return;
    }
    if (index < points.length - 1 && x >= points[index + 1].getX()){
        return;
    }
    points[index].setX(x);
}

// Метод возвращения ординаты указанной точки
public double getPointY(int index){
    return points[index].getY();
}

// Метод установки значения ординаты у конкретной точки
public void setPointY(int index, double y){
    points[index].setY(y);
}
```

Рис. 8 – Сеттеры и геттеры для работы с указанной точкой

Заметим, что в **setPointX** используется аналогичная проверка, что и **setPoint**. **setPointY** не требует такой проверки, нам не важно, какое значение устанавливать.

Задание 6

Опишем методы, изменяющие количество точек в таб. функции.

```java
// Метод удаления указанной точки
public void deletePoint(int index){

    FunctionPoint[] newPoints = new FunctionPoint[points.length - 1]; // Создаём новый массив на 1 элемент меньше
    System.arraycopy(points, srcPos:0, newPoints, destPos:0, index); // Копируем точки до удаляемой
    System.arraycopy(points, index + 1, newPoints, index, points.length - index - 1); // Копируем точки после удаляемой

    points = newPoints;
}
```

Рис. 9 – Метод удаления указанной точки

Будем создавать массив с длиной на единицу меньше. Затем использовать метод **arraycopy**, будем копировать исходный массив до точки, которую

нужно удалить, и вставлять в **newPoints**, затем копировать после точки и тоже вставлять в **newPoints**.

```java
// Метод добавления указанной точки
public void addPoint(FunctionPoint point){
    int insert_index = 0; // индекс, куда встанет новая точка
    while (insert_index < points.length && point.getX() > points[insert_index].getX()){
        insert_index++;
    }

    // Проверяем на дубликат
    if (insert_index < points.length && point.getX() == points[insert_index].getX())
        return;

    // Создаем массив на 1 элемент больше
    FunctionPoint[] newPoints = new FunctionPoint[points.length + 1];

    System.arraycopy(points, srcPos:0, newPoints, destPos:0, insert_index);
    newPoints[insert_index] = new FunctionPoint(point); // вставляем новую точку
    System.arraycopy(points, insert_index, newPoints, insert_index + 1, points.length - insert_index);

    points = newPoints;
}
```

Рис. 10 – Метод добавления указанной точки

Создадим переменную, которая будет хранить индекс, куда встанет новая точка. Определять куда встанет точка будем через цикл **while**, в котором будем сравнивать аргумент новой точки с аргументом соседней справа.

Добавим проверку на дубликат, чтобы не было двух точек с одинаковыми аргументами

Аналогичная процедура как и предыдущем методе. Используем arraycopy. Добавляем новую точку. Копируем оставшийся массив.

Задание 7 (Код main будет ниже)

Проверим работоспособность наших методов.

Начнем с вывода базовой информации о функции ($y = x^2$).

```
ФУНКЦИЯ x^2
Область определения: [0.0, 5.0]
Количество точек: 6
Точки функции:
[0] 0.0 0.0
[1] 1.0 1.0
[2] 2.0 4.0
[3] 3.0 9.0
[4] 4.0 16.0
[5] 5.0 25.0
```

Рис. 11 – Вывод инфы о функции

Как видим, все корректно отображается.

Протестируем значения функции в разных точках

double[] testPointsX = {-1, 0, 0.5, 1, 1.45, 1.6, 5, 6}:

```
ФУНКЦИЯ x^2
Значение функции не определено (аргумент x, мне области определения функции)
f(x) = 0.0
f(x) = 0.5
f(x) = 1.0
f(x) = 2.3499999999999996
f(x) = 2.8000000000000003
f(x) = 25.0
Значение функции не определено (аргумент x, мне области определения функции)
```

Рис. 12 – Вывод значений функции в разных точках

Методы работают корректно, функция не определена вне отрезка [0, 5] и правильно считает внутри отрезка с помощью метода линейной интерполяции.

Попробуем удалить точку и добавить:

```
ФУНКЦИЯ x^2
Количество точек до удаления: 6
Количество точек после удаления: 5
Все точки после удаления:
[0] 0.0 0.0
[1] 1.0 1.0
[2] 3.0 9.0
[3] 4.0 16.0
[4] 5.0 25.0
```

```
ФУНКЦИЯ x^2
Количество точек до добавления: 6
Количество точек после добавления точки (2.5; 6.25): 7
Количество точек после попытки добавить точку с существующим X = 3.0: 7
Все точки после добавления:
[0] 0.0 0.0
[1] 1.0 1.0
[2] 2.0 4.0
[3] 2.5 6.25
[4] 3.0 9.0
[5] 4.0 16.0
[6] 5.0 25.0
```

Рис. 13,14 – Удаление и добавление точки

```
До изменения точки с индексом 2:
[2] 2.5 6.25
После изменения Y точки 2 на 10.0:
[2] 2.5 10.0
Пытаемся изменить X точки 2 на 5.0 (не должно измениться):
[2] 2.5 10.0
После изменения X точки 2 на 2.2 (успешно):
[2] 2.2 10.0
После setPoint (заменяем указанную точку на переданную):
[2] 2.2 5.0
```

Рис. 15 – Тестирование изменения координат указанной точки

Методы изменения корректно отрабатывает.

Лабораторная работа выполнена

Примечание: код main

```
public class Main {
    public static void main(String[] args){
        // Создадим простейшую функцию y = x^2 на отрезке [0,
5]
        double[] values = {0, 1, 4, 9, 16, 25};

        TabulatedFunction f = new TabulatedFunction(0.0, 5.0,
values);

        System.out.println("ФУНКЦИЯ x^2");
        functionInfo(f);
        testFunctionValue(f);
        testDeletePoint(f);
        testAddPoint(f);
        testPointModification(f);



    }

    public static void functionInfo(TabulatedFunction f){
        System.out.println("Область определения: [" +
f.getLeftDomainBorder() + ", " + f.getRightDomainBorder() +
"]");
        System.out.println("Количество точек: " +
f.getPointsCount());

        System.out.println("Точки функции:");
        for (int i = 0; i < f.getPointsCount(); i++) {
            System.out.println("[" + i + "] " + f.getPointX(i)
+ " " + f.getPointY(i));
        }
    }

    public static void testFunctionValue(TabulatedFunction f){
        double[] testPointsX = {-1, 0, 0.5, 1, 1.45, 1.6, 5,
6};

        for (double x: testPointsX){
```

```
            double y = f.getFunctionValue(x);
            if (Double.isNaN(y))
                System.out.println("Значение функции не
определено (аргумент x, мне области определения функции)");
            else
                System.out.println("f(x) = " + y);

        }
    }

    // Тестирование удаления точки
    public static void testDeletePoint(TabulatedFunction f) {
        System.out.println("Количество точек до удаления: " +
f.getPointsCount());

        // Удаляем точку с индексом 2
        f.deletePoint(2);
        System.out.println("Количество точек после удаления: "
+ f.getPointsCount());

        System.out.println("Все точки после удаления:");
        for (int i = 0; i < f.getPointsCount(); i++) {
            System.out.println("[" + i + "] " + f.getPointX(i)
+ " " + f.getPointY(i));
        }
    }

    // Тестирование добавления точки
    public static void testAddPoint(TabulatedFunction f) {
        System.out.println("Количество точек до добавления: " +
f.getPointsCount());

        // Добавляем точку в середину
        f.addPoint(new FunctionPoint(2.5, 6.25));
        System.out.println("Количество точек после добавления
точки (2.5; 6.25): " + f.getPointsCount());

        // Добавляем точку с существующим X (не должна
добавиться)
        f.addPoint(new FunctionPoint(3.0, 100.0));
```

```
        System.out.println("Количество точек после попытки
добавить точку с существующим X = 3.0: " + f.getPointsCount());

        System.out.println("Все точки после добавления:");
        for (int i = 0; i < f.getPointsCount(); i++) {
            System.out.println("[" + i + "] " + f.getPointX(i)
+ " " + f.getPointY(i));
        }
    }

    // Тестирование изменения точек
    public static void testPointModification(TabulatedFunction
f) {
        System.out.println("До изменения точки с индексом 2:");
        System.out.println("[" + 2 + "] " + f.getPointX(2) + "
" + f.getPointY(2));

        // Изменяем Y точки
        f.setPointY(2, 10.0);
        System.out.println("После изменения Y точки 2 на
10.0:");
         System.out.println("[" + 2 + "] " + f.getPointX(2) + "
" + f.getPointY(2));

        // Изменяем X точки (неуспешно - нарушит порядок)
        System.out.println("Пытаемся изменить X точки 2 на 5.0
(не должно измениться):");
        f.setPointX(2, 5.0); // Не изменится, т.к. 5.0 >
следующей точки
         System.out.println("[" + 2 + "] " + f.getPointX(2) + "
" + f.getPointY(2));

        // Изменяем X точки на допустимое значение
        f.setPointX(2, 2.2);
        System.out.println("После изменения X точки 2 на 2.2
(успешно):");
        System.out.println("[" + 2 + "] " + f.getPointX(2) + "
" + f.getPointY(2));

        FunctionPoint newPoint = new FunctionPoint(2.2, 5.0);
```

```
        f.setPoint(2, newPoint);
        System.out.println("После setPoint (заменяем указанную
точку на переданную):");
        System.out.println("[" + 2 + "] " + f.getPointX(2) + "
" + f.getPointY(2));
    }
}
```