Лабораторная Работа №2 по Предмету ООП

Лабораторная Работа Выполнена:

Нурмагамбетовым Данилом Сериковичем

Группа: 6203-010302D

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

Task 2:

По заданию необходимо создать класс FunctionPoint, который имеет два поля, соответственно для координат X и У:

```java
package functions;

//  Представляет одну точку табулированной функции с координатами (x, y)
public class FunctionPoint {
    private double x;
    private double y;

    // Создаёт точку с координатами (0, 0).
    public FunctionPoint() {
        this.x = 0.0;
        this.y = 0.0;
    }

    // Создаёт точку с заданными координатами.
    public FunctionPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Конструктор копирования.
    public FunctionPoint(FunctionPoint other) {
        this.x = other.x;
        this.y = other.y;
    }

    public double getX() {return x;}
    public double getY() {return y;}

    public void setX(double x) {this.x = x;}
    public void setY(double y) {this.y = y;}
}
```

В Классе описаны следующие методы: Конструктор по умолчанию, где координаты нулевые, конструктор где указаны параметры для координат и Конструктор копирования, Геттеры и Сеттеры для координат также описаны.

Task 3:

По заданию нужно создать TabulatedFunction. реализующий табулированную функцию с точками, упорядоченными по x.

```java
package functions;

// Класс для представления табулированной функции одной переменной.
// Точки всегда хранятся в порядке возрастания x.
public class TabulatedFunction {
    private FunctionPoint[] points;   // массив с запасом ёмкости
    private int count;                // текущее количество точек

    public TabulatedFunction(double leftX, double rightX, int pointCount) {
        if (pointCount < 2) {
            throw new IllegalArgumentException(s:"Требуется минимум 2 точки.");
        }
        this.count = pointCount;
        this.points = new FunctionPoint[2 * count]; // выделяем память с запасом
        double step = (rightX - leftX) / (count - 1);
        for (int i = 0; i < count; i++) {
            points[i] = new FunctionPoint(leftX + step * i, y:0.0);
        }
    }

    public TabulatedFunction(double leftX, double rightX, double[] values) {
        if (values.length < 2) {
            throw new IllegalArgumentException(s:"Требуется минимум 2 значения.");
        }
        this.count = values.length;
        this.points = new FunctionPoint[2 * count];
        double step = (rightX - leftX) / (count - 1);
        for (int i = 0; i < count; i++) {
            points[i] = new FunctionPoint(leftX + step * i, values[i]);
        }
    }
```

Реализован класс TabulatedFunction, хранящий упорядоченный массив объектов FunctionPoint. Точки всегда расположены по возрастанию координаты x.
Два конструктора:

1.TabulatedFunction(double leftX, double rightX, int pointCount) — создаёт функцию с равномерно распределёнными по x точками, значения y инициализируются нулями,

2.TabulatedFunction(double leftX, double rightX, double[] values) — аналогичен первому, но значения y берутся из массива values.

Оба конструктора проверяют, что количество точек не меньше двух. Внутренний массив выделяется с **запасом памяти** (2 * count), что повышает эффективность при последующих вставках.

Task 4:

В TabulatedFunction создадим методы для того, чтобы работать с функцией:

```
public double getLeftLimit() {return points[0].getX();}
public double getRightLimit() {return points[count - 1].getX();}

public double getFunctionValue(double x) {
    if (x < points[0].getX() || x > points[count - 1].getX()) {
        return Double.NaN;
    }

    // Сначала проверяем точное совпадение — это ускоряет работу и улучшает точность
    for (int i = 0; i < count; i++) {
        if (Double.compare(points[i].getX(), x) == 0) {
            return points[i].getY();
        }
    }

    // Поиск интервала [i, i+1], в который попадает x
    int i = 0;
    while (i < count - 1 && points[i + 1].getX() < x) {
        i++;
    }

    // Линейная интерполяция между points[i] и points[i+1]
    double x1 = points[i].getX();
    double y1 = points[i].getY();
    double x2 = points[i + 1].getX();
    double y2 = points[i + 1].getY();
    return y1 + (y2 - y1) * (x - x1) / (x2 - x1);
}
```

getLeftLimit() и getRightLimit() возвращают x первой и последней точек.

getFunctionValue(double x) вычисляет значение функции с использованием линейной интерполяции.Перед интерполяцией выполняется проверка точного совпадения с помощью Double.compare() — это улучшает точность и устойчивость. Если x выходит за пределы области определения, возвращается Double.NaN.

Task 5:

В TabulatedFunction опишем нужные методы, чтобы работать с отдельными точками

```java
public int getPointsCount() {
    return count;
}

public FunctionPoint getPoint(int index) {
    return new FunctionPoint(points[index]); // возвращаем копию — инкапсуляция
}

public void setPoint(int index, FunctionPoint point) {
    double newX = point.getX();
    if (index == 0) {
        // Для первой точки достаточно, чтобы x был меньше следующей
        if (newX < points[1].getX()) {
            points[index] = new FunctionPoint(point);
        }
    } else if (index == count - 1) {
        // Для последней — больше предыдущей
        if (newX > points[count - 2].getX()) {
            points[index] = new FunctionPoint(point);
        }
    } else {
        // Для внутренних — должен лежать строго между соседями
        if (points[index - 1].getX() < newX && newX < points[index + 1].getX()) {
            points[index] = new FunctionPoint(point);
        }
    }
}

public double getPointX(int index) {
    return points[index].getX();
}
```

```java
public void setPointX(int index, double x) {
    setPoint(index, new FunctionPoint(x, points[index].getY()));
}

public double getPointY(int index) {
    return points[index].getY();
}

public void setPointY(int index, double y) {
    points[index].setY(y); // y не влияет на порядок — проверка не нужна
}
```

Метод getPoint(int index) возвращает копию объекта FunctionPoint, что предотвращает прямое изменение внутреннего состояния массива снаружи ,это гарантирует соблюдение инкапсуляции.

Метод setPoint(int index, FunctionPoint point) заменяет точку только в том случае, если её новая координата x не нарушает упорядоченность массива: для внутренних точек x должен лежать строго между соседними, для первой

и последней ,соответственно быть меньше следующей или больше предыдущей.

Методы setPointX и setPointY позволяют изменять координаты по отдельности. При этом setPointX использует логику setPoint и также проверяет корректность позиции, в то время как setPointY не требует проверок, так как изменение y не влияет на порядок точек.

Task 6:

В классе TabulatedFunction реализуем методы, изменяющие количество точек: deletePoint(int index) и addPoint(FunctionPoint point). Оба метода обеспечивают сохранение упорядоченности массива по координате x.

```
public void deletePoint(int index) {
    if (count <= 2) return; // нельзя оставить меньше двух точек
    for (int i = index; i < count - 1; i++) {
        points[i] = points[i + 1];
    }
    points[count - 1] = null; // избегаем утечки ссылок
    count--;
}
```

Метод deletePoint(int index) удаляет точку с указанным индексом. Перед удалением проверяется, что в функции останется не менее двух точек, по заданию. Удаление выполняется путём сдвига всех последующих элементов на одну позицию влево. После сдвига последний элемент массива явно обнуляется (points[count - 1] = null) для предотвращения утечки ссылок и упрощения работы сборщика мусора.

```java
public void addPoint(FunctionPoint point) {
    double x = point.getX();

    // Если x уже есть — заменяем значение (как указано в задании)
    for (int i = 0; i < count; i++) {
        if (Double.compare(points[i].getX(), x) == 0) {
            setPoint(i, point);
            return;
        }
    }

    // При нехватке места расширяем массив на 50%
    if (count >= points.length) {
        FunctionPoint[] newPoints = new FunctionPoint[points.length + (points.length / 2)];
        System.arraycopy(points, srcPos:0, newPoints, destPos:0, count);
        points = newPoints;
    }

    // Находим позицию для вставки, сохраняя порядок по x
    int insertIndex = 0;
    while (insertIndex < count && points[insertIndex].getX() < x) {
        insertIndex++;
    }

    // Сдвигаем хвост вправо
    System.arraycopy(points, insertIndex, points, insertIndex + 1, count - insertIndex);
    points[insertIndex] = new FunctionPoint(point); // копируем, а не присваиваем ссылку
    count++;
}
```

Метод addPoint(FunctionPoint point) добавляет новую точку в правильную позицию, сохраняя порядок по x. Если точка с таким же значением x уже существует, вызывается метод setPoint, что соответствует требованию задания ("set is used"). При необходимости расширения массива (если count >= points.length) создаётся новый массив с увеличенной ёмкостью на 50% (используется System.arraycopy). Новая точка вставляется в найденную позицию, а все последующие элементы сдвигаются вправо.

Task 7:

Создаем файл Main и запускаем программу.

```java
import functions.*;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Создаём функцию на основе произвольных значений
        double[] values = {1.0, 3.0, 5.0, 2.0, 6.0, 7.0, 26.0, 8.0};
        TabulatedFunction func = new TabulatedFunction(leftX:2.0, rightX:9.0, values);

        System.out.println(x:"Исходная функция:");
        func.printTabFun();

        System.out.println(x:"\n Удаляем точку с индексом 2");
        func.deletePoint(index:2);
        func.printTabFun();

        System.out.println(x:"\n Добавляем новую точку (0.0, -1.0)");
        func.addPoint(new FunctionPoint(x:0.0, -1.0));
        func.printTabFun();

        System.out.println(x:"\n Интерполяция в различных точках");
        double[] testX = {-1.0, 0.5, 3.0, 7.5, 9.0, 10.0};
        for (double x : testX) {
            double y = func.getFunctionValue(x);
            if (Double.isNaN(y)) {
                System.out.printf(format:"f(%.1f) = вне области определения%n", x);
            } else {
                System.out.printf(format:"f(%.1f) = %.3f%n", x, y);
            }
        }

        System.out.println(x:"\n Изменяем точку с индексом 1");
        func.setPoint(index:1, new FunctionPoint(x:1.0, y:100.0));
        func.printTabFun();

        System.out.println("\n Границы функции: [" +
                func.getLeftLimit() + "; " +
                func.getRightLimit() + "]");
    }
}
```

Результат:

```
PS C:\Users\user\Desktop\Lab-2-2025> cd "c:\Users\user\Desktop\Lab-2-2025\" ; if ($?) { javac Main.java } ; if ($?) { java Main }
Исходная функция:
 x: 2,000 y: 1,000
 x: 3,000 y: 3,000
 x: 4,000 y: 5,000
 x: 5,000 y: 2,000
 x: 6,000 y: 6,000
 x: 7,000 y: 7,000
 x: 8,000 y: 26,000
 x: 9,000 y: 8,000

 Удаляем точку с индексом 2
 x: 2,000 y: 1,000
 x: 3,000 y: 3,000
 x: 5,000 y: 2,000
 x: 6,000 y: 6,000
 x: 7,000 y: 7,000
 x: 8,000 y: 26,000
 x: 9,000 y: 8,000

 Добавляем новую точку (0.0, -1.0)
 x: 0,000 y: -1,000
 x: 2,000 y: 1,000
 x: 3,000 y: 3,000
 x: 5,000 y: 2,000
 x: 6,000 y: 6,000
 x: 7,000 y: 7,000
 x: 8,000 y: 26,000
 x: 9,000 y: 8,000

 Интерполяция в различных точках
f(-1,0) = вне области определения
f(0,5) = -0,500
f(3,0) = 3,000
f(7,5) = 16,500
f(9,0) = 8,000
f(10,0) = вне области определения

 Изменяем точку с индексом 1
 x: 0,000 y: -1,000
 x: 1,000 y: 100,000
 x: 3,000 y: 3,000
 x: 5,000 y: 2,000
 x: 6,000 y: 6,000
 x: 7,000 y: 7,000
 x: 8,000 y: 26,000
 x: 9,000 y: 8,000

 Границы функции: [0.0; 9.0]
PS C:\Users\user\Desktop\Lab-2-2025>
```