Лабораторная работа № 4

Титульный лист

Дисциплина: Объектно-ориентированное программирование
Выполнил: Мостовщиков Владимир Витальевич
Группа: 6204-010302D
Преподаватель: Борисов Дмитрий Сергеевич
Год: 2025

Содержание:

Задание 1. Реализация конструкторов из массива точек

В классах ArrayTabulatedFunction и LinkedListTabulatedFunction я реализовал конструкторы, получающие массив FunctionPoint. Проверяю, что точек не меньше двух, абсциссы строго возрастают с учётом EPSILON, входные точки

копируются внутрь, чтобы внешние изменения не ломали структуру.

```
// Конструктор получающий все точки сразу, в виде массива
public ArrayTabulatedFunction(FunctionPoint[] arr) {  5 usages  new *
    if (arr == null || arr.length < 2) // если точек < 2
        throw new IllegalArgumentException("At least 2 points required");
    // Проверка строгого порядка по x и отсутствия дублирования x
    for (int i = 1; i < arr.length; ++i) {
        double prevX = arr[i - 1].getX();
        double curX = arr[i].getX();
        if (!lt(prevX, curX)) {
            // Если curX == prevX или curX < prevX бросаем исключение
            throw new IllegalArgumentException("Points must be strictly increasing by x");
        }
    }
    // Копируем точки во внутренний массив
    this.size = arr.length;
    this.points = new FunctionPoint[Math.max(size, 2)];
    for (int i = 0; i < size; ++i) {
        this.points[i] = new FunctionPoint(arr[i]); // копия
    }
}
```

Рисунок 1 – реализация конструктора public ArrayTabulatedFunction(FunctionPoint[] arr)

```
// Конструктор из массива точек
public LinkedListTabulatedFunction(FunctionPoint[] arr) {  3 usages  new *
    this();
    if (arr == null || arr.length < 2) {
        throw new IllegalArgumentException("At least 2 points required");
    }
    // Строгий порядок по x
    for (int i = 1; i < arr.length; ++i) {
        if (!lt(arr[i - 1].getX(), arr[i].getX()))
            throw new IllegalArgumentException("Points must be strictly increasing by x");
    }
    // Добавляем копии точек
    for (FunctionPoint p : arr) {
        try {
            addPoint(new FunctionPoint(p));
        } catch (InappropriateFunctionPointException e) {
            throw new IllegalArgumentException("Invalid points order or duplicate x near " + p.getX(), e);
        }
    }
}
```

Рисунок 2 – реализация конструктора public LinkedListTabulatedFunction(FunctionPoint[] arr)

Задание 2. Интерфейсы Function и TabulatedFunction

Для выполнения 2-го задания я создал базовый интерфейс Function. TabulatedFunction теперь от него наследуется и оставляет только операции с табличными точками. Тем самым табулированная функция становится частным случаем обычной.

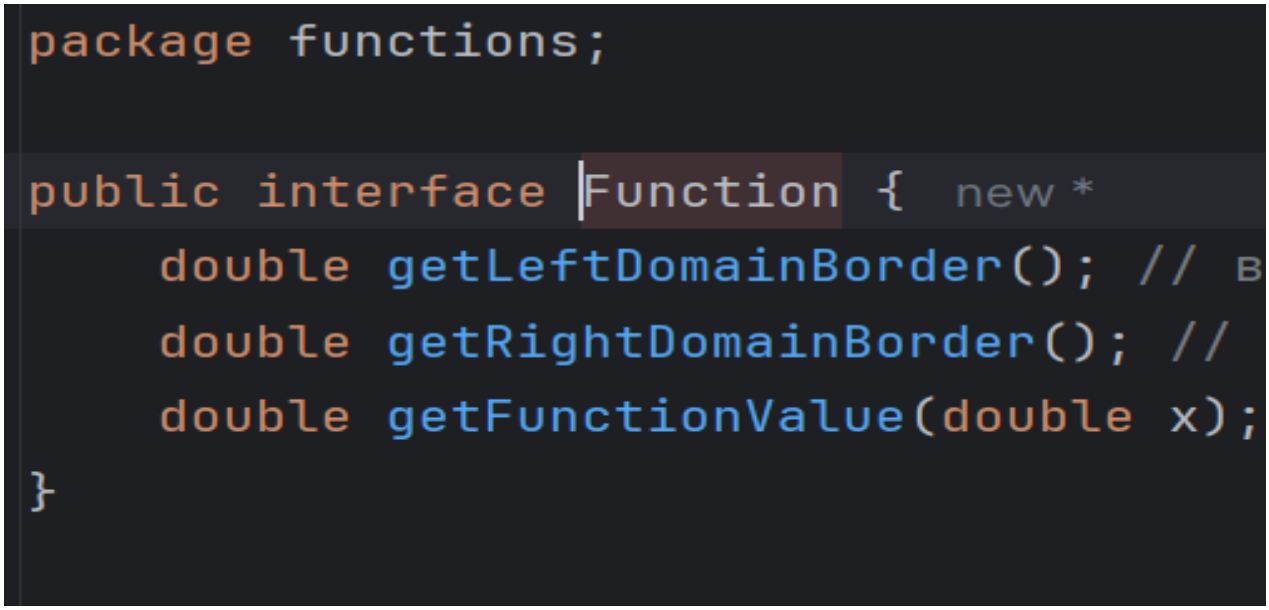


Рисунок 3 – Реализация интерфейса Function

Задание 3. Пакет functions.basic

В ходе выполнения задания были реализованы основные аналитические функции. Для экспоненциальной и логарифмической функций созданы отдельные классы. Функции синуса, косинуса и тангенса реализованы как наследники общего базового класса TrigonometricFunction, описывающего тригонометрические функции на всей числовой прямой.

Экспоненциальная функция (Exp) вычисляется с помощью Math.exp(x) и определена для всех вещественных значений аргумента.

Логарифмическая функция (Log) реализуется через формулу Math.log(x)/Math.log(base). При этом обязательно проводится проверка основания: основание должно быть положительным и не равно единице. Для значений аргумента, не превышающих ноль, функция возвращает NaN.

Тригонометрические функции (Sin, Cos, Tan) унаследованы от общего класса, описывающего поведение тригонометрических функций на всей

числовой оси. Значения вычисляются с помощью стандартных методов библиотеки Math.

```java
package functions.basic;
import functions.Function;
public class Log implements Function {  1 usage  new *
    private static final double EPSILON = 1e-12;  1 usage
    private final double base;  2 usages
    public Log(double base) {  3 usages  new *
        if (base <= 0.0 || Math.abs(base - 1.0) < EPSILON) {
            throw new IllegalArgumentException("Log base must be > 0 and != 1");
        }
        this.base = base;
    }
    @Override  new *
    public double getLeftDomainBorder() {return Double.MIN_VALUE;}
    @Override  new *
    public double getRightDomainBorder() {return Double.POSITIVE_INFINITY;}
    @Override  new *
    public double getFunctionValue(double x) {
        if (x <= 0.0) return Double.NaN;    // вне области определения
        return Math.log(x) / Math.log(base);
    }
}
```

Рисунок 4 – реализация класса Log

```java
package functions.basic;
import functions.Function;

public class Exp implements Function {  1 usage
    @Override  new *
    public double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }

    @Override  new *
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }

    @Override  new *
    public double getFunctionValue(double x) {
        return Math.exp(x);
    }
}
```

Рисунок 5 – реализация класса Exp

```java
package functions.basic;

public class Cos extends TrigonometricFunction {
    @Override  new *
    public double getFunctionValue(double x) {
        return Math.cos(x);
    }
}
```

Рисунок 6 – Реализация Cos

```java
package functions.basic;

public class Sin extends TrigonometricFunction {
    @Override  new *
    public double getFunctionValue(double x) {
        return Math.sin(x);
    }
}
```

Рисунок 7 – Реализация Sin

```java
package functions.basic;

public class Tan extends TrigonometricFunction {
    @Override  new *
    public double getFunctionValue(double x) {
        return Math.tan(x);
    }
}
```

Рисунок 8 – реализация Tan

```java
package functions.basic;

import functions.Function;

public abstract class TrigonometricFunction implements Function {
    @Override  new *
    public final double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }
    @Override  new *
    public final double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }
    @Override  3 implementations  new *
    public abstract double getFunctionValue(double x);
}
```

Рисунок 9 – реализация класса TrigonometricFunction

Задание 4. Пакет functions.meta

В рамках четвертого задания (пакет functions.meta) были реализованы функции-обёртки: Sum, Mult, Power, Scale, Shift и Composition. При построении функций особое внимание уделялось корректному определению области допустимых значений для результата, чтобы функция была определена только там, где это допустимо для всех исходных функций и параметров.

```
package functions.meta;

import functions.Function;

public class Composition implements Function {  1 usage   new *
    private final Function outer, inner;  2 usages

    public Composition(Function outer, Function inner) {  3 usages   new *
        if (outer == null || inner == null) throw new IllegalArgumentException("Functions must not be null");
        this.outer = outer; this.inner = inner;
    }

    @Override public double getLeftDomainBorder()  { return inner.getLeftDomainBorder(); }  new *
    @Override public double getRightDomainBorder() { return inner.getRightDomainBorder(); }  new *
    @Override public double getFunctionValue(double x) { return outer.getFunctionValue(inner.getFunctionValue(x)); }
}
```

Рисунок 10 – реализация Composition

```
package functions.meta;
import functions.Function;

public class Mult implements Function {  1 usage   new *
    private final Function f1, f2;  4 usages

    public Mult(Function f1, Function f2) {  3 usages   new *
        if (f1 == null || f2 == null) throw new IllegalArgumentException("Functions must not be null");
        this.f1 = f1; this.f2 = f2;
    }

    @Override public double getLeftDomainBorder()  { return Math.max(f1.getLeftDomainBorder(),  f2.getLeftDomainBorder()); }
    @Override public double getRightDomainBorder() { return Math.min(f1.getRightDomainBorder(), f2.getRightDomainBorder()); }
    @Override public double getFunctionValue(double x) { return f1.getFunctionValue(x) * f2.getFunctionValue(x); }  new *
}
```

Рисунок 11 – реализация класса Mult

```java
package functions.meta;

import functions.Function;

public class Power implements Function {  1 usage  new *
    private final Function f;  4 usages
    private final double p;  2 usages

    public Power(Function f, double p) {  3 usages  new *
        if (f == null) throw new IllegalArgumentException("Function must not be null");
        this.f = f; this.p = p;
    }

    @Override public double getLeftDomainBorder()  { return f.getLeftDomainBorder(); }  new *
    @Override public double getRightDomainBorder() { return f.getRightDomainBorder(); }  new *
    @Override public double getFunctionValue(double x) { return Math.pow(f.getFunctionValue(x), p); }
}
```

Рисунок 12 –Реализация Power

```java
package functions.meta;

import functions.Function;

public class Shift implements Function {  1 usage  new *
    private final Function f;  4 usages
    private final double shiftX, shiftY;  4 usages

    public Shift(Function f, double shiftX, double shiftY) {  3 usages  new *
        if (f == null) throw new IllegalArgumentException("Function must not be null");
        this.f = f; this.shiftX = shiftX; this.shiftY = shiftY;
    }

    @Override public double getLeftDomainBorder()  { return f.getLeftDomainBorder()  + shiftX; }  new *
    @Override public double getRightDomainBorder() { return f.getRightDomainBorder() + shiftX; }  new *
    @Override public double getFunctionValue(double x) { return f.getFunctionValue( x: x - shiftX) + shiftY; }
}
```

Рисунок 13 –Реализация Shift

```java
package functions.meta;
import functions.Function;
public class Sum implements Function {  1 usage  new *
    private final Function f1, f2;  4 usages

    public Sum(Function f1, Function f2) {  3 usages  new *
        if (f1 == null || f2 == null) throw new IllegalArgumentException("Functions must not be null");
        this.f1 = f1;
        this.f2 = f2;
    }
    @Override  new *
    public double getLeftDomainBorder() {
        return Math.max(f1.getLeftDomainBorder(), f2.getLeftDomainBorder());
    }
    @Override  new *
    public double getRightDomainBorder() { return Math.min(f1.getRightDomainBorder(), f2.getRightDomainBorder());}
    @Override  new *
    public double getFunctionValue(double x) { return f1.getFunctionValue(x) + f2.getFunctionValue(x);}
}
```

Рисунок 14 - Реализация Sum

```java
package functions.meta;
import functions.Function;

public class Scale implements Function {  1 usage  new *
    private static final double EPSILON = 1e-12;  1 usage

    private final Function f;  6 usages
    private final double scaleX, scaleY;  6 usages

    public Scale(Function f, double scaleX, double scaleY) {  3 usages  new *
        if (f == null) throw new IllegalArgumentException("Function must not be null");
        if (Math.abs(scaleX) < EPSILON) throw new IllegalArgumentException("scaleX must not be 0");
        this.f = f;
        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }
    @Override  new *
    public double getLeftDomainBorder() {
        double L = f.getLeftDomainBorder();
        double R = f.getRightDomainBorder();
        double a = L * scaleX;
        double b = R * scaleX;
        return Math.min(a, b); // учитываем возможный отрицательный scaleX
```

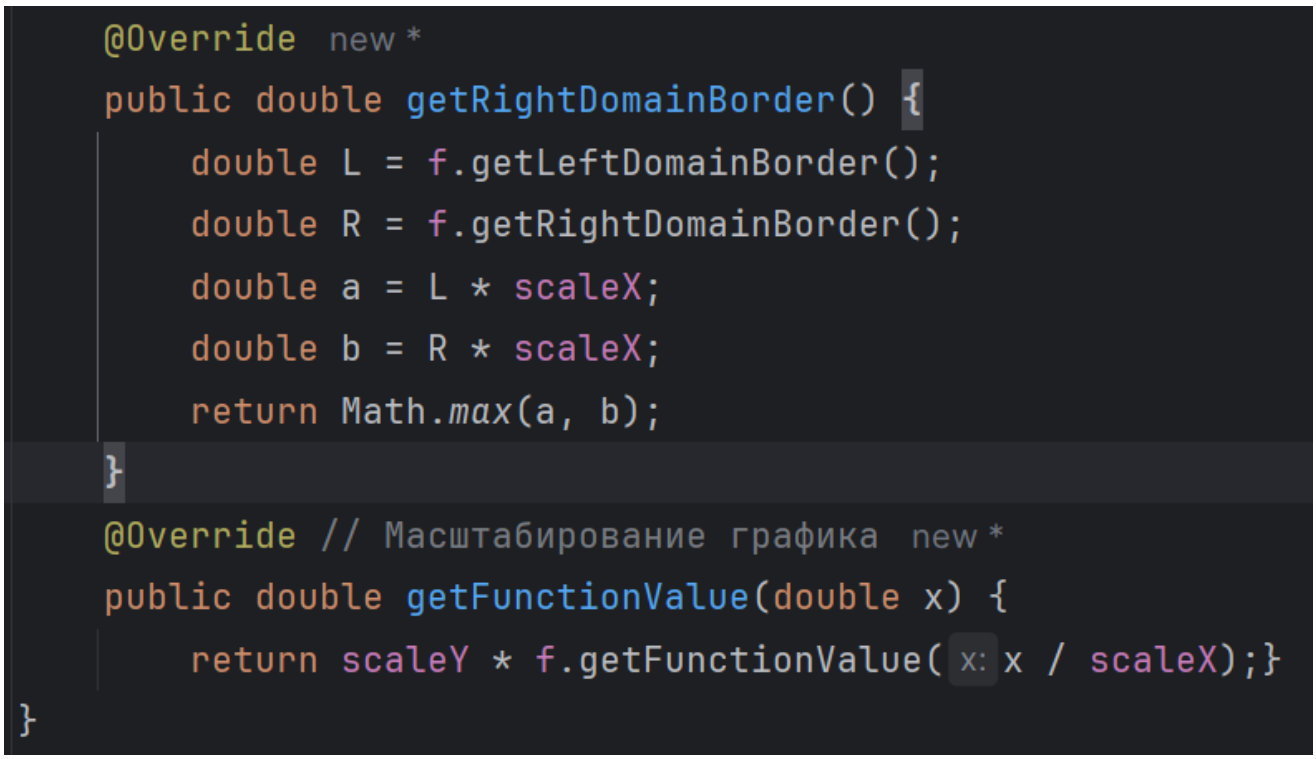


Рисунок 15,16 - Реализация Scale

Задание 5. Вспомогательный класс Functions

В пятом задании был реализован вспомогательный класс Functions, предназначенный для создания функций-обёрток с помощью специальных статических методов. Данный класс содержит в себе методы, обеспечивающие построение различных функций без необходимости явно создавать отдельные объекты для каждой операции. Создание экземпляров этого класса исключено на уровне реализации т.к. конструктор объявлен приватным и при попытке обращения выбрасывает исключение AssertionError. Это гарантирует, что все операции с функциями-обёртками выполняются только через статические методы данного класса.

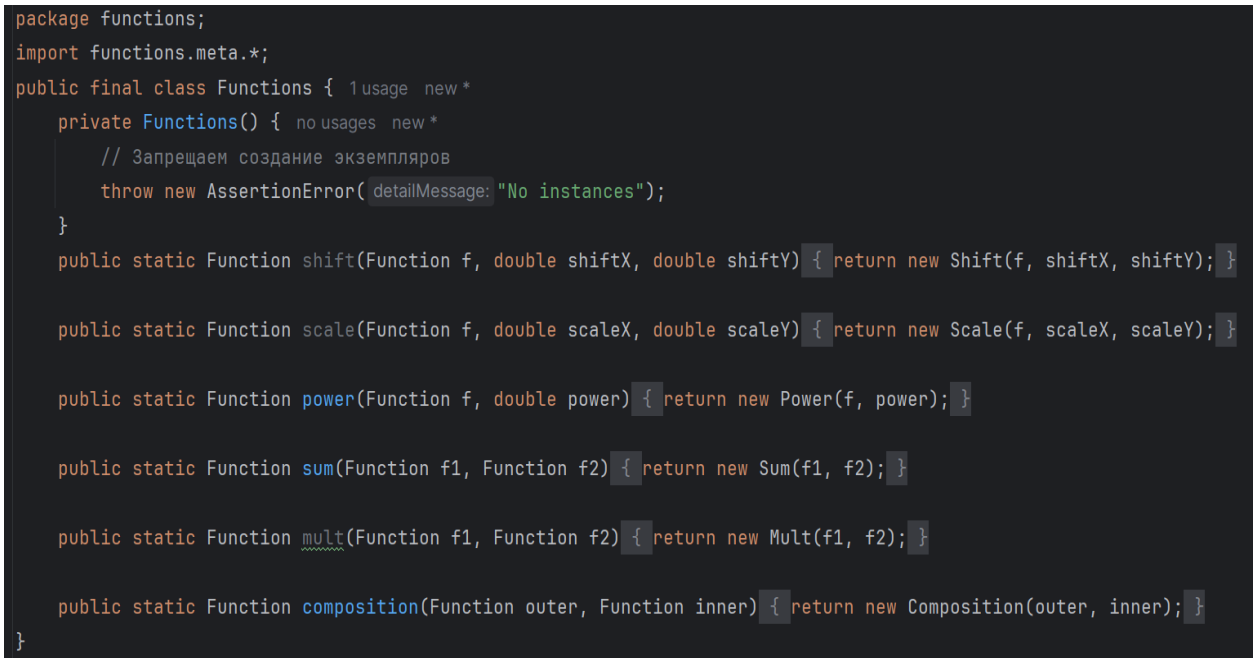


Рисунок 17 - Реализация класса Functions

Задание 6. Класс TabulatedFunctions.tabulate

В шестом задании реализован статический метод tabulate класса TabulatedFunctions, предназначенный для построения табличного представления произвольной функции на заданном интервале. Метод принимает на вход функцию f, границы интервала (left, right) и количество точек n.

Перед выполнением вычислений проводится строгая проверка входных параметров проверка на то, что значения границ интервала заданы корректно (левая граница строго меньше правой, число точек больше одного), дополнительно проверяю, что весь указанный интервал полностью принадлежит области определения функции (с учетом возможной погрешности, учитываемой через значение EPSILON).

После валидации границ формируется равномерное разбиение интервала на n точек. Для каждого такого значения аргумента вычисляется значение исходной функции. На выходе метод возвращает объект табулированной функции. Если параметры заданы некорректно, метод выкидывает соответствующие исключения, что позволяет своевременно выявлять и предотвращать ошибочные ситуации при построении табличного представления функции.

```java
package functions;
import java.io.*;
import java.util.ArrayList;
import java.util.List;
public final class TabulatedFunctions {  11 usages  new *
    private TabulatedFunctions() {  no usages  new *
        // Запрещаем создание экземпляров
        throw new AssertionError( detailMessage: "No instances");
    }

    // Сравнения с эпсилоном
    private static final double EPSILON = 1e-9;  2 usages
    private static boolean le(double a, double b){ return a <= b + EPSILON; }  1 usage  new *
    private static boolean ge(double a, double b){ return a >= b - EPSILON; }  1 usage  new *

    // Табуляция функции на отрезке
    public static TabulatedFunction tabulate(Function function, double leftX, double rightX, int pointsCount) {
        if (function == null) {throw new IllegalArgumentException("function is null");}
        if (pointsCount < 2) {throw new IllegalArgumentException("pointsCount must be >= 2");}
        if (!(leftX < rightX)) {throw new IllegalArgumentException("leftX must be < rightX");}
```

Рисунок 18 - класс TabulatedFunctions

```java
// Табуляция функции на отрезке
public static TabulatedFunction tabulate(Function function, double leftX, double rightX, int pointsCount) {
    if (function == null) {throw new IllegalArgumentException("function is null");}
    if (pointsCount < 2) {throw new IllegalArgumentException("pointsCount must be >= 2");}
    if (!(leftX < rightX)) {throw new IllegalArgumentException("leftX must be < rightX");}
    // Проверяем, что отрезок внутри области определения функции
    if (!ge(leftX, function.getLeftDomainBorder()) ||
        !le(rightX, function.getRightDomainBorder())) {
        throw new IllegalArgumentException("Tabulation segment lies outside function domain");}

    double step = (rightX - leftX) / (pointsCount - 1);
    FunctionPoint[] pts = new FunctionPoint[pointsCount];
    for (int i = 0; i < pointsCount; ++i) {
        double x = leftX + i * step;
        double y = function.getFunctionValue(x);
        pts[i] = new FunctionPoint(x, y);}
    return new ArrayTabulatedFunction(pts);}
```

Рисунок 19 - Реализация метода tabulate класса TabulatedFunctions

Задание 7. Ввод/вывод табулированных функций

В седьмом задании реализованы оба способа хранения табулированных функций: бинарный и текстовый форматы. Каждый вариант предусматривает отдельные методы для записи и чтения данных, а также учитывает обработку ошибок ввода-вывода. Для бинарного варианта реализованы следующие методы:

outputTabulatedFunction осуществляет последовательную запись количества точек (целое число n), после чего в поток поочередно записываются n пар значений (x, y).

inputTabulatedFunction читает из входного потока число точек, затем извлекает все пары координат и на их основе формирует новый объект табулированной функции.

Для текстового представления реализованы следующие методы:

writeTabulatedFunction выводит в одну строку количество точек, а далее через пробелы все значения x и y по порядку.

readTabulatedFunction читает строку из потока, последовательно, разбирая все числа с помощью класса StreamTokenizer, и формирует табулированную функцию на их основе.

```
 // Бинарный вывод пишет N, затем пары (x, y) для всех точек
public static void outputTabulatedFunction(TabulatedFunction function, OutputStream out) {
    try {
        DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(out));
        int n = function.getPointsCount(); // Количество точек в функции
        dos.writeInt(n); // Сначала пишем это количество
        for (int i = 0; i < n; ++i) {
            // Для каждой точки записываем x и y как double
            dos.writeDouble(function.getPointX(i));
            dos.writeDouble(function.getPointY(i));
        }
        dos.flush(); // Принудительно записываем данные в поток
    } catch (IOException e) { // Если произошла ошибка
        throw new UncheckedIOException(e);
    }
}
```

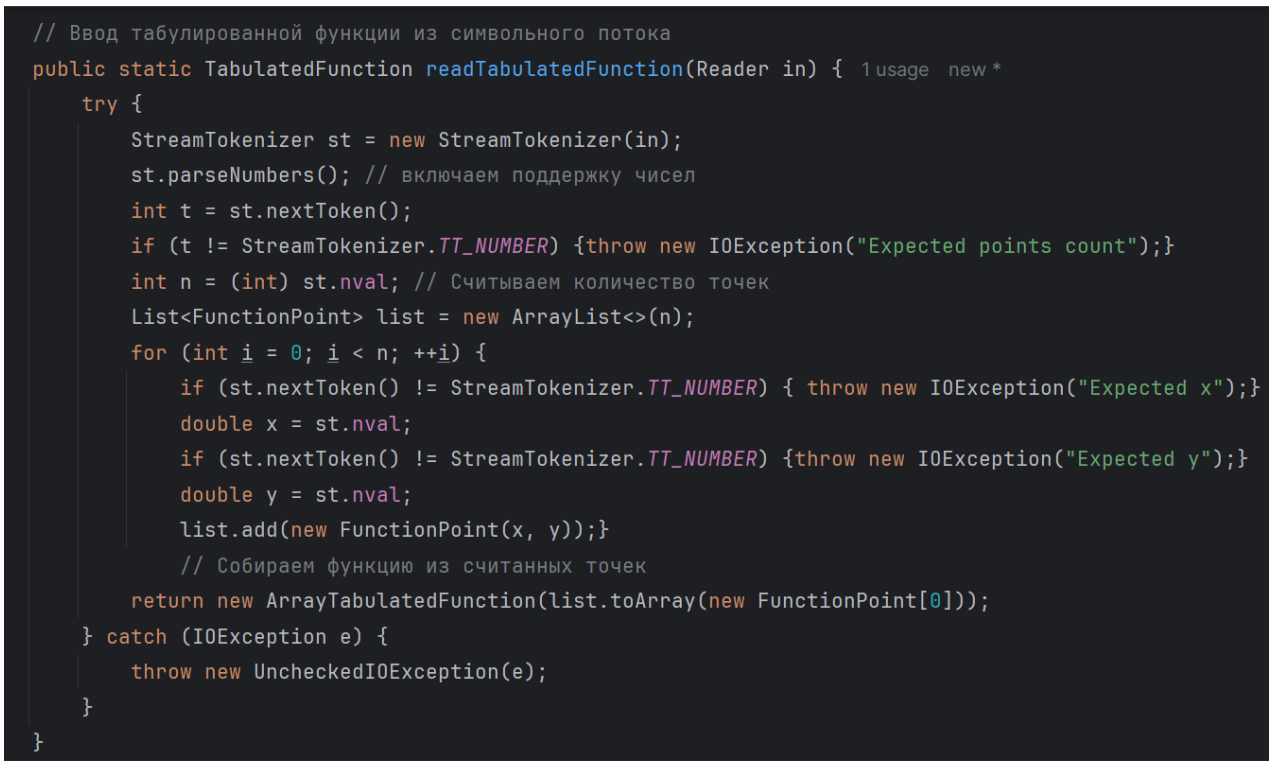
Рисунок 20 - Реализация метода outputTabulatedFunction

```java
// Бинарный ввод читает N, затем N пар (x, y), собирает TabulatedFunction
public static TabulatedFunction inputTabulatedFunction(InputStream in) { 1 usage  new *
    try {
        // Оборачиваем поток для удобного чтения
        DataInputStream dis = new DataInputStream(new BufferedInputStream(in));
        int n = dis.readInt();
        FunctionPoint[] pts = new FunctionPoint[n];
        for (int i = 0; i < n; ++i) {
            double x = dis.readDouble();
            double y = dis.readDouble();
            pts[i] = new FunctionPoint(x, y);
        }
         // Возвращаем новую табулированную функцию на основе прочитанного массива точек
        return new ArrayTabulatedFunction(pts);
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

Рисунок 21 - Реализация метода writeTabulatedFunction

```java
// Запись табулированной функции в символьный поток
public static void writeTabulatedFunction(TabulatedFunction function, Writer out) {
    PrintWriter pw = new PrintWriter(new BufferedWriter(out));
    int n = function.getPointsCount(); // Сначала выводим количество точек
    pw.print(n);
    for (int i = 0; i < n; ++i) {
        // Затем для каждой точки: x и y через пробел
        pw.print(' ');
        pw.print(function.getPointX(i));
        pw.print(' ');
        pw.print(function.getPointY(i));
    }
    pw.println(); // Завершаем строку
    pw.flush(); // Выгружаем данные в поток
}
```

Рисунок 22 - Реализация метода inputTabulatedFunction



```
// Ввод табулированной функции из символьного потока
public static TabulatedFunction readTabulatedFunction(Reader in) {  1 usage   new *
    try {
        StreamTokenizer st = new StreamTokenizer(in);
        st.parseNumbers(); // включаем поддержку чисел
        int t = st.nextToken();
        if (t != StreamTokenizer.TT_NUMBER) {throw new IOException("Expected points count");}
        int n = (int) st.nval; // Считываем количество точек
        List<FunctionPoint> list = new ArrayList<>(n);
        for (int i = 0; i < n; ++i) {
            if (st.nextToken() != StreamTokenizer.TT_NUMBER) { throw new IOException("Expected x");}
            double x = st.nval;
            if (st.nextToken() != StreamTokenizer.TT_NUMBER) {throw new IOException("Expected y");}
            double y = st.nval;
            list.add(new FunctionPoint(x, y));}
            // Собираем функцию из считанных точек
        return new ArrayTabulatedFunction(list.toArray(new FunctionPoint[0]));
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

Рисунок 23 - Реализация метода inputTabulatedFunction

В реализованных методах ввода и вывода все возникающие исключения типа IOException не выбрасываются напрямую, а преобразуются в UncheckedIOException. Такой подход избавляет код от необходимости явно обрабатывать проверяемые исключения на каждом этапе работы, что способствует повышению читаемости и удобства использования методов. Управление жизненным циклом потоков полностью возложено на того, кто данный поток создает, поэтому методы не закрывают переданные им потоки. Это позволяет избежать риска преждевременного или повторного закрытия потока.

Задание 8. Проверка работы в Main.

Проверим работу функций в Main:
1) Аналитические функции, создаем Sin и Cos. Печатаем их значения на [0; PI] с шагом 0.1, чтобы показать исходные значения.

```
=== 1) Аналитические sin и cos на [0, pi] шаг 0.1
sin domain = [-Infinity, Infinity]
cos domain = [-Infinity, Infinity]
x= 0,0  sin=0,000000  cos=1,000000
x= 0,1  sin=0,099833  cos=0,995004
x= 0,2  sin=0,198669  cos=0,980067
x= 0,3  sin=0,295520  cos=0,955336
x= 0,4  sin=0,389418  cos=0,921061
x= 0,5  sin=0,479426  cos=0,877583
x= 0,6  sin=0,564642  cos=0,825336
x= 0,7  sin=0,644218  cos=0,764842
x= 0,8  sin=0,717356  cos=0,696707
x= 0,9  sin=0,783327  cos=0,621610
x= 1,0  sin=0,841471  cos=0,540302
```

Рисунок 24 – Вывод аналитических значений sin/cos

2) Табулирование и сравнение с аналитикой ,табулирует sin и cos на [0; π] по 10 точкам. Для каждого x с шагом 0.1 сравнивает табличные значения с аналитическими и выводит максимальные отклонения.

```
=== 2) Табулированные sin/cos vs аналитические точки ===
x= 0,0  tsin=0,000000  sin=0,000000   tcos=1,000000  cos=1,000000
x= 0,1  tsin=0,097982  sin=0,099833   tcos=0,982723  cos=0,995004
x= 0,2  tsin=0,195963  sin=0,198669   tcos=0,965446  cos=0,980067
x= 0,3  tsin=0,293945  sin=0,295520   tcos=0,948170  cos=0,955336
x= 0,4  tsin=0,385907  sin=0,389418   tcos=0,914355  cos=0,921061
x= 0,5  tsin=0,472070  sin=0,479426   tcos=0,864608  cos=0,877583
x= 0,6  tsin=0,558234  sin=0,564642   tcos=0,814862  cos=0,825336
x= 0,7  tsin=0,643982  sin=0,644218   tcos=0,764620  cos=0,764842
x= 0,8  tsin=0,707935  sin=0,717356   tcos=0,688404  cos=0,696707
x= 0,9  tsin=0,771888  sin=0,783327   tcos=0,612188  cos=0,621610
x= 1,0  tsin=0,835841  sin=0,841471   tcos=0,535972  cos=0,540302
```

```
x= 2,8  tsin=0,334698  sin=0,334988   tcos=-0,940984  cos=-0,942222
x= 2,9  tsin=0,236716  sin=0,239249   tcos=-0,958261  cos=-0,970958
x= 3,0  tsin=0,138735  sin=0,141120   tcos=-0,975537  cos=-0,989992
x= 3,1  tsin=0,040753  sin=0,041581   tcos=-0,992814  cos=-0,999135
max|sin - tsin| = 0,0147659, max|cos - tcos| = 0,0146202
```

Рисунок 24,25 – Вывод табулированных значений sin/cos и сравнение с аналитическими значениями

3) Анализ точности выражения $\sin^2(x) + \cos^2(x)$

```
=== 3) (tsin)^2 + (tcos)^2 и влияние количества  точек на точность вычислений
n= 5  max|sin^2+cos^2 - 1| = 0,146445
n=10  max|sin^2+cos^2 - 1| = 0,0301537
n=25  max|sin^2+cos^2 - 1| = 0,00427757
n=50  max|sin^2+cos^2 - 1| = 0,00102728
n=75  max|sin^2+cos^2 - 1| = 0,000450513
n=100  max|sin^2+cos^2 - 1| = 0,000251729
```

Рисунок 26 – Демонстрирует зависимость погрешности от количества выбранных точек

4) Проверка метафункций. В частности, проверяется, что композиция натурального логарифма и экспоненты (composition(ln, exp)) вычисляет функцию, идентичную x, что иллюстрирует правильную работу композиции

функций.

```
=== 4) Meta-функции Sum, Mult, Power, Scale, Shift, Composition ===
sum domain       = [-Infinity, Infinity]
mult domain      = [-Infinity, Infinity]
power(sin,2) dom  = [-Infinity, Infinity]
scale dom        = [-Infinity, Infinity]
shift dom        = [-Infinity, Infinity]
composition dom   = [-Infinity, Infinity]
x=-3,142  sum=-1,000000  mult=0,000000  sin^2=0,000000  scale=3,000000  shift=-1,040302  comp=-3,141593
x=-1,000  sum=-0,301169  mult=-0,454649  sin^2=0,708073  scale=1,438277  shift=-0,916147  comp=-1,000000
x=-0,500  sum=0,398157  mult=-0,420735  sin^2=0,229849  scale=0,742212  shift=-0,429263  comp=-0,500000
x= 0,000  sum=1,000000  mult=0,000000  sin^2=0,000000  scale=-0,000000  shift=0,040302  comp=0,000000
x= 0,500  sum=1,357008  mult=0,420735  sin^2=0,229849  scale=-0,742212  shift=0,377583  comp=0,500000
x= 1,000  sum=1,381773  mult=0,454649  sin^2=0,708073  scale=-1,438277  shift=0,500000  comp=1,000000
x= 3,142  sum=-1,000000  mult=-0,000000  sin^2=0,000000  scale=-3,000000  shift=-1,040302  comp=3,141593
```

Рисунок 27 - Демонстрация значений различных метафункций и их областей определения на тестовых точках.

5) Проверка ошибок при создании табулированных функций**,** в коде отдельно рассматриваются ситуации, при которых функции не должны корректно строиться:

• попытка табулировать натуральный логарифм на отрезке, включающем 0,
• указание некорректного отрезка (левая и правая граница совпадают),
• слишком малое число точек для табуляции (меньше двух).

Во всех этих случаях проверяется, что программа корректно выбрасывает исключение IllegalArgumentException.

```
=== 5) Tabulate: проверки границ области определения ===
Ошибка: ожидалось исключение для ln на [0,10]
OK: degenerate отрезок отклонён
OK: pointsCount<2 отклонён
```

Рисунок 28 - Результаты проверки корректности обработки недопустимых аргументов при табулировании функций

6) В шестом шаге программы происходит проверка корректности текстового ввода-вывода табулированной функции. Функция exp(x) сначала табулируется на отрезке [0; 10] с шагом 1, результаты сохраняются в

текстовый файл, затем читаются обратно из этого файла. После этого для каждого x сравнивается исходное и считанное значение функции, чтобы убедиться, что операция записи-чтения не приводит к потере точности.

```
=== 6) Text IO: write/read exp на [0,10] с шагом 1 ===
x= 0  write=1,000000000  read=1,000000000  diff=0,00
x= 1  write=2,718281828  read=2,718281828  diff=0,00
x= 2  write=7,389056099  read=7,389056099  diff=0,00
x= 3  write=20,085536923  read=20,085536923  diff=0,00
x= 4  write=54,598150033  read=54,598150033  diff=7,11e-15
x= 5  write=148,413159103  read=148,413159103  diff=0,00
x= 6  write=403,428793493  read=403,428793493  diff=0,00
x= 7  write=1096,633158428  read=1096,633158428  diff=0,00
x= 8  write=2980,957987042  read=2980,957987042  diff=0,00
x= 9  write=8103,083927575  read=8103,083927575  diff=0,00
x=10  write=22026,465794807  read=22026,465794807  diff=3,64e-12
```

Рисунок 29 - Сравнение значений функции exp(x), полученных до и после записи в текстовый файл

7) В шаге 7 я проверяю, как работает сохранение и восстановление табулированной функции ln(x) в бинарном формате.

```
=== 7) Binary IO: output/input ln на [1e-4,10] с шагом 1
x= 1  write=-0,000738951  read=-0,000738951  diff=0,00
x= 2  write=0,693131731  read=0,693131731  diff=0,00
x= 3  write=1,098607240  read=1,098607240  diff=0,00
x= 4  write=1,386292100  read=1,386292100  diff=0,00
x= 5  write=1,609436755  read=1,609436755  diff=0,00
x= 6  write=1,791758843  read=1,791758843  diff=0,00
x= 7  write=1,945909810  read=1,945909810  diff=0,00
x= 8  write=2,079441371  read=2,079441371  diff=0,00
x= 9  write=2,197224511  read=2,197224511  diff=0,00
x=10  write=2,302585093  read=2,302585093  diff=0,00
```

Рисунок 30 - Проверка бинарного ввода-вывода: значения табулированной функции ln(x) до записи (write) и после чтения

Задание 9. Сериализация: Serializable и Externalizable

На этом этапе я тестирую две разные схемы сериализации табулированных функций:

- Serializable (стандартная сериализация для классов типа ArrayTabulatedFunction)
- Externalizable (ручное управление сериализацией для LinkedListTabulatedFunction)

В ходе проверки объект табулированной функции записывается в файл, затем считывается обратно, и значения функции сравниваются до и после сериализации. В обоих случаях максимальное расхождение между значениями составляет ровно ноль, то есть сериализация полностью корректна и не приводит к потере данных.

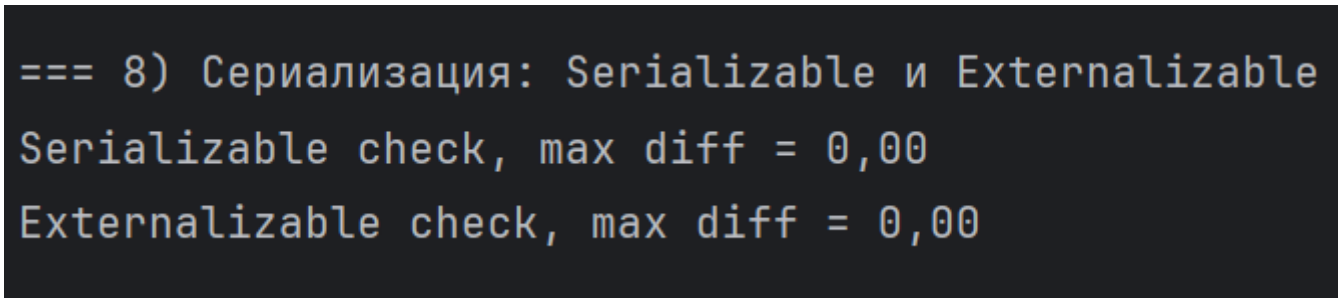


Рисунок 31 – сериализация табулированных функций

Вывод по итогам сравнения форматов хранения:

- **Текстовый формат (write/read):**
  Удобен для просмотра и ручного редактирования, но возможна минимальная потеря точности из-за преобразования чисел в строку и обратно (видно только на очень больших или маленьких значениях). Файлы больше по размеру, но читаемы человеком.
- **Бинарный формат (output/input):**
  Компактнее, обеспечивает точную передачу данных double без потерь, удобен для быстрого сохранения и загрузки больших объёмов данных. Файл нечитаем для человека, изменить вручную невозможно.
- **Serializable/Externalizable:**
  Позволяют сохранять целые объекты со всеми внутренними полями, напрямую через стандартные механизмы Java.
  Сохраняют структуру и все поля объекта бит-в-бит, максимально удобно при сложных типах данных.
  Недостаток: сериализованный файл жёстко связан с версией класса (при изменении структуры класса файл может стать нечитабельным).
  Файлы компактные, чтение/запись очень быстрые, читаемость человеком — отсутствует.