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 № 4

«Расширение пакета для работы с функциями: аналитические функции, операции и сериализация»

по курсу
Объектно-ориентированное программирование

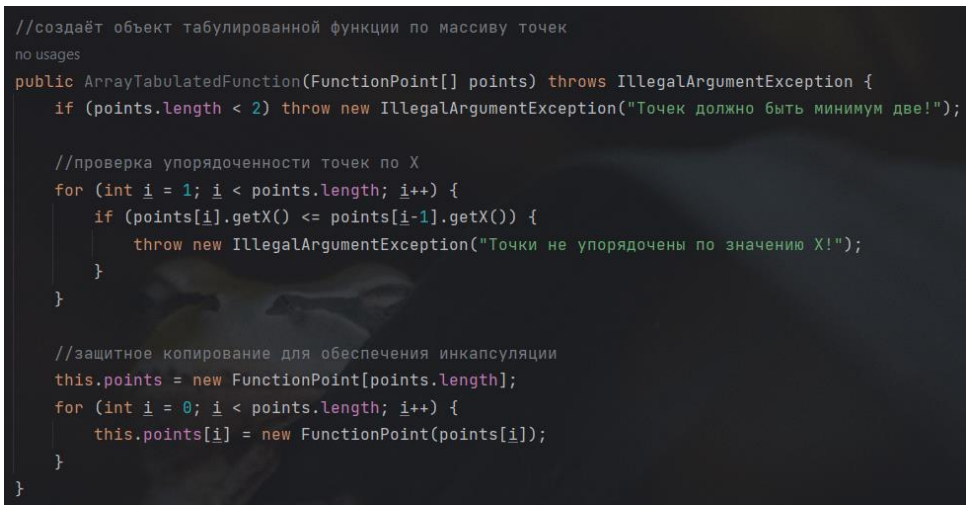
Выполнила: Яньшина Анастасия Юрьевна
Группа 6203-010302D

Содержание

Титульный лист 1
Содержание 2
Задание 1 3
Задание 2 4
Задание 3 5
Задание 4 10
Задание 5 15
Задание 6 16
Задание 7 17
Задание 8 19
Задание 9 27

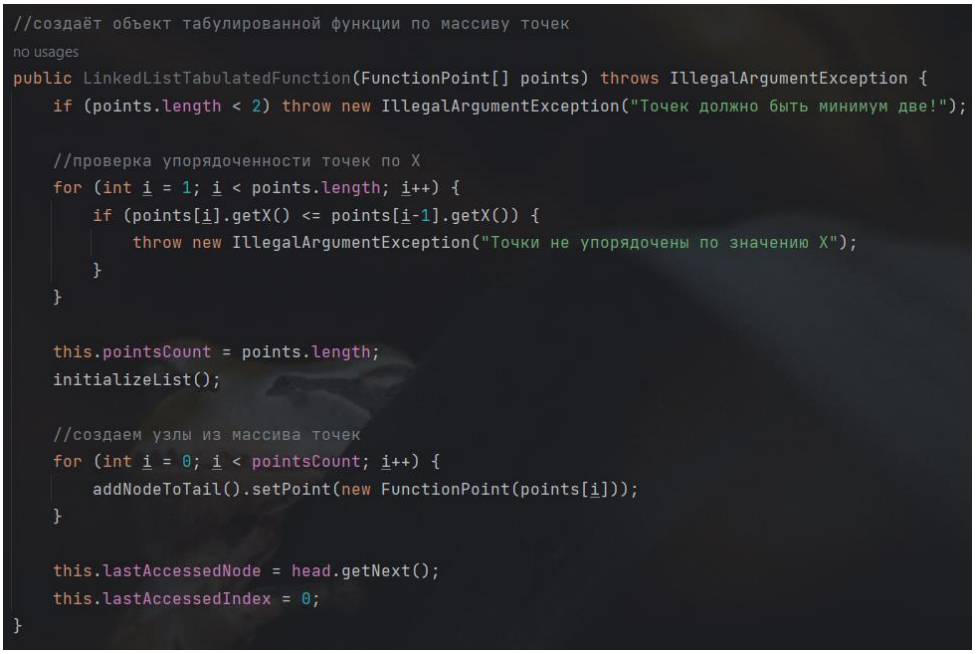
Задание 1

Начинаем с добавления новых конструкторов в классы `ArrayTabulatedFunction` и `LinkedListTabulatedFunction` (Рис. 1 и рис. 2). Для сохранения инкапсуляции в первом классе создаём новый массив точек и копируем их туда, а во втором создаём новые точки, используя параметры старых.



```
//создаёт объект табулированной функции по массиву точек
no usages
public ArrayTabulatedFunction(FunctionPoint[] points) throws IllegalArgumentException {
    if (points.length < 2) throw new IllegalArgumentException("Точек должно быть минимум две!");

    //проверка упорядоченности точек по X
    for (int i = 1; i < points.length; i++) {
        if (points[i].getX() <= points[i-1].getX()) {
            throw new IllegalArgumentException("Точки не упорядочены по значению X!");
        }
    }

    //защитное копирование для обеспечения инкапсуляции
    this.points = new FunctionPoint[points.length];
    for (int i = 0; i < points.length; i++) {
        this.points[i] = new FunctionPoint(points[i]);
    }
}
```

Рис. 1



```
//создаёт объект табулированной функции по массиву точек
no usages
public LinkedListTabulatedFunction(FunctionPoint[] points) throws IllegalArgumentException {
    if (points.length < 2) throw new IllegalArgumentException("Точек должно быть минимум две!");

    //проверка упорядоченности точек по X
    for (int i = 1; i < points.length; i++) {
        if (points[i].getX() <= points[i-1].getX()) {
            throw new IllegalArgumentException("Точки не упорядочены по значению X");
        }
    }

    this.pointsCount = points.length;
    initializeList();

    //создаем узлы из массива точек
    for (int i = 0; i < pointsCount; i++) {
        addNodeToTail().setPoint(new FunctionPoint(points[i]));
    }

    this.lastAccessedNode = head.getNext();
    this.lastAccessedIndex = 0;
}
```

Рис. 2

Задание 2

В пакете `functions` создаём интерфейс `Function` (Рис. 3), а также меняем уже существующий интерфейс `TabulatedFunction` таким образом, чтобы он расширял интерфейс `Function` (Рис. 4).

После изменений табулированные функции будут частным случаем функций одной переменной.

```
package functions;

no usages
public interface Function {
    no usages
    double getLeftDomainBorder();
    no usages
    double getRightDomainBorder();
    no usages
    double getFunctionValue(double x);
}
```

Рис. 3

```
package functions;

14 usages  2 implementations
public interface TabulatedFunction extends Function {
    |
    3 usages  2 implementations
    int getPointsCount();
    2 usages  2 implementations
    FunctionPoint getPoint(int index) throws FunctionPointIndexOutOfBoundsException;
    4 usages  2 implementations
    void setPoint(int index, FunctionPoint point) throws FunctionPointIndexOutOfBoundsException, InappropriateFunctionPointExce
    4 usages  2 implementations
    double getPointX(int index) throws FunctionPointIndexOutOfBoundsException;
    1 usage  2 implementations
    void setPointX(int index, double x) throws FunctionPointIndexOutOfBoundsException, InappropriateFunctionPointException;
    1 usage  2 implementations
    double getPointY(int index) throws FunctionPointIndexOutOfBoundsException;
    5 usages  2 implementations
    void setPointY(int index, double y) throws FunctionPointIndexOutOfBoundsException;
    3 usages  2 implementations
    void deletePoint(int index) throws FunctionPointIndexOutOfBoundsException, IllegalStateException;
    2 usages  2 implementations
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;
}
```

Рис. 4

Задание 3

Следующим шагом создаём пакет `functions.basic` для описания классов ряда функций, заданных аналитически (Рис. 5). В этом пакете создаём и реализуем классы `Exp` и `Log` для вычисления значения экспоненты и значения логарифма по заданному основанию соответственно. Для вычисления значений пользуемся методами из `Math`. Также реализуем обоим классам методы для возвращения границ области определения (Рис. 6 и рис. 7).

Для следующих функций сначала создаём базовый класс `TrigonometricFunction`, реализующий интерфейс `Function` и описывающий методы получения границ области определения (Рис. 8).

Только после этого создаём наследующие от него публичные классы `Sin`, `Cos` и `Tan`, объекты которых вычисляют, соответственно, значения синуса, косинуса и тангенса (Рис. 9, 10 и 11). Для получения значений вновь пользуемся методами `Math`:
`Math.sin()`, `Math.cos()` и `Math.tan()`.

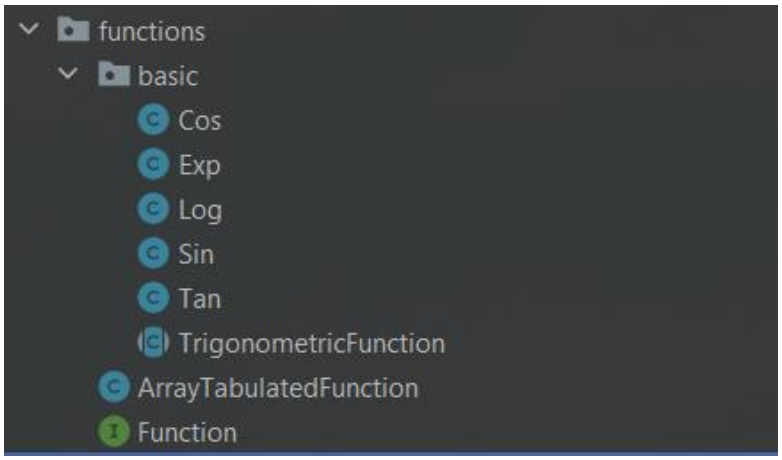
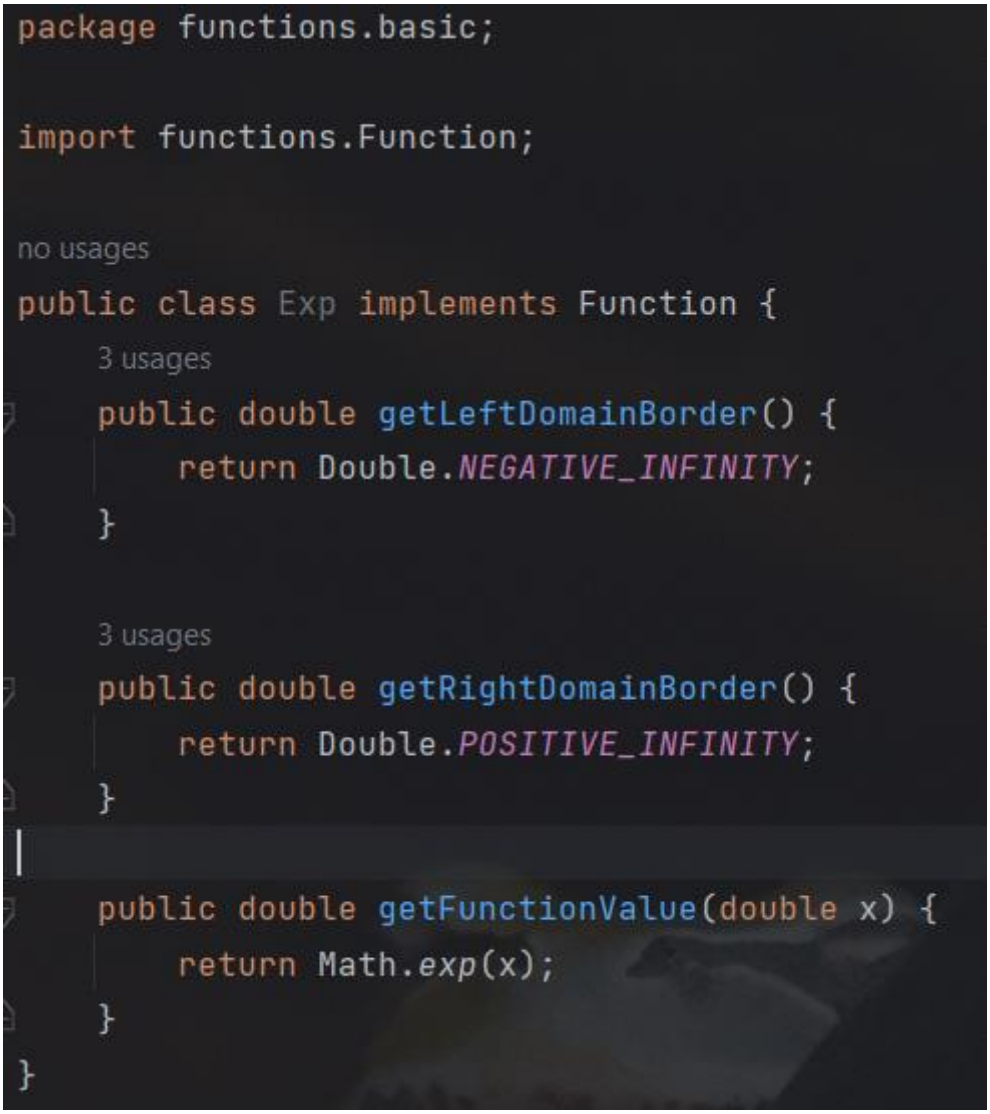


Рис. 5



```
package functions.basic;

import functions.Function;

no usages
public class Exp implements Function {
    3 usages
    public double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }

    3 usages
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }

    public double getFunctionValue(double x) {
        return Math.exp(x);
    }
}
```

Рис. 6

```java
package functions.basic;

import functions.Function;

no usages
public class Log implements Function {
    4 usages
    private double base;
    no usages
    public Log(double base) {this.base = base;}

    //геттер и сеттер для удобства работы с классом
    no usages
    public double getBase() {return this.base;}
    no usages
    public void setBase(double base){ this.base = base;}

    3 usages
    public double getLeftDomainBorder() {return 0;}

    3 usages
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }

    public double getFunctionValue(double x) {
        if (x <= 0) return Double.NaN;
        return Math.log(x) / Math.log(base);
    }
}
```

Рис. 7

```java
package functions.basic;

import functions.Function;

3 usages  3 inheritors
public abstract class TrigonometricFunction implements Function {

    3 usages
    public double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }

    3 usages
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }

    3 implementations
    public abstract double getFunctionValue(double x);
}
```

Рис. 8

```java
package functions.basic;

no usages
public class Sin extends TrigonometricFunction {

    public double getFunctionValue(double x) {
        return Math.sin(x);
    }
}
```

Рис. 9

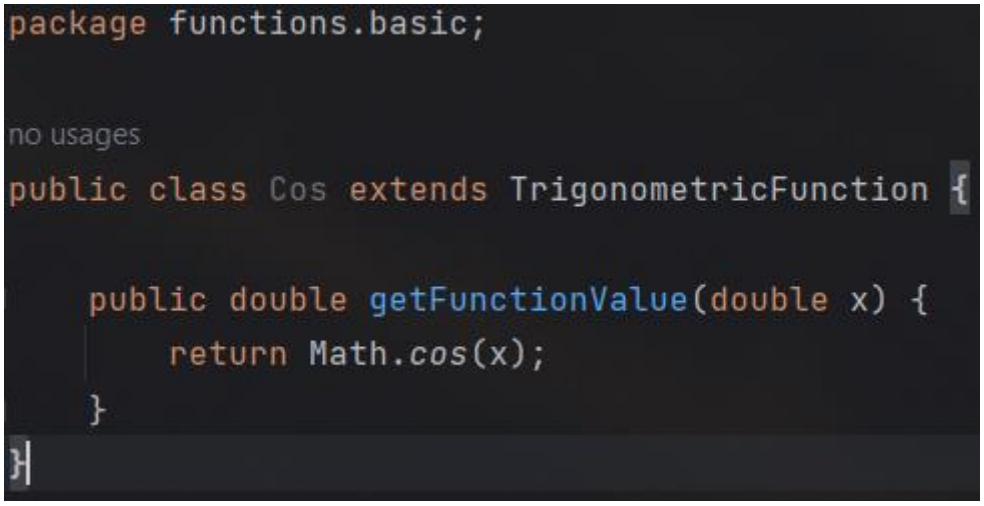

package functions.basic;
no usages
public class Cos extends TrigonometricFunction {
public double getFunctionValue(double x) {
return Math.cos(x);
}
}

Рис. 10

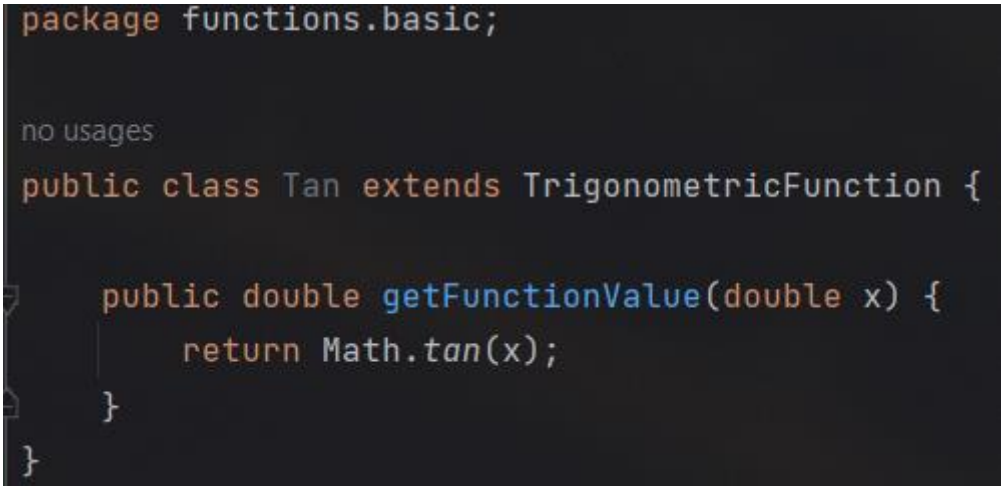

package functions.basic;
no usages
public class Tan extends TrigonometricFunction {
public double getFunctionValue(double x) {
return Math.tan(x);
}
}

Рис. 11

Задание 4

Создадим ещё один пакет, в котором будут классы различных функций – `functions.meta` (Рис. 12). На этот раз в пакете будут описаны классы функций, позволяющие комбинировать функции. Каждый класс должен реализовывать интерфейс `Function`, который был создан в предыдущем задании. Это значит, что все эти операции сами становятся функциями, которые можно дальше комбинировать.

Это задание не вызвало особых сложностей, но были моменты, на которые нужно было обратить особое внимание. Одним из таких было правильное нахождение области определения у функций `Sum` и `Mult`, то есть у тех, где область определения итоговой функции должна быть пересечением двух изначальных областей определения (Рис 13 и рис. 14).

Также интересный момент – возвращение правильного значения левой и правой границы у функции `Scale`. Поскольку в задании прописано, что коэффициенты могут быть и отрицательными, это значит, что при определённных значениях правая и левая граница могут поменяться местами. Поэтому нужно было добавить проверку на отрицательный коэффициент в методе, возвращающем границы области определения (Рис 15).

Остальные же функции, в целом, были немного проще в реализации (Рис 16, 17 и 18).

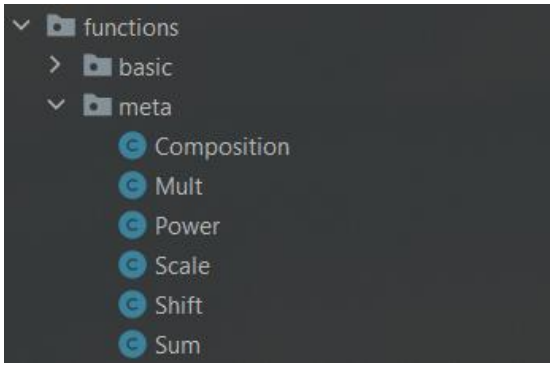


Рис. 12

```java
package functions.meta;

import functions.Function;

public class Mult implements Function {
    private final Function firstFunction;
    private final Function secondFunction;

    public Mult(Function firstFunction, Function secondFunction) {
        if (firstFunction == null || secondFunction == null) throw new IllegalArgumentException();
        if (secondFunction.getRightDomainBorder()<firstFunction.getLeftDomainBorder() || firstFunction.getRightDomainBorder()< secondFunction.getLeftDomainBorder()) throw new IllegalArgumentException("Области опре
        this.firstFunction = firstFunction;
        this.secondFunction = secondFunction;
    }

    public double getLeftDomainBorder() {return Math.max(firstFunction.getLeftDomainBorder(), secondFunction.getLeftDomainBorder());}

    public double getRightDomainBorder() {return Math.min(firstFunction.getRightDomainBorder(), secondFunction.getRightDomainBorder());}

    public double getFunctionValue(double x) {
        //проверяем, что x принадлежит пересечению областей определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {return Double.NaN;}
        return firstFunction.getFunctionValue(x) * secondFunction.getFunctionValue(x);
    }
}
```

Рис. 13

```java
package functions.meta;

import functions.Function;

public class Sum implements Function {
    private final Function firstFunction;
    private final Function secondFunction;

    public Sum(Function firstFunction, Function secondFunction) {
        if (firstFunction == null || secondFunction == null) throw new IllegalArgumentException();
        if (secondFunction.getRightDomainBorder()<firstFunction.getLeftDomainBorder() || firstFunction.getRightDomainBorder()< secondFunction.getLeftDomainBorder()) throw new IllegalArgumentException("Области определе
        this.firstFunction = firstFunction;
        this.secondFunction = secondFunction;
    }

    public double getLeftDomainBorder() {return Math.max(firstFunction.getLeftDomainBorder(), secondFunction.getLeftDomainBorder());}

    public double getRightDomainBorder() {return Math.min(firstFunction.getRightDomainBorder(), secondFunction.getRightDomainBorder());}

    public double getFunctionValue(double x) {
        //проверяем, что x принадлежит пересечению областей определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {return Double.NaN;}
        return firstFunction.getFunctionValue(x) + secondFunction.getFunctionValue(x);
    }
}
```

Рис. 14

```java
import functions.Function;

no usages
public class Scale implements Function {
    6 usages
    private Function function;
    12 usages
    private double scaleX;
    4 usages
    private double scaleY;

    //геттеры и сеттеры для удобства возможной дальнейшей работы с полями класса
    no usages
    public double getScaleX() { return this.scaleX; }
    no usages
    public double getScaleY() { return this.scaleY; }
    no usages
    public void setScaleX(double scaleX) { this.scaleX = scaleX; }
    no usages
    public void setScaleY(double scaleY) { this.scaleY = scaleY; }

    no usages
    public Scale(Function function, double scaleX, double scaleY) {
        if (function == null) throw new IllegalArgumentException();
        this.function = function;
        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }

    public double getLeftDomainBorder() {
        if (this.scaleX < 0) return function.getRightDomainBorder() * scaleX;
        if (this.scaleX >= 0) return function.getLeftDomainBorder() * scaleX;
        return Double.NEGATIVE_INFINITY;
    }

    public double getRightDomainBorder() {
        if (this.scaleX < 0) return function.getLeftDomainBorder() * scaleX;
        if (this.scaleX >= 0) return function.getRightDomainBorder() * scaleX;
        return Double.POSITIVE_INFINITY;
    }

    public double getFunctionValue(double x) {
        //масштабируем аргумент и результат
        return function.getFunctionValue( x: x * scaleX) * scaleY;
    }
}
```

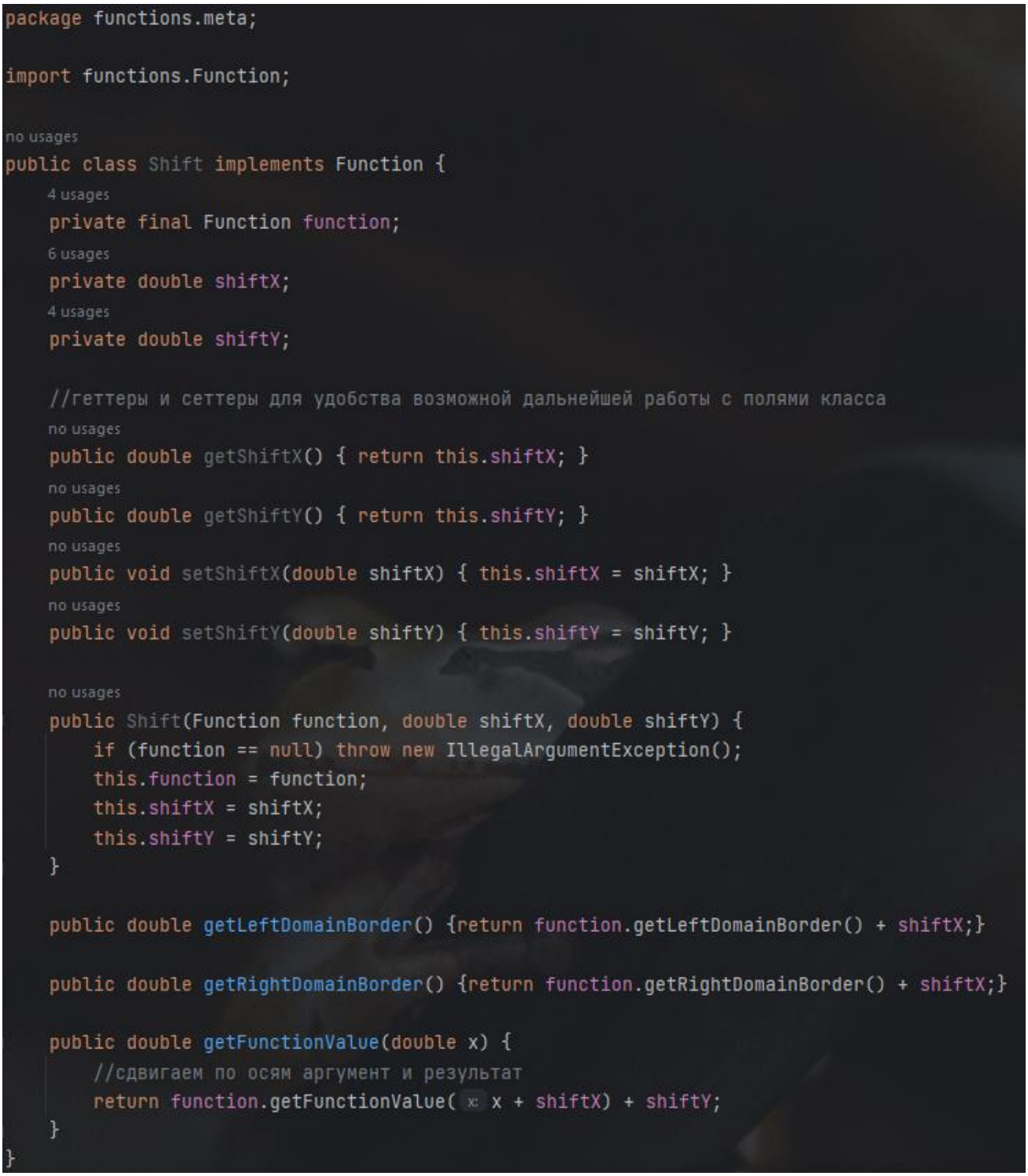
Рис. 15

```java
package functions.meta;

import functions.Function;

no usages
public class Power implements Function {
    4 usages
    private final Function function;
    4 usages
    private double power;

    //геттер и сеттер для удобства работы с классом
    no usages
    public double getPower() {return power;}
    no usages
    public void setPower(double power) {this.power = power;}

    no usages
    public Power (Function function, double power) {
        if (function == null) throw new IllegalArgumentException();
        this.function = function;
        this.power = power;
    }

    public double getLeftDomainBorder() {return function.getLeftDomainBorder();}

    public double getRightDomainBorder() {return function.getRightDomainBorder();}

    public double getFunctionValue(double x) {return Math.pow(function.getFunctionValue(x), power);}
}
```

Рис. 16

```java
package functions.meta;

import functions.Function;

no usages
public class Composition implements Function {
    3 usages
    private final Function outerFunction;
    3 usages
    private final Function innerFunction;

    no usages
    public Composition(Function outerFunction, Function innerFunction) {
        if (outerFunction == null || innerFunction == null) throw new IllegalArgumentException();
        this.outerFunction = outerFunction;
        this.innerFunction = innerFunction;
    }

    public double getLeftDomainBorder() {return innerFunction.getLeftDomainBorder();}

    public double getRightDomainBorder() {return outerFunction.getRightDomainBorder();}

    public double getFunctionValue(double x) {return outerFunction.getFunctionValue(innerFunction.getFunctionValue(x));}
}
```

Рис. 17



```java
package functions.meta;

import functions.Function;

no usages
public class Shift implements Function {
    4 usages
    private final Function function;
    6 usages
    private double shiftX;
    4 usages
    private double shiftY;

    //геттеры и сеттеры для удобства возможной дальнейшей работы с полями класса
    no usages
    public double getShiftX() { return this.shiftX; }
    no usages
    public double getShiftY() { return this.shiftY; }
    no usages
    public void setShiftX(double shiftX) { this.shiftX = shiftX; }
    no usages
    public void setShiftY(double shiftY) { this.shiftY = shiftY; }

    no usages
    public Shift(Function function, double shiftX, double shiftY) {
        if (function == null) throw new IllegalArgumentException();
        this.function = function;
        this.shiftX = shiftX;
        this.shiftY = shiftY;
    }

    public double getLeftDomainBorder() {return function.getLeftDomainBorder() + shiftX;}

    public double getRightDomainBorder() {return function.getRightDomainBorder() + shiftX;}

    public double getFunctionValue(double x) {
        //сдвигаем по осям аргумент и результат
        return function.getFunctionValue( x: x + shiftX) + shiftY;
    }
}
```

Рис. 18

Задание 5

В пакете `functions` создаём класс `Functions`, содержащий вспомогательные статические методы для работы с функциями. По сути этот класс будет являться утилитным.

При выполнении этого задания важным будет отметить то, что мы реализуем приватный конструктор. Это сделано как раз – таки для того, чтобы в программе вне этого класса нельзя было создать его объект (Рис. 19).

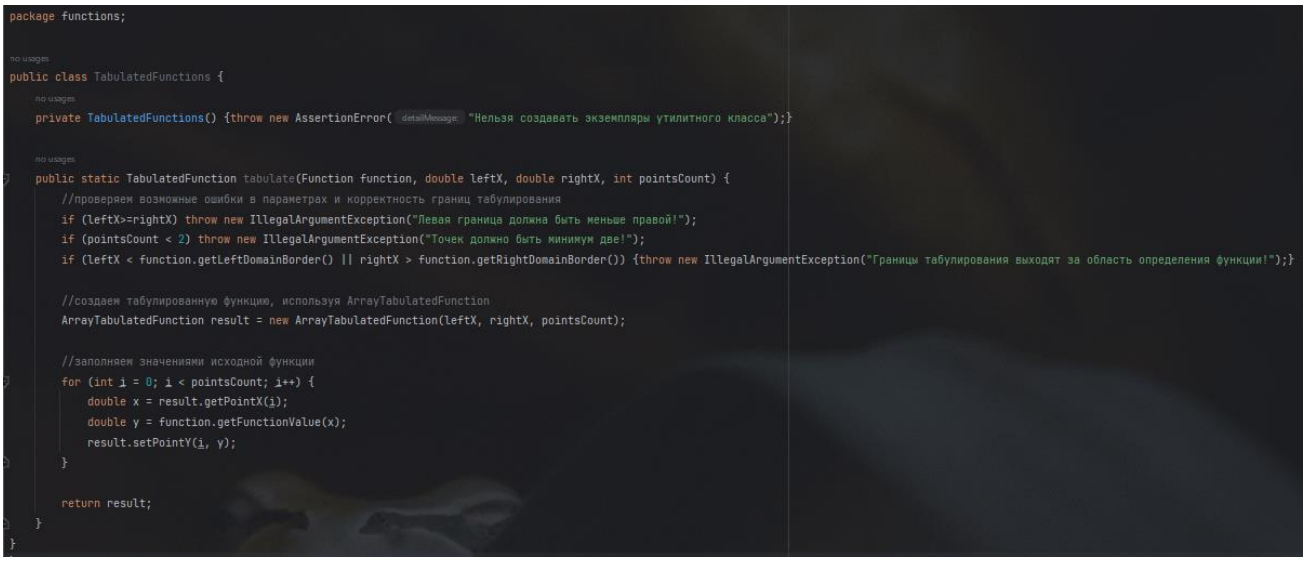
```
package functions;
import functions.meta.*;
no usages
public class Functions {
    no usages
    private Functions() {throw new AssertionError( detailMessage: "Нельзя создавать экземпляры утилитного класса");}
    no usages
    public static Function shift(Function f, double shiftX, double shiftY) {return new Shift(f, shiftX, shiftY);}
    no usages
    public static Function scale(Function f, double scaleX, double scaleY) {return new Scale(f, scaleX, scaleY);}
    no usages
    public static Function power(Function f, double power) {return new Power(f, power);}
    no usages
    public static Function sum(Function f1, Function f2) {return new Sum(f1, f2);}
    no usages
    public static Function mult(Function f1, Function f2) {return new Mult(f1, f2);}
    no usages
    public static Function composition(Function f1, Function f2) {return new Composition(f1, f2);}
}
```

Рис. 19

Задание 6

Всё в том же пакете `functions` создаём ещё один утилитный класс `TabulatedFunctions`. Аналогично делаем приватный конструктор, а также реализуем метод, позволяющий превратить существующую функцию в табулированную (Рис. 20).

Поскольку, по заданию лабораторной, можно возвращать объект любого из классов, реализующих этот интерфейс, я возвращаю объект `ArrayTabulatedFunction`.



```
package functions;

no usages
public class TabulatedFunctions {
    no usages
    private TabulatedFunctions() {throw new AssertionError( detailMessage: "Нельзя создавать экземпляры утилитного класса");}

    no usages
    public static TabulatedFunction tabulate(Function function, double leftX, double rightX, int pointsCount) {
        //проверяем возможные ошибки в параметрах и корректность границ табулирования
        if (leftX>=rightX) throw new IllegalArgumentException("Левая граница должна быть меньше правой!");
        if (pointsCount < 2) throw new IllegalArgumentException("Точек должно быть минимум две!");
        if (leftX < function.getLeftDomainBorder() || rightX > function.getRightDomainBorder()) {throw new IllegalArgumentException("Границы табулирования выходят за область определения функции!");}

        //создаем табулированную функцию, используя ArrayTabulatedFunction
        ArrayTabulatedFunction result = new ArrayTabulatedFunction(leftX, rightX, pointsCount);

        //заполняем значениями исходной функции
        for (int i = 0; i < pointsCount; i++) {
            double x = result.getPointX(i);
            double y = function.getFunctionValue(x);
            result.setPointY(i, y);
        }

        return result;
    }
}
```

Рис. 20

Задание 7

Добавим в TabulatedFunctions методы для сохранения функций в файлы и чтения их из файлов (Рис 21 – рис. 24). Причём работать нужно будет как с байтовым потоком, так и с символьным. Сложности возникли с StreamTokenizer для чтения – пришлось изучать, как он работает. Настраиваем его для чтения чисел с плавающей точкой (Рис 24).

Исключения IOException, на мой взгляд, наверное можно преобразовать в RuntimeException, реализовав это через try – catch и try – with – recources.

К примеру:

```
public static void outputTabulatedFunction(Tabula
tedFunction function, OutputStream out) {
    try (DataOutputStream dos = new DataOutputStr
eam(out)) {
        // Записываем количество точек
        dos.writeInt(function.getPointsCount());

        // Записываем все точки
        for (int i = 0; i < function.getPointsCou
nt(); i++) {
            dos.writeDouble(function.getPointX(i)
);
            dos.writeDouble(function.getPointY(i)
);
        }
    } catch (IOException e) {
```

```
            throw new RuntimeException("Ошибка записи
", e);
        }
}
```

Обоснование: Во – первых, ошибки ввода/вывода обычно фатальны для операции; Во – вторых, такая реализация упрощает код – не нужно везде добавлять throws; А в третьих, при такой реализации сохраняется информация об исходной ошибке.

На счёт закрытия потоков: Байтовые потоки в таком случае будут закрываться автоматически в try – with – recources; Символьные потоки не закрываем, потому что они могут использоваться дальше; PrintWriter не закрываем, потому что он не должен закрывать переданный Writer.

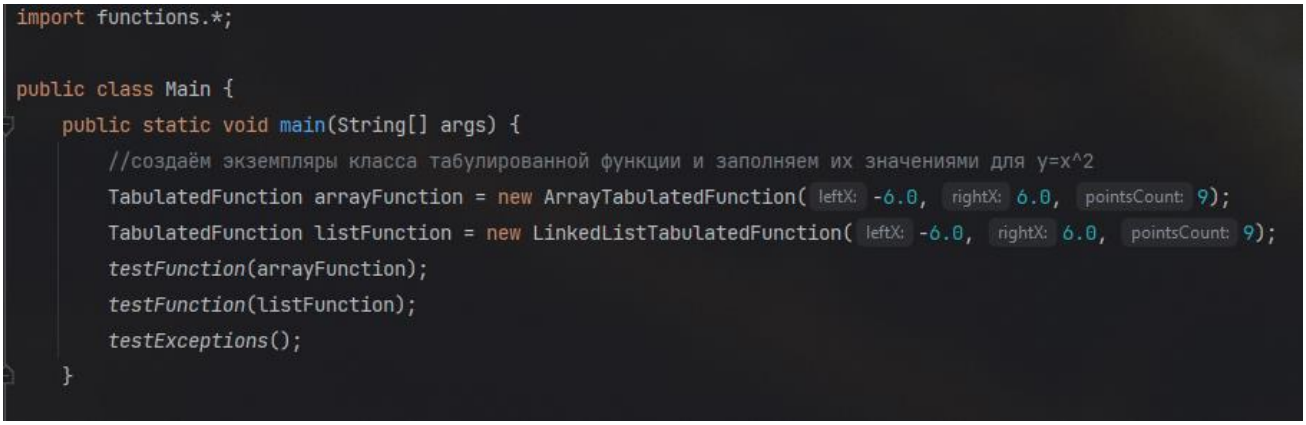


Рис. 22

Задание 8

В данном задании лабораторной требуется проверить работу написанных классов. Не самая сложная, но определённо самая долгая и объёмная часть лабораторной работы.

Начинаем с создания объектов аналитических функций – синуса и косинуса (Рис. 23). Эти функции определены на всей числовой прямой, что подтвердилось при выводе их областей определения. Затем выводим значения этих функций на отрезке от 0 до π с шагом 0.1. Результаты показали, что функции работают корректно – значения синуса начинаются с 0, достигают 1 в середине и возвращаются к 0, в то время как косинус начинается с 1 и уменьшается до -1 (Рис. 24).

С помощью метода `TabulatedFunctions.tabulate()` создаём табулированные версии синуса и косинуса, используя 10 точек на отрезке [0, π] (Рис. 25). При сравнении значений аналитических и табулированных функций видны небольшие расхождения (Рис. 26). Это ожидаемо, поскольку табулированная функция использует линейную интерполяцию между точками, что даёт приближённое значение.

Затем создаём функцию, представляющую сумму квадратов табулированных синуса и косинуса, и также выводим её значения в консоль (Рис. 25 и рис. 27). Результаты показали, что для большинства точек значение действительно близко к 1, что подтверждает корректность работы как

табулированных функций, так и операций над ними. Небольшие отклонения от 1 объясняются погрешностью табулирования и интерполяции.

Далее тестируем работу с файлами. Заодно проверим работу класса экспоненциальной функции, создав её табулированный аналог на отрезке [0, 10] с 11 точками. Затем сохраняем функцию в текстовый файл с помощью `TabulatedFunctions.writeTabulatedFunction()` и загружаем обратно, пользуясь `TabulatedFunctions.readTabulatedFunction()` (Рис. 28). Результаты показали, что после записи и чтения из файла функция полностью сохранила свои значения (Рис. 29).

Аналогичный тест проводим с натуральным логарифмом, но используя бинарный формат через `TabulatedFunctions.outputTabulatedFunction()` и `TabulatedFunctions.inputTabulatedFunction()` (Рис. 30). Результат так же успешный – функция корректно сохранилась и восстановилась (Рис. 31).

Изучив содержимое файлов можно сделать выводы о преимуществах и недостатках каждого из форматов хранения. Текстовый формат оказался удобным для отладки, так как файлы можно просматривать в любом текстовом редакторе. Это позволяет визуально проверить корректность сохранённых данных (Рис. 32). Бинарный формат более эффективен с точки зрения размера файлов и скорости работы, но менее удобен для ручной проверки, однако его недостаток в невозможности прочтения человеком файла (Рис. 33). Оба

формата показали одинаковую надёжность в сохранении и восстановлении данных.

```java
import functions.*;
import functions.basic.*;
import functions.meta.*;
import java.io.*;

public class Main {
    public static void main(String[] args) {
        //создаем объекты функций
        Sin sinFunction = new Sin();
        Cos cosFunction = new Cos();

        System.out.println("Функция Sin определена на отрезке ["+sinFunction.getLeftDomainBorder()+", "+sinFunction.getRightDomainBorder()+"]");
        System.out.println("Функция Cos определена на отрезке ["+cosFunction.getLeftDomainBorder()+", "+cosFunction.getRightDomainBorder()+"]");

        System.out.println();

        //выводим значения с шагом 0.1 на отрезке [0, π]
        System.out.println("Значения функций из 10 точек на отрезке [0, π] с шагом 0.1:");
        System.out.println("x\t\tSin(x)\t\tCos(x)");
        System.out.println("-----------------------------------------");
        for (double x = 0; x <= Math.PI; x += 0.1) {
            double sinValue = sinFunction.getFunctionValue(x);
            double cosValue = cosFunction.getFunctionValue(x);
            System.out.println(x+"\t\t"+sinValue+"\t\t"+cosValue);
        }

        System.out.println();
```

Рис. 23

```
"C:\Program Files\Java\jdk-21.0.6\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1.3\lib\
Функция Sin определена на отрезке [-Infinity, Infinity]
Функция Cos определена на отрезке [-Infinity, Infinity]

Значения функций из 10 точек на отрезке [0, π] с шагом 0.1:
x       Sin(x)      Cos(x)
-----------------------------------------
0.0     0.0     1.0
0.1     0.09983341664682815     0.9950041652780258
0.2     0.19866933079506122     0.9800665778412416
0.30000000000000004     0.2955202066613396      0.955336489125606
0.4     0.3894183423086505      0.9210609940028851
0.5     0.479425538604203       0.8775825618903728
0.6     0.5646424733950354      0.8253356149096783
0.7     0.644217687237691       0.7648421872844885
0.7999999999999999      0.7173560908995227      0.6967067093471655
0.8999999999999999      0.7833269096274833      0.6216099682706645
0.9999999999999999      0.8414709848078964      0.5403023058681398
1.0999999999999999      0.8912073600614353      0.4535961214255775
1.2     0.9320390859672263      0.3623577544766736
1.3     0.963558185417193       0.26749882862458735
1.4000000000000001      0.9854497299884603      0.16996714290024081
1.5000000000000002      0.9974949866040544      0.07073720166770268
1.6000000000000003      0.9995736030415051      -0.029199522301289037
1.7000000000000004      0.9916648104524686      -0.12884449429552508
1.8000000000000005      0.973847630878195       -0.22720209469308753
1.9000000000000006      0.9463000876874142      -0.32328956686350396
2.0000000000000004      0.9092974268256815      -0.4161468365471428
2.1000000000000005      0.8632093666488735      -0.5048461045998579
2.2000000000000006      0.8084964038195899      -0.5885011172553463
2.3000000000000007      0.7457052121767197      -0.6662760212798248
2.400000000000001       0.6754631805511503      -0.737393715541246
2.500000000000001       0.5984721441039558      -0.8011436155469343
2.600000000000001       0.5155013718214634      -0.8568887533689476
2.700000000000001       0.42737988023382895     -0.9040721420170617
2.800000000000001       0.3349881501559038      -0.9422223406686585
2.9000000000000012      0.23924932921398112     -0.9709581651495908
3.0000000000000013      0.1411200080598659      -0.9899924966004456
3.1000000000000014      0.0415806624332891 6    -0.9991351502732795
```

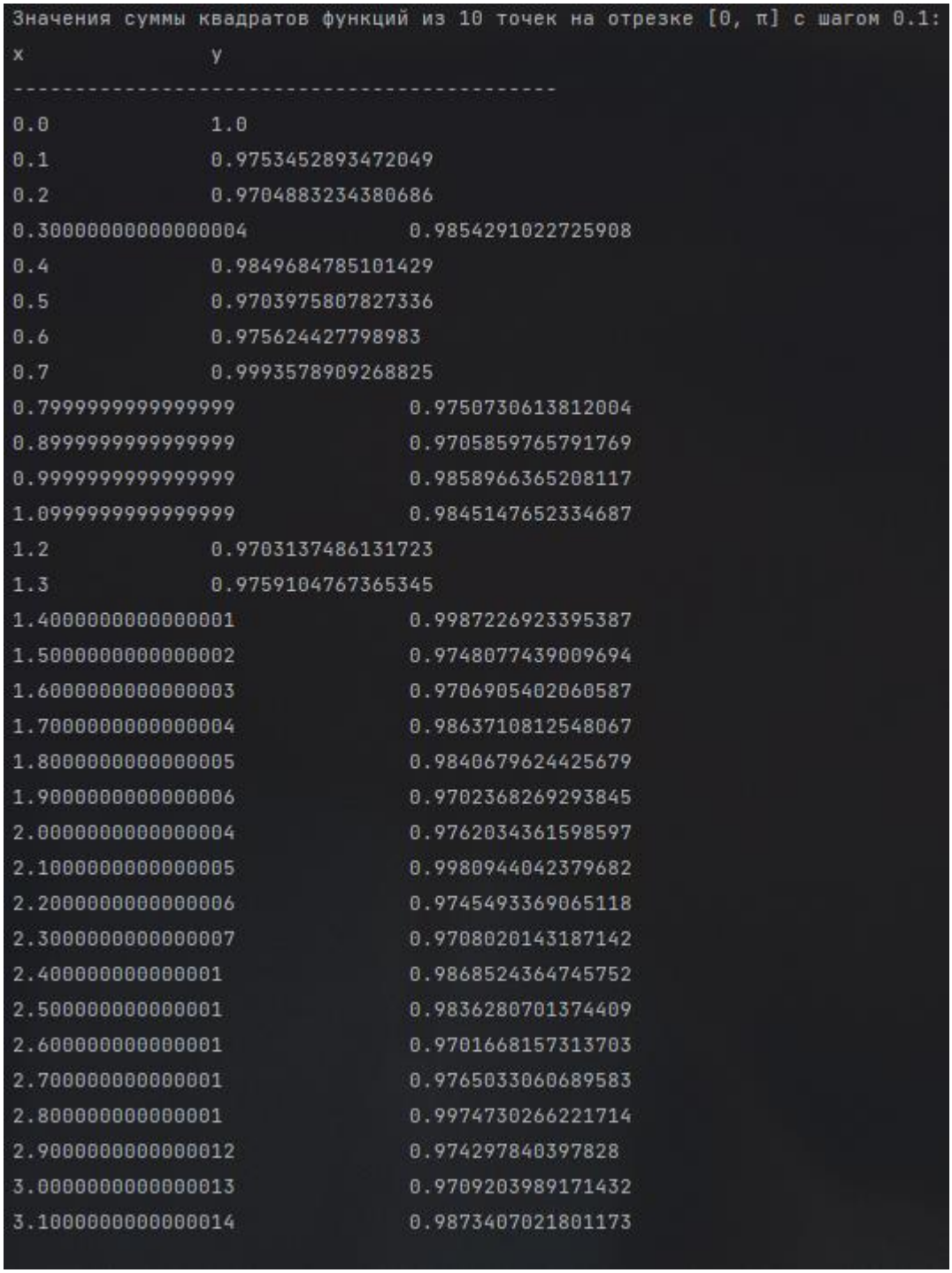
Рис. 24

```
TabulatedFunction sinTabulated = TabulatedFunctions.tabulate(sinFunction, leftX: 0, Math.PI, pointsCount: 10);
TabulatedFunction cosTabulated = TabulatedFunctions.tabulate(cosFunction, leftX: 0, Math.PI, pointsCount: 10);

System.out.println();

//выводим значения с шагом 0.1 на отрезке [0, π]
System.out.println("Значения табулированных функций из 10 точек на отрезке [0, π] с шагом 0.1:");
System.out.println("x\t\tSin(x)\t\tCos(x)");
System.out.println("-----------------------------------------");
for (double x = 0; x <= Math.PI; x += 0.1) {
    double sinValue = sinTabulated.getFunctionValue(x);
    double cosValue = cosTabulated.getFunctionValue(x);
    System.out.println(x+"\t\t"+sinValue+"\t\t"+cosValue);
}

System.out.println();

Sum sumSquares = new Sum(new Power(sinTabulated, power: 2), new Power(cosTabulated, power: 2));
System.out.println("Значения суммы квадратов функций из 10 точек на отрезке [0, π] с шагом 0.1:");
System.out.println("x\t\t\ty");
System.out.println("-----------------------------------------");
for (double x = 0; x <= Math.PI; x += 0.1) {
    System.out.println(x+"\t\t\t"+ sumSquares.getFunctionValue(x));
}

System.out.println();
```

Рис. 25

```
Значения табулированных функций из 10 точек на отрезке [0, π] с шагом 0.1:
x       Sin(x)      Cos(x)
-----------------------------------------
0.0     0.0     1.0
0.1     0.09798155360510165     0.9827232084876878
0.2     0.1959631072102033      0.9654464169753757
0.30000000000000004     0.2939466081530496      0.9481696254630635
0.4     0.38590680571121505     0.914354644443779
0.5     0.47207033789781927     0.864608105941514
0.6     0.5582338700844235      0.8148615674392491
0.7     0.6439824415274444      0.7646204979800333
0.7999999999999999      0.707935358675545       0.6884043792119047
0.8999999999999999      0.7718882758236456      0.6121882604437761
0.9999999999999999      0.8358411929717461      0.5359721416756473
1.0999999999999999      0.8839933571281424      0.45063345391435955
1.2     0.9180219935851436      0.35714054363391856
1.3     0.9520506300421447      0.26364763353347763
1.4000000000000001      0.984807753012208       0.16993052093895483
1.5000000000000002      0.984807753012208       0.07043744393442489
1.6000000000000003      0.984807753012208       -0.029055633070105086
1.7000000000000004      0.984807753012208       -0.12854871007463503
1.8000000000000005      0.966204042925035       -0.22476145104951817
1.9000000000000006      0.932175406468034       -0.3182543613299591
2.0000000000000004      0.8981467700110329      -0.41174727161039987
2.1000000000000005      0.8624409082617227      -0.5042718354168345
2.2000000000000006      0.7984879911136221      -0.5804879541849632
2.3000000000000007      0.7345350739655215      -0.656704072953092
2.400000000000001       0.670582156817421       -0.7329201917212208
2.500000000000001       0.594071569547527       -0.7941706620070894
2.600000000000001       0.5079080373609227      -0.8439172005093544
2.700000000000001       0.42174450517431844     -0.8936637390116193
2.800000000000001       0.3346977889881713      -0.9409837494179168
2.9000000000000012      0.23671623538306957     -0.958260540930229
3.0000000000000013      0.1387346817779678      -0.9755373324425413
3.1000000000000014      0.04075312817286608     -0.9928141239548535
```

Рис. 26



```
Значения суммы квадратов функций из 10 точек на отрезке [0, π] с шагом 0.1:
x                  y
-------------------------------------------
0.0                1.0
0.1                0.9753452893472049
0.2                0.9704883234380686
0.30000000000000004                0.9854291022725908
0.4                0.9849684785101429
0.5                0.9703975807827336
0.6                0.975624427798983
0.7                0.9993578909268825
0.7999999999999999                 0.9750730613812004
0.8999999999999999                 0.9705859765791769
0.9999999999999999                 0.9858966365208117
1.0999999999999999                 0.9845147652334687
1.2                0.9703137486131723
1.3                0.9759104767365345
1.4000000000000001                 0.9987226923395387
1.5000000000000002                 0.9748077439009694
1.6000000000000003                 0.9706905402060587
1.7000000000000004                 0.9863710812548067
1.8000000000000005                 0.9840679624425679
1.9000000000000006                 0.9702368269293845
2.0000000000000004                 0.9762034361598597
2.1000000000000005                 0.9980944042379682
2.2000000000000006                 0.9745493369065118
2.3000000000000007                 0.9708020143187142
2.400000000000001                  0.9868524364745752
2.500000000000001                  0.9836280701374409
2.600000000000001                  0.9701668157313703
2.700000000000001                  0.9765033060689583
2.800000000000001                  0.9974730266221714
2.9000000000000012                 0.974297840397828
3.0000000000000013                 0.9709203989171432
3.1000000000000014                 0.9873407021801173
```

Рис. 27

```java
Exp exp = new Exp();
TabulatedFunction expTabulated = TabulatedFunctions.tabulate(exp, leftX: 0, rightX: 10, pointsCount: 11);
System.out.println("Значения табулированной функции экспоненты:");
printTabulatedFunction(expTabulated);

System.out.println();

try {
    File textFile = new File( pathname: "exp.txt");
    FileWriter fw = new FileWriter(textFile);
    TabulatedFunctions.writeTabulatedFunction(expTabulated, fw);
    fw.close();
    FileReader fr = new FileReader(textFile);
    TabulatedFunction newExpTabulated = TabulatedFunctions.readTabulatedFunction(fr);
    System.out.println("Значения табулированной функции экспоненты после записи и чтения из файла:");
    printTabulatedFunction(newExpTabulated);
} catch (IOException ioe) {System.out.println("Ошибка: " + ioe);}

System.out.println();

Log log = new Log(Math.E);
TabulatedFunction logTabulated = TabulatedFunctions.tabulate(log, leftX: 0, rightX: 10, pointsCount: 11);
System.out.println("Значения табулированной функции логарифма:");
printTabulatedFunction(logTabulated);

System.out.println();
```

Рис. 28

```
Значения табулированной функции экспоненты:
x               y
-----------------------------------------
0.0             1.0
1.0             2.718281828459045
2.0             7.38905609893065
3.0             20.085536923187668
4.0             54.598150033144236
5.0             148.4131591025766
6.0             403.4287934927351
7.0             1096.6331584284585
8.0             2980.9579870417283
9.0             8103.083927575384
10.0                22026.465794806718

Значения табулированной функции экспоненты после записи и чтения из файла:
x               y
-----------------------------------------
0.0             1.0
1.0             2.718281828459045
2.0             7.38905609893065
3.0             20.085536923187668
4.0             54.598150033144236
5.0             148.4131591025766
6.0             403.4287934927351
7.0             1096.6331584284585
8.0             2980.9579870417283
9.0             8103.083927575384
10.0                22026.465794806718
```

Рис. 29

```
    Log log = new Log(Math.E);
    TabulatedFunction logTabulated = TabulatedFunctions.tabulate(log, leftX: 0, rightX: 10, pointsCount: 11);
    System.out.println("Значения табулированной функции логарифма:");
    printTabulatedFunction(logTabulated);

    System.out.println();

    try {
        File textFile = new File( pathname: "log.txt");
        FileOutputStream fos = new FileOutputStream(textFile);
        TabulatedFunctions.outputTabulatedFunction(logTabulated, fos);
        fos.close();
        FileInputStream fis = new FileInputStream(textFile);
        TabulatedFunction newLogTabulated = TabulatedFunctions.inputTabulatedFunction(fis);
        System.out.println("Значения табулированной функции логарифма после записи и чтения из файла:");
        printTabulatedFunction(newLogTabulated);
    } catch (IOException ioe) {System.out.println("Ошибка: " + ioe);}
}
4 usages
private static void printTabulatedFunction(TabulatedFunction function) {
    System.out.println("x\t\t\t\ty");
    System.out.println("-----------------------------------------");
    for (int i = 0; i < function.getPointsCount(); i++) {
        System.out.println(function.getPointX(i)+"\t\t\t\t"+function.getPointY(i));
    }
}
```

Рис. 30

```
Значения табулированной функции логарифма:
x               y
-----------------------------------------
0.0             NaN
1.0             0.0
2.0             0.6931471805599453
3.0             1.0986122886681098
4.0             1.3862943611198906
5.0             1.6094379124341003
6.0             1.791759469228055
7.0             1.9459101490553132
8.0             2.0794415416798357
9.0             2.1972245773362196
10.0                2.302585092994046

Значения табулированной функции логарифма после записи и чтения из файла:
x               y
-----------------------------------------
0.0             NaN
1.0             0.0
2.0             0.6931471805599453
3.0             1.0986122886681098
4.0             1.3862943611198906
5.0             1.6094379124341003
6.0             1.791759469228055
7.0             1.9459101490553132
8.0             2.0794415416798357
9.0             2.1972245773362196
10.0                2.302585092994046

Process finished with exit code 0
```

Рис. 31

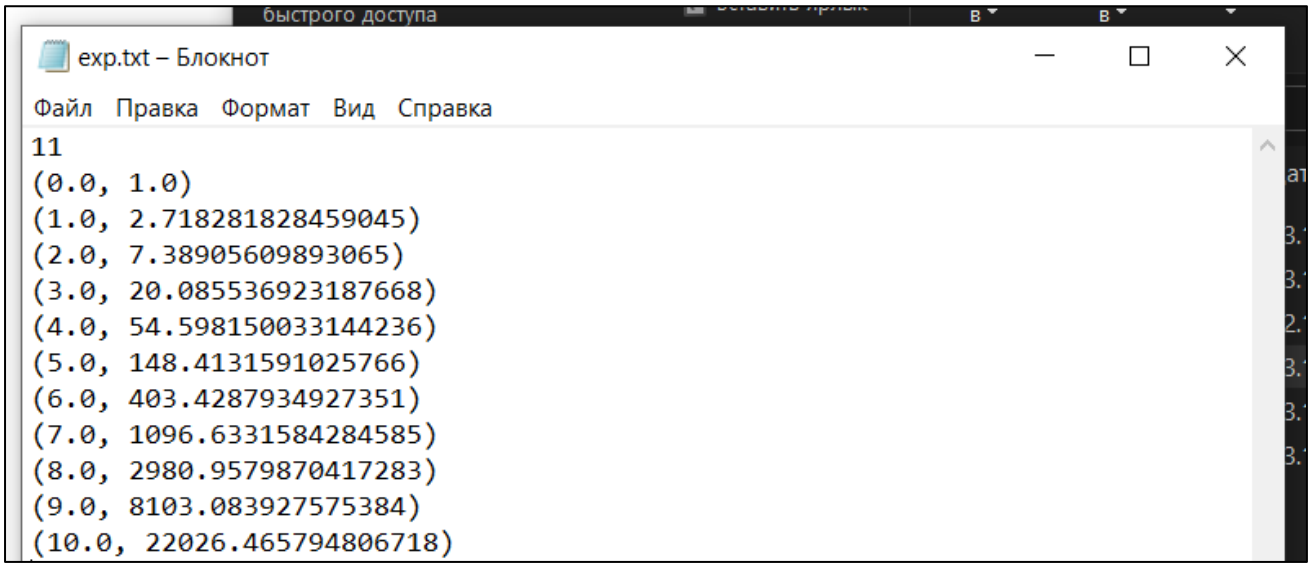

быстрого доступа
exp.txt – Блокнот
Файл Правка Формат Вид Справка
11
(0.0, 1.0)
(1.0, 2.718281828459045)
(2.0, 7.38905609893065)
(3.0, 20.085536923187668)
(4.0, 54.598150033144236)
(5.0, 148.4131591025766)
(6.0, 403.4287934927351)
(7.0, 1096.6331584284585)
(8.0, 2980.9579870417283)
(9.0, 8103.083927575384)
(10.0, 22026.465794806718)

Рис. 32

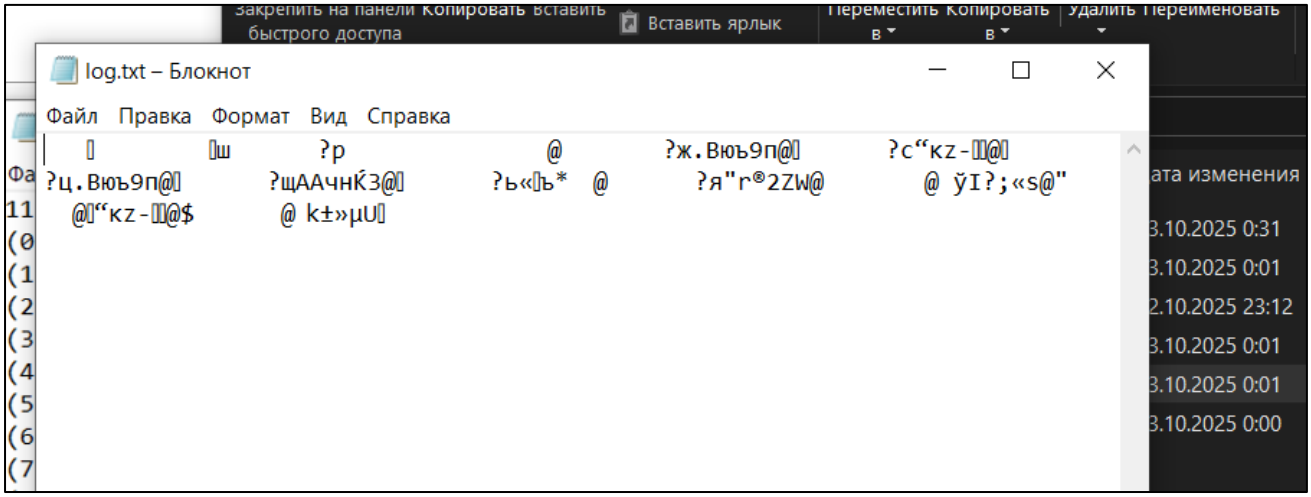

Закрепить на панели Копировать Вставить
быстрого доступа
Вставить ярлык
Переместить Копировать
Удалить Переименовать
log.txt – Блокнот
Файл Правка Формат Вид Справка
?p @ ?ж.Вюъ9п@ ?с“кz-@
?ц.Вюъ9п@ ?щААчнЌЗ@ ?ь«ъ* @ ?я"r®2ZW@ @ ўI?;«s@"
@“кz-@$ @ k±»µU
ата изменения
3.10.2025 0:31
3.10.2025 0:01
2.10.2025 23:12
3.10.2025 0:01
3.10.2025 0:01
3.10.2025 0:00

Рис. 33

Задание 9

В этой части лабораторной работы нужно было добавить поддержку сериализации для классов табулированных функций. Сериализация позволяет сохранять объекты в файлы и затем восстанавливать их из этих файлов. Рассмотрим два подхода: автоматическую сериализацию через `Serializable` и ручную через `Externalizable`.

Для начала реализуем сериализацию с использованием `Serializable`. Для этого нужно добавить `import java.io.Serializable` и прописать `Serializable` в нужных нам классах (Рис. 34 – 40). Это очень удобно – не нужно ничего писать, всё работает автоматически. Но за эту простоту приходится платить: файлы получаются довольно большими, потому что Java сохраняет не только сами данные, но и много служебной информации о классе, его структуре и так далее.

`Externalizable` требует больше работы – нужно реализовать два метода: `writeExternal` для сохранения и `readExternal` для загрузки. Здесь обязательно нужно прописать, какие именно данные сохранять и в каком порядке. Также для работы обязательно нужны конструкторы без параметров, без них просто ничего не получится (для десериализации). Зато я получаю полный контроль над процессом. В результате файлы получаются более компактными, потому что сохраняются только сами координаты точек, без всякой лишней информации (Рис. 41 – 49).

Что касается скорости, то здесь тоже видна разница. `Serializable` работает медленнее, потому что ему нужно анализировать структуру объекта, проверять типы данных, сохранять метаинформацию. `Externalizable` же работает практически на максимальной скорости, так как просто записывает готовые числа в файл в заранее определённом порядке.

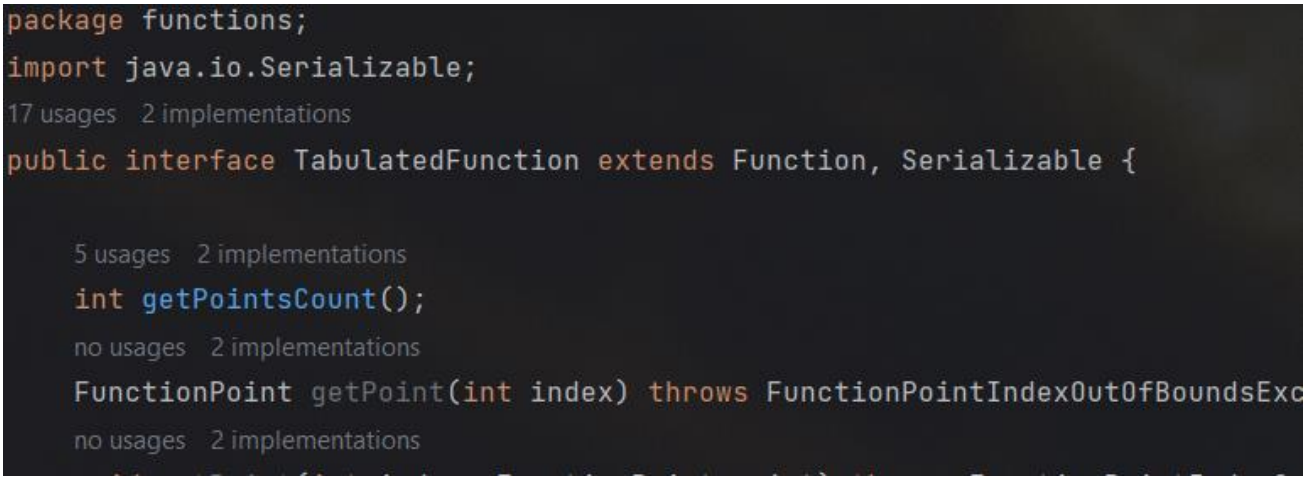


Рис. 34

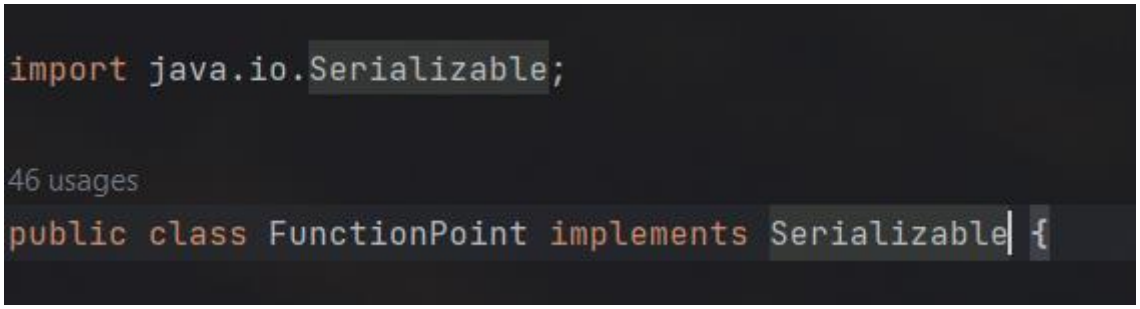


Рис. 35

Рис. 36

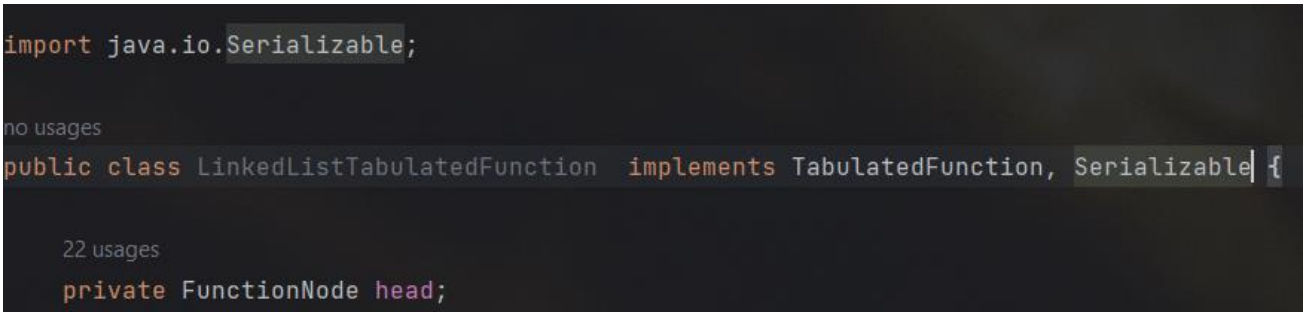


Рис. 37

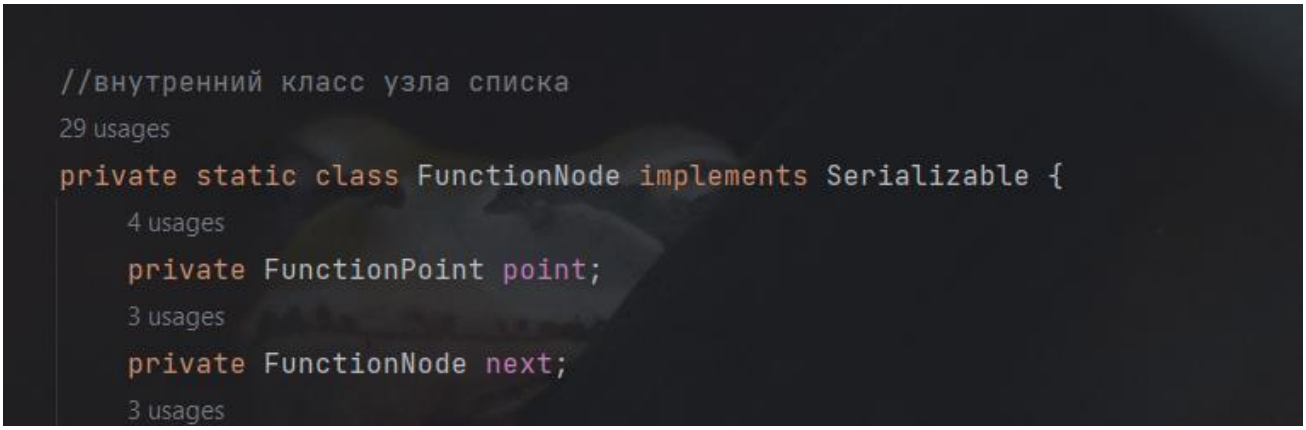


Рис. 38

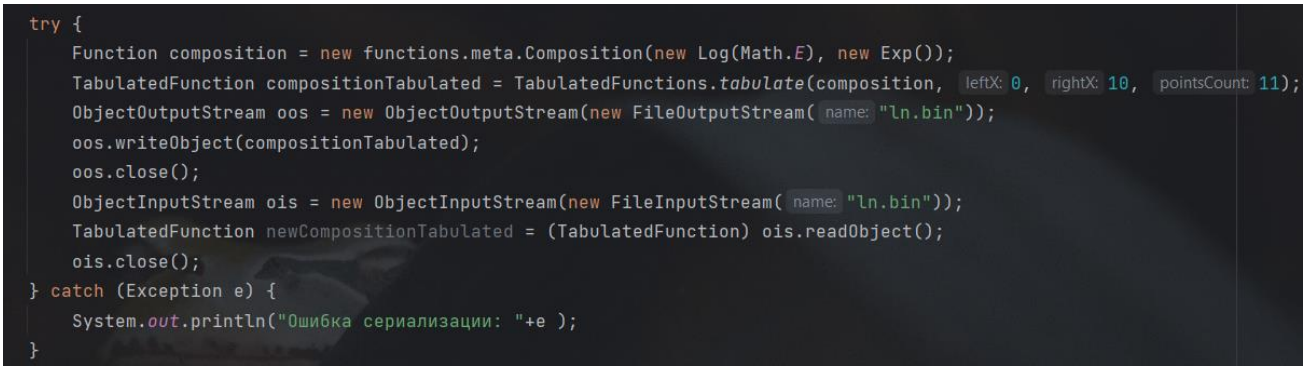


Рис. 39

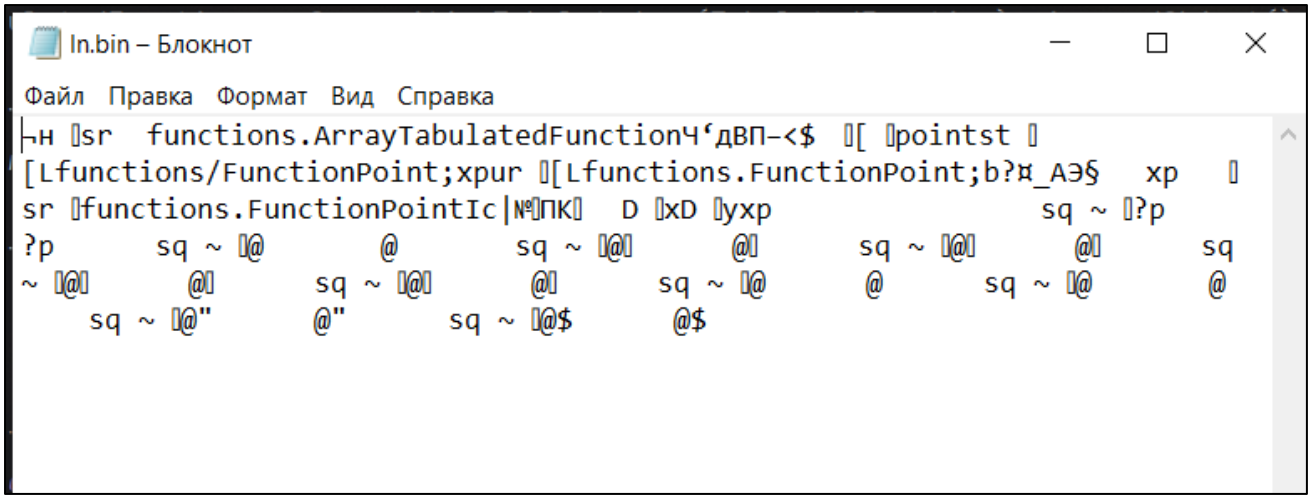

In.bin – Блокнот
Файл Правка Формат Вид Справка
functions.ArrayTabulatedFunction
[Lfunctions/FunctionPoint;xpur
[Lfunctions.FunctionPoint;
functions.FunctionPoint

Рис. 40

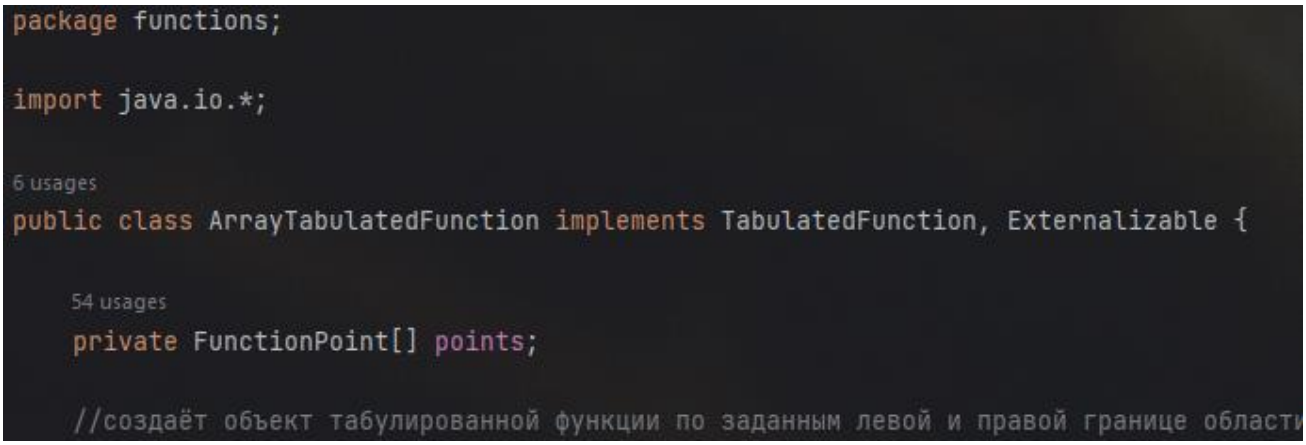
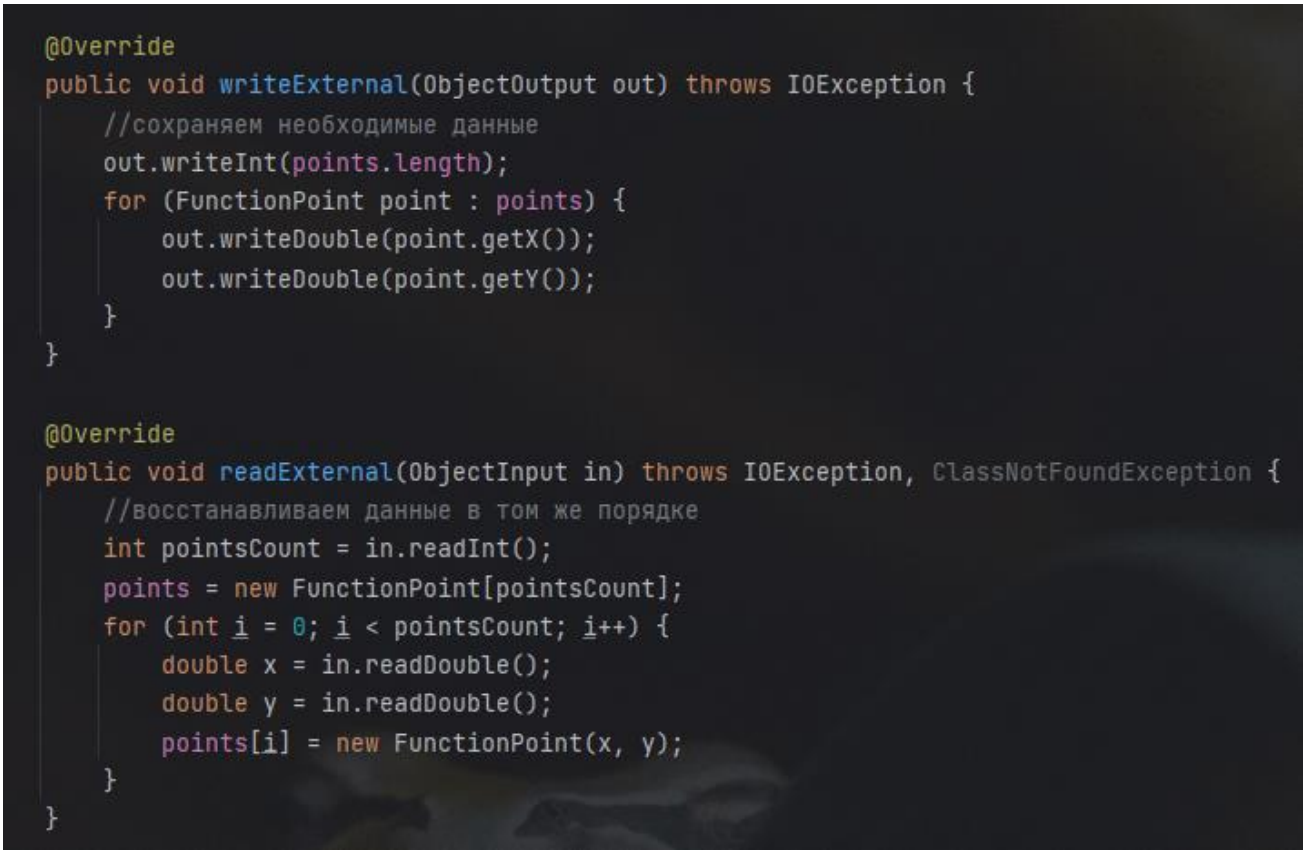
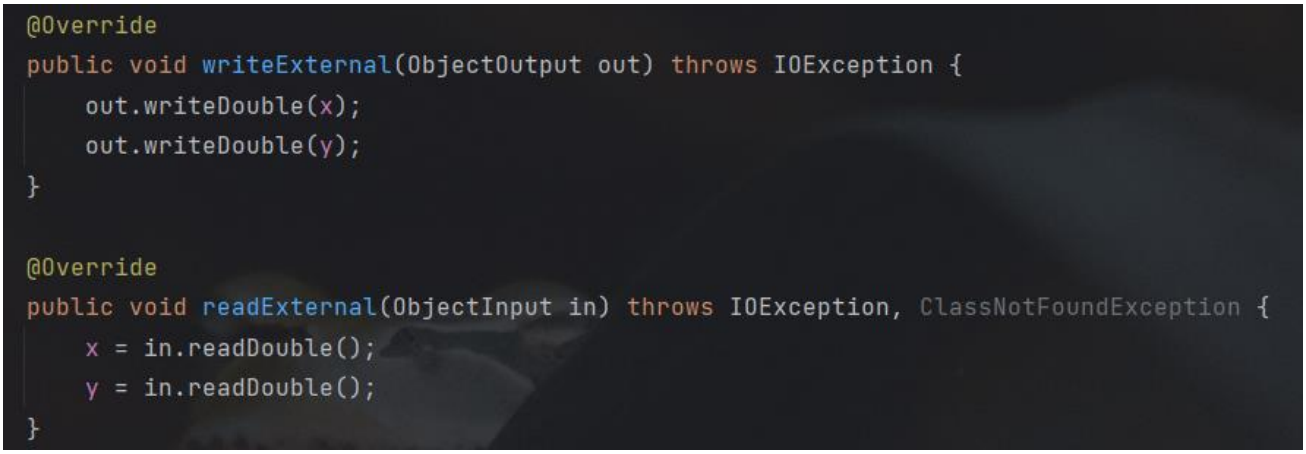

package functions;
import java.io.*;
6 usages
public class ArrayTabulatedFunction implements TabulatedFunction, Externalizable {
54 usages
private FunctionPoint[] points;
//создаёт объект табулированной функции по заданным левой и правой границе области

Рис. 41



```java
@Override
public void writeExternal(ObjectOutput out) throws IOException {
    //сохраняем необходимые данные
    out.writeInt(points.length);
    for (FunctionPoint point : points) {
        out.writeDouble(point.getX());
        out.writeDouble(point.getY());
    }
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    //восстанавливаем данные в том же порядке
    int pointsCount = in.readInt();
    points = new FunctionPoint[pointsCount];
    for (int i = 0; i < pointsCount; i++) {
        double x = in.readDouble();
        double y = in.readDouble();
        points[i] = new FunctionPoint(x, y);
    }
}
```

Рис. 42



```java
@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeDouble(x);
    out.writeDouble(y);
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    x = in.readDouble();
    y = in.readDouble();
}
```

Рис. 43

```java
@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(point);
    // Не сериализуем ссылки next и prev - они будут восстановлены при восстановлении списка
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    point = (FunctionPoint) in.readObject();
    // Ссылки next и prev будут установлены при добавлении в список
}
```

Рис. 44

```java
6 usages
public FunctionNode getPrev() {return prev;}
7 usages
public void setPrev(FunctionNode prev) {this.prev = prev;}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(point);
    //не сериализуем ссылки next и prev - они будут восстановлены при восстановлении списка
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    point = (FunctionPoint) in.readObject();
    //ссылки next и prev будут установлены при добавлении в список
}
}
```

Рис. 45

```java
@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(pointsCount);

    // Сохраняем все точки
    FunctionNode current = head.getNext();
    while (current != head) {
        out.writeDouble(current.getPoint().getX());
        out.writeDouble(current.getPoint().getY());
        current = current.getNext();
    }
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    int count = in.readInt();

    // Инициализируем список
    initializeList();

    // Восстанавливаем точки
    for (int i = 0; i < count; i++) {
        double x = in.readDouble();
        double y = in.readDouble();
        addNodeToTail().setPoint(new FunctionPoint(x, y));
    }

    // Инициализируем кэш
    lastAccessedNode = head.getNext();
    lastAccessedIndex = 0;
}
```

Рис. 46

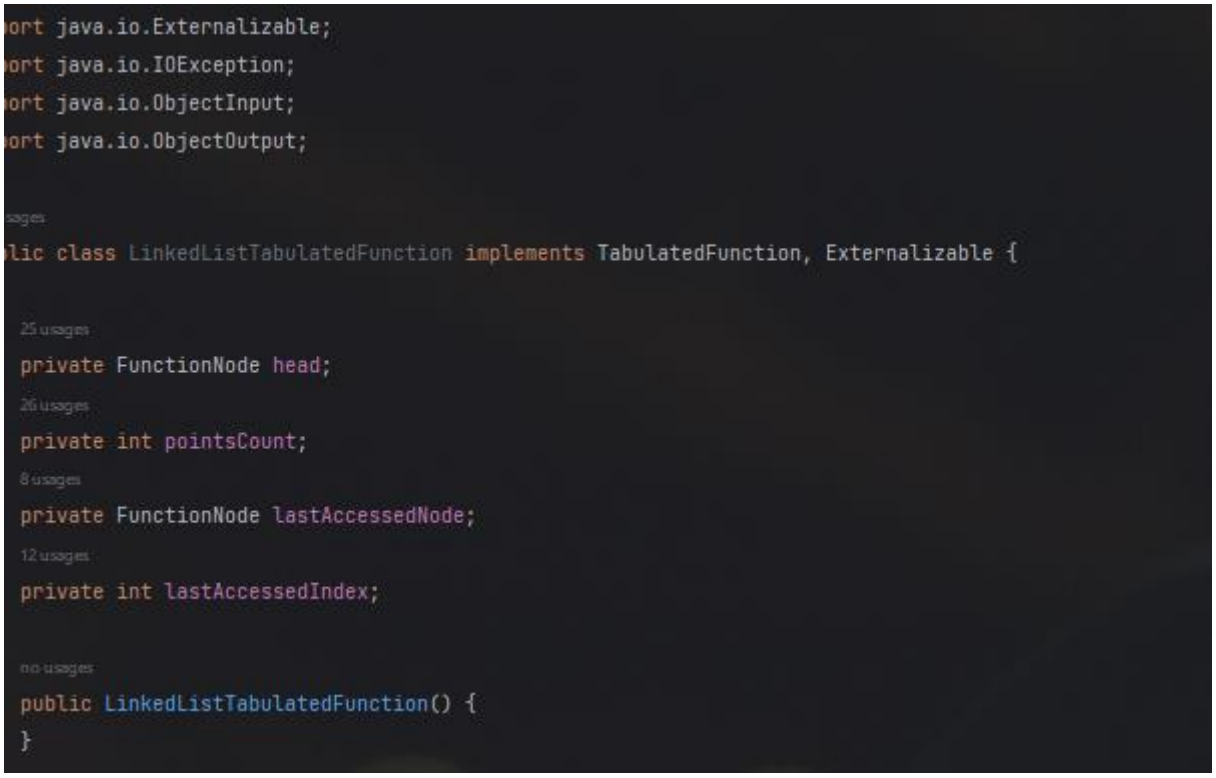

ort java.io.Externalizable;
ort java.io.IOException;
ort java.io.ObjectInput;
ort java.io.ObjectOutput;
sages
lic class LinkedListTabulatedFunction implements TabulatedFunction, Externalizable {
25 usages
private FunctionNode head;
26 usages
private int pointsCount;
8 usages
private FunctionNode lastAccessedNode;
12 usages
private int lastAccessedIndex;
no usages
public LinkedListTabulatedFunction() {
}

Рис. 47

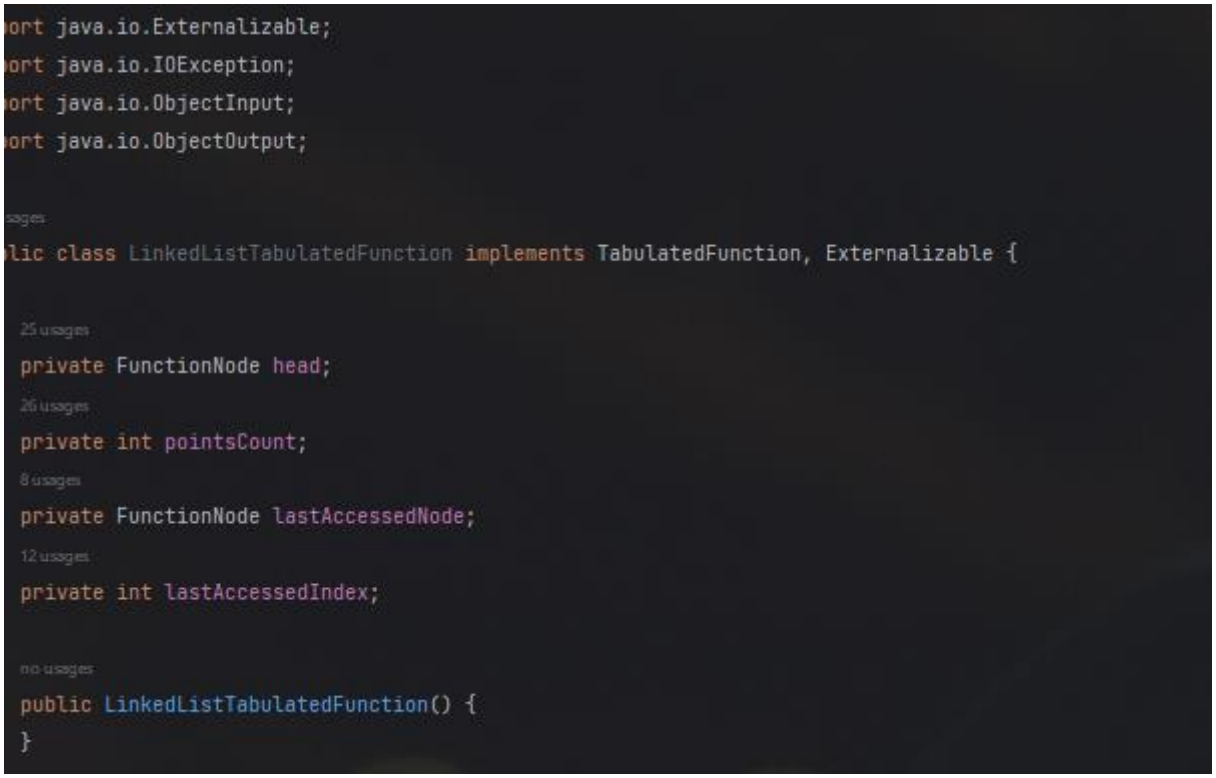

ort java.io.Externalizable;
ort java.io.IOException;
ort java.io.ObjectInput;
ort java.io.ObjectOutput;
sages
lic class LinkedListTabulatedFunction implements TabulatedFunction, Externalizable {
25 usages
private FunctionNode head;
26 usages
private int pointsCount;
8 usages
private FunctionNode lastAccessedNode;
12 usages
private int lastAccessedIndex;
no usages
public LinkedListTabulatedFunction() {
}

Рис. 48

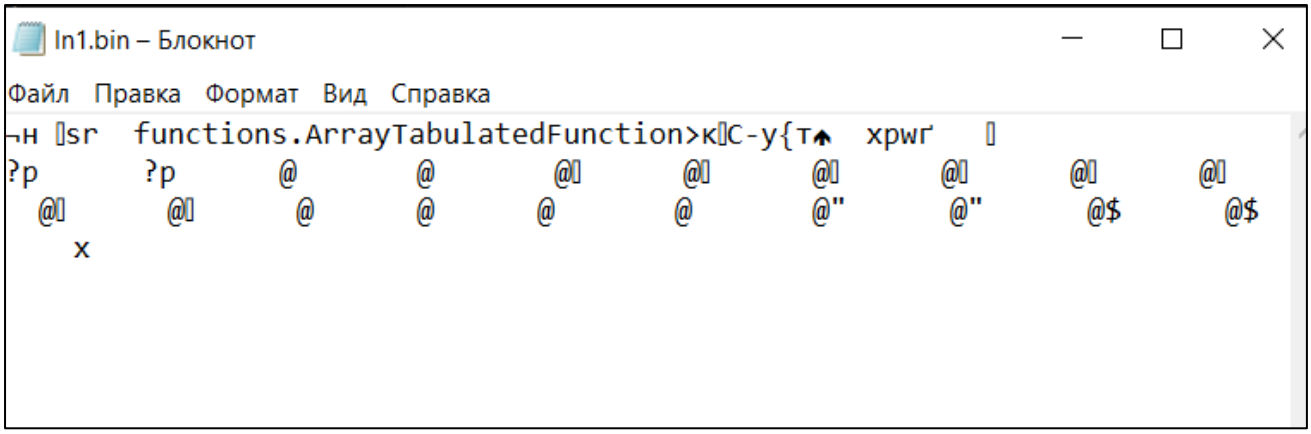

In1.bin – Блокнот
Файл Правка Формат Вид Справка
¬н sr functions.ArrayTabulatedFunction>к C-y{т хрwґ
?p ?p @ @ @ @ @ @ @ @
@ @ @ @ @ @ @" @" @$ @$
x

Рис. 49