Задание 2

В пакете functions создал интерфейс Function, описывающий функции одной переменной и содержащий следующие методы:

• public double getLeftDomainBorder() – возвращает значение левой границы области определения функции;

• public double getRightDomainBorder() – возвращает значение правой границы области определения функции;

• public double getFunctionValue(double x) – возвращает значение функции в заданной точке.

```
package functions;

You, 6 дней назад | 1 author (You)
/**
 * Интерфейс, описывающий функции одной переменной
 */
You, 6 дней назад | 1 author (You)
public interface Function {

    double getLeftDomainBorder();

    double getRightDomainBorder();

    double getFunctionValue(double x);
}
```

Исключил соответствующие методы из интерфейса TabulatedFunction и сделал так, чтобы он расширял интерфейс Function. Теперь табулированные функции буду частным случаем функций одной переменной.

```
package functions;

You, 6 дней назад | 1 author (You)
public interface TabulatedFunction extends Function {

    int getPointsCount();

    // Методы работы с точками
    FunctionPoint getPoint(int index);
    void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException;

    // Методы работы с координатами точек
    double getPointX(int index);
    void setPointX(int index, double x) throws InappropriateFunctionPointException;
    double getPointY(int index);
    void setPointY(int index, double y);

    // Методы модификации набора точек
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;
    void deletePoint(int index);
}       You, 6 дней назад • Uncommitted changes
```

Задание 3

Создайте пакет functions.basic, в нём описаны классы ряда функций, заданных аналитически.

Создайте в пакете публичный класс Exp, объекты которого должны вычислять значение экспоненты. Класс должен реализовывать интерфейс Function. Для вычисления экспоненты следует воспользоваться методом Math.exp(), а для возвращения значений границ области определения – константами из класса Double.

```java
package functions.basic;

import functions.Function;
You, 6 дней назад | 1 author (You)
/**
 * Класс, представляющий экспоненциальную функцию e^x
 */
You, 6 дней назад | 1 author (You)
public class Exp implements Function {

    /**
     * Возвращает значение левой границы области определения экспоненты
     * @return -∞ (Double.NEGATIVE_INFINITY)
     */
    @Override
    public double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }
    /**
     * Возвращает значение правой границы области определения экспоненты
     * @return +∞ (Double.POSITIVE_INFINITY)
     */
    @Override
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }
    /**
     * Вычисляет значение экспоненты в заданной точке
     * @param x - аргумент функции
     * @return e^x
     */
    @Override
    public double getFunctionValue(double x) {
        return Math.exp(x);
    }
```

Аналогично, создайте класс Log, объекты которого должны вычислять значение логарифма по заданному основанию. Основание должно передаваться как параметр конструктора. Для вычисления логарифма следует воспользоваться методом Math.log().

```java
package functions.basic;

import functions.Function;

/**
 * Класс, представляющий логарифмическую функцию log_base(x)
 */
public class Log implements Function {
    private final double base;
    /**
     * Конструктор логарифма с заданным основанием
     * @param base основание логарифма (должно быть положительным и не равным 1)
     * @throws IllegalArgumentException если основание некорректно
     */
    public Log(double base) {
        if (base <= 0 || Math.abs(base - 1.0) < 1e-10) {
            throw new IllegalArgumentException("Основание логарифма должно быть положительным и не равным 1");
        }
        this.base = base;
    }
    /**
     * Возвращает значение левой границы области определения логарифма
     * @return 0 (x > 0)
     */
    @Override
    public double getLeftDomainBorder() {
        return 0.0; // Логарифм определен для x > 0
    }
    /**
     * Возвращает значение правой границы области определения логарифма
     * @return +∞ (Double.POSITIVE_INFINITY)
     */
    @Override
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }
    /**
     * Вычисляет значение логарифма в заданной точке
     * @param x - аргумент функции (должен быть положительным)
     * @return log_base(x), или Double.NaN если x <= 0
     */
    @Override
    public double getFunctionValue(double x) {
        if (x <= 0) {
            return Double.NaN; // Логарифм не определен для неположительных аргументов
        }
        return Math.log(x) / Math.log(base);
    }
    /**
     * Возвращает основание логарифма
     * @return основание логарифма
     */
    public double getBase() {
        return base;
    }
    /**
     * Возвращает строковое представление функции
     * @return "log_base(x)"
     */
```

Прежде, чем перейти к описанию классов для тригонометрических функций (синуса, косинуса и тангенса), обратите внимание на то, что область определения этих функций совпадает, поэтому описывать одинаковые методы в этих классах будет достаточно странным. Проще будет описать базовый класс с реализацией этих методов, а классы конкретных функций наследовать от него.

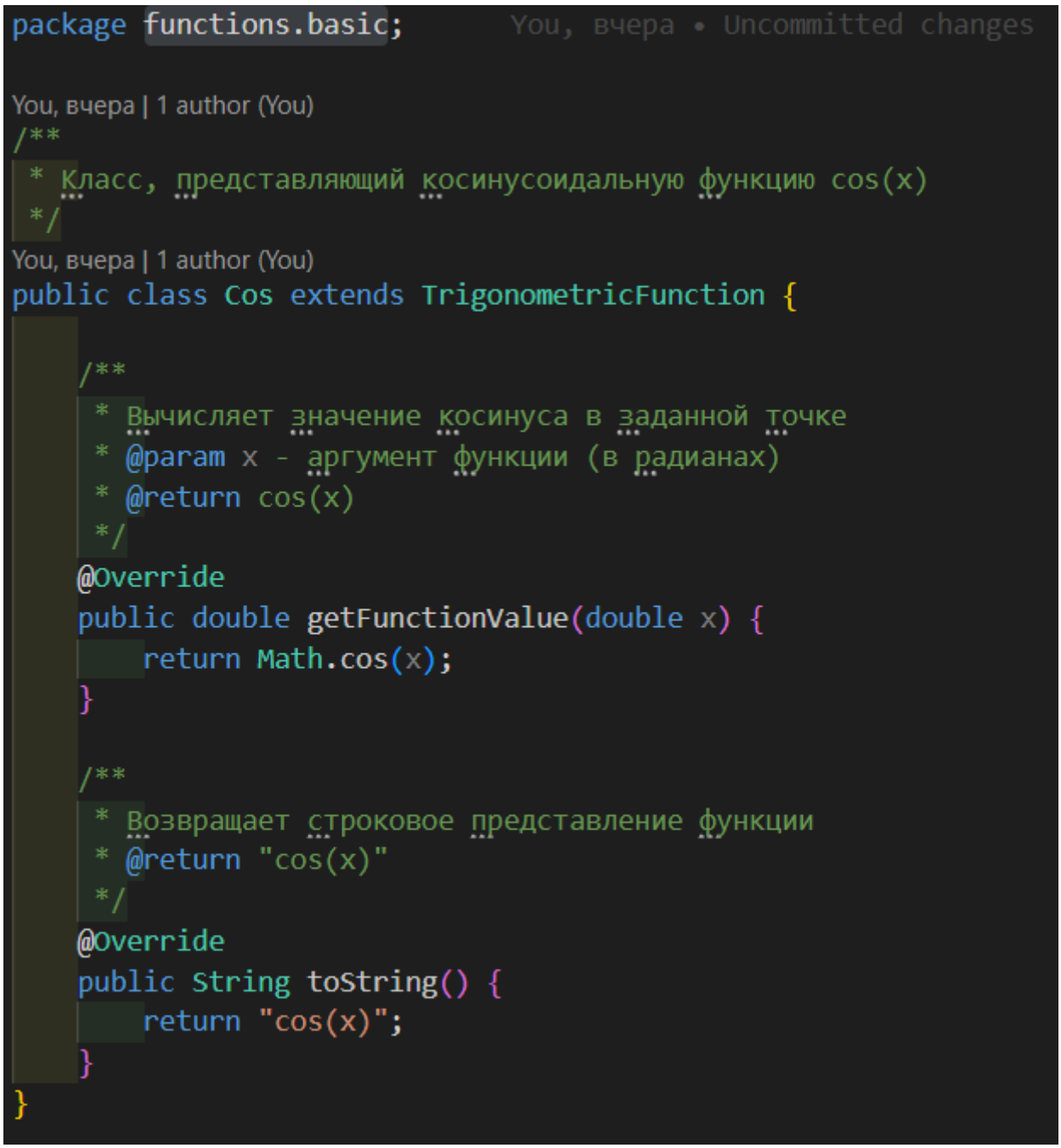
Создайте класс TrigonometricFunction, реализующий интерфейс Function и описывающий методы получения границ области определения.

```java
package functions.basic;       You, вчера • Uncommitted changes

import functions.Function;

You, вчера | 1 author (You)
/**
 * Базовый класс для тригонометрических функций
 * Определяет общую область определения (-∞, +∞)
 */
You, вчера | 1 author (You)
public abstract class TrigonometricFunction implements Function {

    /**
     * Возвращает значение левой границы области определения тригонометрических функций
     * @return -∞ (Double.NEGATIVE_INFINITY)
     */
    @Override
    public double getLeftDomainBorder() {
        return Double.NEGATIVE_INFINITY;
    }

    /**
     * Возвращает значение правой границы области определения тригонометрических функций
     * @return +∞ (Double.POSITIVE_INFINITY)
     */
    @Override
    public double getRightDomainBorder() {
        return Double.POSITIVE_INFINITY;
    }

    /**
     * Абстрактный метод для вычисления значения тригонометрической функции
     * @param x - аргумент функции (в радианах)
     * @return значение функции
     */
```

Создайте наследующие от него публичные классы Sin, Cos и Tan, объекты которых вычисляют, соответственно, значения синуса, косинуса и тангенса. Для получения значений следует воспользоваться методами Math.sin(), Math.cos() и Math.tan().

```java
package functions.basic;

You, вчера | 1 author (You)
/**
 * Класс, представляющий синусоидальную функцию sin(x)
 */
You, вчера | 1 author (You)
public class Sin extends TrigonometricFunction {

    /**
     * Вычисляет значение синуса в заданной точке
     * @param x - аргумент функции (в радианах)
     * @return sin(x)
     */
    @Override
    public double getFunctionValue(double x) {
        return Math.sin(x);
    }

    /**
     * Возвращает строковое представление функции
     * @return "sin(x)"
     */
    @Override
    public String toString() {
        return "sin(x)";
    }
}       You, вчера • Uncommitted changes
```

```java
package functions.basic;

You, вчера | 1 author (You)
/**
 * Класс, представляющий тангенс функцию tan(x)
 * Обратите внимание: тангенс имеет разрывы в точках π/2 + πk
 */
You, вчера | 1 author (You)
public class Tan extends TrigonometricFunction {

    /**
     * Вычисляет значение тангенса в заданной точке
     * @param x - аргумент функции (в радианах)
     * @return tan(x), или Double.NaN если cos(x) = 0 (разрыв)
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, не является ли точка точкой разрыва
        double cosValue = Math.cos(x);
        if (Math.abs(cosValue) < 1e-10) {
            return Double.NaN;
        }
        return Math.tan(x);
    }

    /**
     * Возвращает строковое представление функции
     * @return "tan(x)"
     */
    @Override
    public String toString() {
        return "tan(x)";
    }
}       You, вчера • Uncommitted changes
```



Задание 4

Создайте пакет functions.meta, в нём будут описаны классы функций, позволяющие комбинировать функции.

Создайте класс Sum, объекты которого представляют собой функции, являющиеся суммой двух других функций. Класс должен реализовывать интерфейс Function. Конструктор класса должен получать ссылки типа Function на объекты суммируемых функций, а область определения функции должна получаться как пересечение областей определения исходных функций.

```java
import functions.Function;

/**
 * Класс, представляющий сумму двух функций: f(x) + g(x)
 */
public class Sum implements Function {
    private final Function f;
    private final Function g;

    /**
     * Конструктор суммы двух функций
     * @param f первая функция
     * @param g вторая функция
     */
    public Sum(Function f, Function g) {
        this.f = f;
        this.g = g;
    }

    /**
     * Возвращает левую границу области определения суммы функций
     * @return максимальная из левых границ областей определения
     */
    @Override
    public double getLeftDomainBorder() {
        return Math.max(f.getLeftDomainBorder(), g.getLeftDomainBorder());
    }

    /**
     * Возвращает правую границу области определения суммы функций
     * @return минимальная из правых границ областей определения
     */
    @Override
    public double getRightDomainBorder() {
        return Math.min(f.getRightDomainBorder(), g.getRightDomainBorder());
    }

    /**
     * Вычисляет значение суммы функций в заданной точке
     * @param x аргумент функции
     * @return f(x) + g(x), или Double.NaN если точка вне области определения
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит пересечению областей определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
            return Double.NaN;
        }
        return f.getFunctionValue(x) + g.getFunctionValue(x);
    }

    /**
     * Возвращает строковое представление функции
     * @return "(f + g)"
     */
    @Override
    public String toString() {
        return "(" + f + " + " + g + ")";
    }
}
```

Аналогично, создайте класс Mult, объекты которого представляют собой функции, являющиеся произведением двух других функций.

```java
import functions.Function;

/**
 * Класс, представляющий произведение двух функций: f(x) * g(x)
 */
public class Mult implements Function {
    private final Function f;
    private final Function g;

    /**
     * Конструктор произведения двух функций
     * @param f первая функция
     * @param g вторая функция
     */
    public Mult(Function f, Function g) {
        this.f = f;
        this.g = g;
    }

    /**
     * Возвращает левую границу области определения произведения функций
     * @return максимальная из левых границ областей определения
     */
    @Override
    public double getLeftDomainBorder() {
        return Math.max(f.getLeftDomainBorder(), g.getLeftDomainBorder());
    }

    /**
     * Возвращает правую границу области определения произведения функций
     * @return минимальная из правых границ областей определения
     */
    @Override
    public double getRightDomainBorder() {
        return Math.min(f.getRightDomainBorder(), g.getRightDomainBorder());
    }

    /**
     * Вычисляет значение произведения функций в заданной точке
     * @param x аргумент функции
     * @return f(x) * g(x), или Double.NaN если точка вне области определения
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит пересечению областей определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
            return Double.NaN;
        }
        return f.getFunctionValue(x) * g.getFunctionValue(x);
    }

    /**
     * Возвращает строковое представление функции
     * @return "(f * g)"
     */
    @Override
    public String toString() {
        return "(" + f + " * " + g + ")";
    }
}
```

Создайте класс Power, объекты которого представляют собой функции, являющиеся степенью другой функции. Конструктор класса должен получать ссылку на объекты базовой функции и степень, в которую должны возводиться её значения. Область определения функции можно считать совпадающей с областью определения исходной функции (хотя математически это не всегда так).

```java
package functions.meta;

import functions.Function;

/**
 * Класс, представляющий функцию в степени: [f(x)]^power
 */
public class Power implements Function {
    private final Function f;
    private final double power;

    /**
     * Конструктор функции в степени
     * @param f базовая функция
     * @param power степень
     */
    public Power(Function f, double power) {
        this.f = f;
        this.power = power;
    }

    /**
     * Возвращает левую границу области определения
     * @return левая граница области определения базовой функции
     */
    @Override
    public double getLeftDomainBorder() {
        return f.getLeftDomainBorder();
    }

    /**
     * Возвращает правую границу области определения
     * @return правая граница области определения базовой функции
     */
    @Override
    public double getRightDomainBorder() {
        return f.getRightDomainBorder();
    }

    /**
     * Вычисляет значение функции в заданной точке
     * @param x аргумент функции
     * @return [f(x)]^power, или Double.NaN если точка вне области определения
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит области определения базовой функции
        if (x < f.getLeftDomainBorder() || x > f.getRightDomainBorder()) {
            return Double.NaN;
        }
        double value = f.getFunctionValue(x);

        // Проверяем особые случаи для возведения в степень
        if (Double.isNaN(value)) {
            return Double.NaN;
        }

        return Math.pow(value, power);
    }

    /**
     * Возвращает степень
     * @return степень
     */
    public double getPower() {
        return power;
    }
```

Создайте класс Scale, объекты которого описывают функции, полученные из исходных функций путём масштабирования вдоль осей координат. Конструктор класса должен получать ссылку на объект исходной функции, а также коэффициенты масштабирования вдоль оси абсцисс и оси ординат. Область определения функции должна получаться из области определения исходной функции масштабированием вдоль оси абсцисс, а значение функции – масштабированием значения исходной функции вдоль оси ординат. Коэффициенты масштабирования могут быть отрицательными.\

```java
package functions.meta;

import functions.Function;

/**
 * Класс, представляющий масштабирование функции: a * f(b * x)
 */
public class Scale implements Function {
    private final Function f;
    private final double scaleX;
    private final double scaleY;

    /**
     * Конструктор масштабированной функции
     * @param f исходная функция
     * @param scaleX коэффициент масштабирования по оси X (может быть отрицательным)
     * @param scaleY коэффициент масштабирования по оси Y (может быть отрицательным)
     */
    public Scale(Function f, double scaleX, double scaleY) {
        this.f = f;
        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }

    /**
     * Возвращает левую границу области определения после масштабирования
     * @return левая граница масштабированной области определения
     */
    @Override
    public double getLeftDomainBorder() {
        if (scaleX > 0) {
            return f.getLeftDomainBorder() / scaleX;
        } else if (scaleX < 0) {
            return f.getRightDomainBorder() / scaleX;
        } else {
            // Если scaleX = 0, функция определена только в нуле (если он входит в область определения f)
            // Но математически это сложный случай, вернем специальное значение
            return Double.NEGATIVE_INFINITY;
        }
    }

    /**
     * Возвращает правую границу области определения после масштабирования
     * @return правая граница масштабированной области определения
     */
    @Override
    public double getRightDomainBorder() {
        if (scaleX > 0) {
            return f.getRightDomainBorder() / scaleX;
        } else if (scaleX < 0) {
            return f.getLeftDomainBorder() / scaleX;
        } else {
            // Если scaleX = 0
            return Double.POSITIVE_INFINITY;
        }
    }

    /**
     * Вычисляет значение масштабированной функции в заданной точке
     * @param x аргумент функции
     * @return scaleY * f(scaleX * x)
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит области определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
            return Double.NaN;
```

Аналогично, создайте класс Shift, объекты которого описывают функции, полученные из исходных функций путём сдвига вдоль осей координат.

```java
package functions.meta;

import functions.Function;

/**
 * Класс, представляющий сдвиг функции: f(x + shiftX) + shiftY
 */
public class Shift implements Function {
    private final Function f;
    private final double shiftX;
    private final double shiftY;

    /**
     * Конструктор сдвинутой функции
     * @param f исходная функция
     * @param shiftX сдвиг по оси X (может быть отрицательным)
     * @param shiftY сдвиг по оси Y (может быть отрицательным)
     */
    public Shift(Function f, double shiftX, double shiftY) {
        this.f = f;
        this.shiftX = shiftX;
        this.shiftY = shiftY;
    }

    /**
     * Возвращает левую границу области определения после сдвига
     * @return левая граница сдвинутой области определения
     */
    @Override
    public double getLeftDomainBorder() {
        return f.getLeftDomainBorder() - shiftX;
    }

    /**
     * Возвращает правую границу области определения после сдвига
     * @return правая граница сдвинутой области определения
     */
    @Override
    public double getRightDomainBorder() {
        return f.getRightDomainBorder() - shiftX;
    }

    /**
     * Вычисляет значение сдвинутой функции в заданной точке
     * @param x аргумент функции
     * @return f(x + shiftX) + shiftY
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит области определения
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
            return Double.NaN;
        }

        double innerX = x + shiftX;
        // Проверяем, что innerX принадлежит области определения f
        if (innerX < f.getLeftDomainBorder() || innerX > f.getRightDomainBorder()) {
            return Double.NaN;
        }

        double value = f.getFunctionValue(innerX);
        if (Double.isNaN(value)) {
            return Double.NaN;
        }

        return value + shiftY;
    }
```

Также создайте класс Composition, объекты которого описывают композицию двух исходных функций. Конструктор класса должен получать ссылки на объекты первой и второй функции. Область определения функции можно считать совпадающей с областью определения исходной функции (хотя математически это не всегда так).

```java
package functions.meta;

import functions.Function;

/**
 * Класс, представляющий композицию двух функций: g(f(x))
 */
public class Composition implements Function {
    private final Function f;
    private final Function g;

    /**
     * Конструктор композиции функций
     * @param f внутренняя функция
     * @param g внешняя функция
     */
    public Composition(Function f, Function g) {
        this.f = f;
        this.g = g;
    }

    /**
     * Возвращает левую границу области определения композиции
     * @return левая граница области определения внутренней функции
     */
    @Override
    public double getLeftDomainBorder() {
        return f.getLeftDomainBorder();
    }

    /**
     * Возвращает правую границу области определения композиции
     * @return правая граница области определения внутренней функции
     */
    @Override
    public double getRightDomainBorder() {
        return f.getRightDomainBorder();
    }

    /**
     * Вычисляет значение композиции функций в заданной точке
     * @param x аргумент функции
     * @return g(f(x))
     */
    @Override
    public double getFunctionValue(double x) {
        // Проверяем, что точка принадлежит области определения внутренней функции
        if (x < f.getLeftDomainBorder() || x > f.getRightDomainBorder()) {
            return Double.NaN;
        }

        double innerValue = f.getFunctionValue(x);
        if (Double.isNaN(innerValue)) {
            return Double.NaN;
        }

        // Проверяем, что значение внутренней функции принадлежит области определения внешней функции
        if (innerValue < g.getLeftDomainBorder() || innerValue > g.getRightDomainBorder()) {
            return Double.NaN;
        }

        return g.getFunctionValue(innerValue);
    }

    /**
     * Возвращает строковое представление функции
     * @return "g(f(x))"
```

Задание 5

```java
package functions;

import functions.meta.*;

/**
 * Вспомогательный класс со статическими методами для работы с функциями.
 * Не может быть инстанциирован.
 */
public class Functions {

    /**
     * Приватный конструктор для предотвращения создания экземпляров класса
     */
    private Functions() {
        throw new AssertionError("Нельзя создавать экземпляры класса Functions");
    }

    /**
     * Возвращает функцию, полученную из исходной сдвигом вдоль осей
     * @param f исходная функция
     * @param shiftX сдвиг по оси X (может быть отрицательным)
     * @param shiftY сдвиг по оси Y (может быть отрицательным)
     * @return функция f(x + shiftX) + shiftY
     */
    public static Function shift(Function f, double shiftX, double shiftY) {
        return new Shift(f, shiftX, shiftY);
    }

    /**
     * Возвращает функцию, полученную из исходной масштабированием вдоль осей
     * @param f исходная функция
     * @param scaleX коэффициент масштабирования по оси X (может быть отрицательным)
     * @param scaleY коэффициент масштабирования по оси Y (может быть отрицательным)
     * @return функция scaleY * f(scaleX * x)
     */
    public static Function scale(Function f, double scaleX, double scaleY) {
        return new Scale(f, scaleX, scaleY);
    }

    /**
     * Возвращает функцию, являющуюся заданной степенью исходной
     * @param f исходная функция
     * @param power степень
     * @return функция [f(x)]^power
     */
    public static Function power(Function f, double power) {
        return new Power(f, power);
    }
```

```
49
50      /**
51       * Возвращает функцию, являющуюся суммой двух исходных
52       * @param f1 первая функция
53       * @param f2 вторая функция
54       * @return функция f1(x) + f2(x)
55       */
56      public static Function sum(Function f1, Function f2) {
57          return new Sum(f1, f2);
58      }
59
60      /**
61       * Возвращает функцию, являющуюся произведением двух исходных
62       * @param f1 первая функция
63       * @param f2 вторая функция
64       * @return функция f1(x) * f2(x)
65       */
66      public static Function mult(Function f1, Function f2) {
67          return new Mult(f1, f2);
68      }
69
70      /**
71       * Возвращает функцию, являющуюся композицией двух исходных
72       * @param f1 внутренняя функция
73       * @param f2 внешняя функция
74       * @return функция f2(f1(x))
75       */
76      public static Function composition(Function f1, Function f2) {
77          return new Composition(f1, f2);
78      }
79
80      /**
81       * Дополнительный метод: создает квадрат функции
82       * @param f исходная функция
83       * @return функция [f(x)]^2
84       */
85      public static Function square(Function f) {
86          return new Power(f, 2);
87      }
88
```

```
88
89      /**
90       * Дополнительный метод: создает обратную функцию (1/f(x))
91       * @param f исходная функция
92       * @return функция 1/f(x)
93       */
94      public static Function inverse(Function f) {
95          return new Power(f, -1);
96      }
97
98      /**
99       * Дополнительный метод: создает разность двух функций
100      * @param f1 первая функция
101      * @param f2 вторая функция
102      * @return функция f1(x) - f2(x)
103      */
104     public static Function difference(Function f1, Function f2) {
105         return sum(f1, scale(f2, 1, -1));
106     }
107
108     /**
109      * Дополнительный метод: создает частное двух функций
110      * @param f1 числитель
111      * @param f2 знаменатель
112      * @return функция f1(x) / f2(x)
113      */
114     public static Function quotient(Function f1, Function f2) {
115         return mult(f1, inverse(f2));
116     }
117 }
```

Задание 6

```java
// Метод табулирования функции на заданном отрезке
public static TabulatedFunction tabulate(Function function, double leftX, double rightX, int pointsCount) {
    // Проверка количества точек
    if (pointsCount < 2) {
        throw new IllegalArgumentException(s: "Количество точек должно быть не менее 2");
    }

    // Проверка границ отрезка
    if (leftX >= rightX) {
        throw new IllegalArgumentException(s: "Левая граница должна быть меньше правой границы");
    }

    // Проверка области определения
    if (leftX < function.getLeftDomainBorder() || rightX > function.getRightDomainBorder()) {
        throw new IllegalArgumentException(
            "Границы табулирования [" + leftX + ", " + rightX +
            "] выходят за область определения функции [" +
            function.getLeftDomainBorder() + ", " + function.getRightDomainBorder() + "]"
        );
    }

    // Создание массива точек
    FunctionPoint[] points = new FunctionPoint[pointsCount];
    double step = (rightX - leftX) / (pointsCount - 1);

    // Заполнение массива точек
    for (int i = 0; i < pointsCount; i++) {
        double x = leftX + i * step;
        double y = function.getFunctionValue(x);
        points[i] = new FunctionPoint(x, y);
    }

    // Возвращаем табулированную функцию
    return new ArrayTabulatedFunction(points);
}
```

Задание 7

```java
// Метод вывода табулированной функции в байтовый поток
public static void outputTabulatedFunction(TabulatedFunction function, OutputStream out) throws IOException {
    DataOutputStream dataOut = new DataOutputStream(out);
    int pointsCount = function.getPointsCount();

    dataOut.writeInt(pointsCount);

    for (int i = 0; i < pointsCount; i++) {
        dataOut.writeDouble(function.getPointX(i));
        dataOut.writeDouble(function.getPointY(i));
    }

    dataOut.flush();
}
```

v

```java
// Метод ввода табулированной функции из байтового потока
public static TabulatedFunction inputTabulatedFunction(InputStream in) throws IOException {
    DataInputStream dataIn = new DataInputStream(in);

    int pointsCount = dataIn.readInt();
    FunctionPoint[] points = new FunctionPoint[pointsCount];

    for (int i = 0; i < pointsCount; i++) {
        double x = dataIn.readDouble();
        double y = dataIn.readDouble();
        points[i] = new FunctionPoint(x, y);
    }

    return new ArrayTabulatedFunction(points);
}
```

```java
// Метод записи табулированной функции в символьный поток
public static void writeTabulatedFunction(TabulatedFunction function, Writer out) throws IOException {
    PrintWriter writer = new PrintWriter(out);
    int pointsCount = function.getPointsCount();

    writer.print(pointsCount);

    for (int i = 0; i < pointsCount; i++) {
        writer.print(" " + function.getPointX(i));
        writer.print(" " + function.getPointY(i));
    }

    writer.flush();
}
```

```java
// Метод чтения табулированной функции из символьного потока
public static TabulatedFunction readTabulatedFunction(Reader in) throws IOException {
    StreamTokenizer tokenizer = new StreamTokenizer(in);
    tokenizer.eolIsSignificant(flag: false);

    tokenizer.nextToken();
    if (tokenizer.ttype != StreamTokenizer.TT_NUMBER) {
        throw new IOException(message: "Ожидалось количество точек");
    }
    int pointsCount = (int) tokenizer.nval;

    FunctionPoint[] points = new FunctionPoint[pointsCount];

    for (int i = 0; i < pointsCount; i++) {
        tokenizer.nextToken();
        if (tokenizer.ttype != StreamTokenizer.TT_NUMBER) {
            throw new IOException(message: "Ожидалась координата X");
        }
        double x = tokenizer.nval;

        tokenizer.nextToken();
        if (tokenizer.ttype != StreamTokenizer.TT_NUMBER) {
            throw new IOException(message: "Ожидалась координата Y");
        }
        double y = tokenizer.nval;

        points[i] = new FunctionPoint(x, y);
    }

    return new ArrayTabulatedFunction(points);
}
```

Задание 8

```java
// 1. Тестирование базовых функций Sin и Cos
private static void testSinCos() {
    Function sin = new Sin();
    Function cos = new Cos();

    System.out.println(x: "Значения функций на отрезке [0, π] с шагом 0.1:");
    System.out.println(x: "┌──────────┬──────────────┬──────────────┐");
    System.out.println(x: "│    x     │    sin(x)    │    cos(x)    │");
    System.out.println(x: "├──────────┼──────────────┼──────────────┤");

    for (double x = 0; x <= Math.PI + 1e-9; x += 0.1) {
        System.out.printf(format: "│ %7.3f  │ %10.6f   │ %10.6f   │%n",
            x, sin.getFunctionValue(x), cos.getFunctionValue(x));
    }

    System.out.println(x: "└──────────┴──────────────┴──────────────┘");
}
```

```java
// 2. Тестирование табулированных аналогов Sin и Cos
private static void testTabulatedSinCos() {
    Function sin = new Sin();
    Function cos = new Cos();

    // Создаем табулированные аналоги с 10 точками
    TabulatedFunction tabSin = TabulatedFunctions.tabulate(sin, leftX: 0, Math.PI, pointsCount: 10);
    TabulatedFunction tabCos = TabulatedFunctions.tabulate(cos, leftX: 0, Math.PI, pointsCount: 10);

    System.out.println(x: "Сравнение с исходными функциями (первые 5 точек):");
    System.out.println(x: "┌──────────┬──────────────┬──────────────┬──────────────┬──────────────┐");
    System.out.println(x: "│    x     │  точн.sin    │  таб. sin    │  точн.cos    │  таб. cos    │");
    System.out.println(x: "├──────────┼──────────────┼──────────────┼──────────────┼──────────────┤");

    for (int i = 0; i < Math.min(a: 5, tabSin.getPointsCount()); i++) {
        double x = tabSin.getPointX(i);
        double exactSin = sin.getFunctionValue(x);
        double tabSinVal = tabSin.getPointY(i);
        double exactCos = cos.getFunctionValue(x);
        double tabCosVal = tabCos.getPointY(i);

        System.out.printf(format: "│ %7.3f  │ %10.6f   │ %10.6f   │ %10.6f   │ %10.6f   │%n",
            x, exactSin, tabSinVal, exactCos, tabCosVal);
    }

    System.out.println(x: "└──────────┴──────────────┴──────────────┴──────────────┴──────────────┘");
}
```

```java
// 3. Тестирование суммы квадратов
private static void testSumOfSquares() {
    // Создаем табулированные функции с разным количеством точек
    int[] pointCounts = {5, 10, 20, 50};

    System.out.println(x: "Исследование влияния количества точек на точность sin²(x) + cos²(x):");
    System.out.println(x: "┌──────────────┬──────────────────────┐");
    System.out.println(x: "│ Кол-во точек │ Макс. отклонение     │");
    System.out.println(x: "├──────────────┼──────────────────────┤");

    for (int n : pointCounts) {
        TabulatedFunction tabSin = TabulatedFunctions.tabulate(new Sin(), leftX: 0, Math.PI, n);
        TabulatedFunction tabCos = TabulatedFunctions.tabulate(new Cos(), leftX: 0, Math.PI, n);

        // Создаем сумму квадратов
        Function sinSquared = Functions.power(tabSin, power: 2);
        Function cosSquared = Functions.power(tabCos, power: 2);
        Function sumOfSquares = Functions.sum(sinSquared, cosSquared);

        double maxError = 0;
        for (double x = 0; x <= Math.PI; x += 0.01) {
            double value = sumOfSquares.getFunctionValue(x);
            double error = Math.abs(value - 1.0);
            if (error > maxError) maxError = error;
        }

        System.out.printf(format: "│ %12d │ %18.6f │%n", n, maxError);
    }

    System.out.println(x: "└──────────────┴──────────────────────┘");
}
```

```java
// 4. Тестирование текстового файлового ввода/вывода (экспонента)
private static void testTextFileIOExp() throws IOException {
    // Создаем табулированную экспоненту
    TabulatedFunction tabExp = TabulatedFunctions.tabulate(new Exp(), leftX: 0, rightX: 10, pointsCount: 11);

    // Записываем в текстовый файл
    String textFileName = "exp_function.txt";
    try (FileWriter writer = new FileWriter(textFileName)) {
        TabulatedFunctions.writeTabulatedFunction(tabExp, writer);
        System.out.println("Функция записана в текстовый файл: " + textFileName);
    }

    // Читаем из текстового файла
    TabulatedFunction readExp;
    try (FileReader reader = new FileReader(textFileName)) {
        readExp = TabulatedFunctions.readTabulatedFunction(reader);
        System.out.println(x: "Функция прочитана из текстового файла");
    }

    // Сравниваем значения
    System.out.println(x: "\nСравнение значений с шагом 1:");
    System.out.println(x: "┌──────┬──────────────┬──────────────┬──────────────┐");
    System.out.println(x: "│  x   │  исходная    │  прочитанная │  разница     │");
    System.out.println(x: "├──────┼──────────────┼──────────────┼──────────────┤");

    for (double x = 0; x <= 10; x += 1.0) {
        double original = tabExp.getFunctionValue(x);
        double read = readExp.getFunctionValue(x);
        double diff = Math.abs(original - read);
        System.out.printf(format: "│ %3.0f │ %12.6f │ %12.6f │ %12.6e │%n",
            x, original, read, diff);
    }

    System.out.println(x: "└──────┴──────────────┴──────────────┴──────────────┘");
}
```

```java
// 5. Тестирование бинарного файлового ввода/вывода (логарифм)
private static void testBinaryFileIOLn() throws IOException {
    // Создаем табулированный натуральный логарифм
    TabulatedFunction tabLn = TabulatedFunctions.tabulate(new Log(Math.E), leftX: 0.1, rightX: 10, pointsCount: 11);

    // Записываем в бинарный файл
    String binaryFileName = "log_function.dat";
    try (FileOutputStream out = new FileOutputStream(binaryFileName)) {
        TabulatedFunctions.outputTabulatedFunction(tabLn, out);
        System.out.println("Функция записана в бинарный файл: " + binaryFileName);
    }

    // Читаем из бинарного файла
    TabulatedFunction readLn;
    try (FileInputStream in = new FileInputStream(binaryFileName)) {
        readLn = TabulatedFunctions.inputTabulatedFunction(in);
        System.out.println(x: "Функция прочитана из бинарного файла");
    }

    // Сравниваем значения
    System.out.println(x: "\nСравнение значений с шагом 1:");
    System.out.println(x: "┌────────┬──────────────────┬──────────────────┬──────────────────┐");
    System.out.println(x: "│   x    │  исходная        │  прочитанная     │  разница         │");
    System.out.println(x: "├────────┼──────────────────┼──────────────────┼──────────────────┤");

    for (double x = 0.1; x <= 10; x += 1.0) {
        double original = tabLn.getFunctionValue(x);
        double read = readLn.getFunctionValue(x);
        double diff = Math.abs(original - read);
        System.out.printf(format: "│ %5.1f  │ %12.6f     │ %12.6f     │ %12.6e     │%n",
            x, original, read, diff);
    }

    System.out.println(x: "└────────┴──────────────────┴──────────────────┴──────────────────┘");
}
```

Задание 9

```java
// 6. Тестирование сериализации
private static void testSerialization() throws Exception {
    // Создаем функцию ln(exp(x)) = x
    Function exp = new Exp();
    Function ln = new Log(Math.E);
    Function lnExp = new Composition(ln, exp); // ln(exp(x)) = x

    // Табулируем функцию
    TabulatedFunction tabulatedFunction = TabulatedFunctions.tabulate(lnExp, leftX: 0, rightX: 10, pointsCount: 11);

    System.out.println("Создана функция ln(exp(x)) = x с " +
        tabulatedFunction.getPointsCount() + " точками");

    // Тест 1: Serializable
    System.out.println(x: "\n6.1 Сериализация с использованием Serializable:");
    testSerializable(tabulatedFunction, fileName: "function_serializable.ser");

    // Тест 2: Externalizable
    System.out.println(x: "\n6.2 Сериализация с использованием Externalizable:");
    testExternalizable(tabulatedFunction, fileName: "function_externalizable.ser");
}
```

Вывод консоли

1. ЗАДАНИЕ 8: БАЗОВЫЕ ФУНКЦИИ SIN И COS

Значения функций на отрезке [0, ?] с шагом 0.1:

x	sin(x)	cos(x)
0.000	0.000000	1.000000
0.100	0.099833	0.995004
0.200	0.198669	0.980067
0.300	0.295520	0.955336
0.400	0.389418	0.921061
0.500	0.479426	0.877583
0.600	0.564642	0.825336
0.700	0.644218	0.764842
0.800	0.717356	0.696707
0.900	0.783327	0.621610
1.000	0.841471	0.540302
1.100	0.891207	0.453596
1.200	0.932039	0.362358
1.300	0.963558	0.267499
1.400	0.985450	0.169967
1.500	0.997495	0.070737
1.600	0.999574	-0.029200
1.700	0.991665	-0.128844
1.800	0.973848	-0.227202
1.900	0.946300	-0.323290
2.000	0.909297	-0.416147
2.100	0.863209	-0.504846
2.200	0.808496	-0.588501
2.300	0.745705	-0.666276
2.400	0.675463	-0.737394
2.500	0.598472	-0.801144
2.600	0.515501	-0.856889
2.700	0.427380	-0.904072
2.800	0.334988	-0.942222
2.900	0.239249	-0.970958
3.000	0.141120	-0.989992
3.100	0.041581	-0.999135

2. ЗАДАНИЕ 8: ТАБУЛИРОВАННЫЕ АНАЛОГИ SIN И COS

Сравнение с исходными функциями (первые 5 точек):

x	точн.sin	таб. sin	точн.cos	таб. cos
0.000	0.000000	0.000000	1.000000	1.000000
0.349	0.342020	0.342020	0.939693	0.939693
0.698	0.642788	0.642788	0.766044	0.766044
1.047	0.866025	0.866025	0.500000	0.500000
1.396	0.984808	0.984808	0.173648	0.173648

3. ЗАДАНИЕ 8: СУММА КВАДРАТОВ SIN?(x) + COS?(x)

Исследование влияния количества точек на точность sin?(x) + cos?(x):

Кол-во точек	Макс. отклонение
5	0.146445
10	0.030154
20	0.006819
50	0.001027

4. ЗАДАНИЕ 8: ТЕКСТОВЫЙ ФАЙЛОВЫЙ ВВОД/ВЫВОД (ЭКСПОНЕНТА)

Функция записана в текстовый файл: exp_function.txt
Функция прочитана из текстового файла

Сравнение значений с шагом 1:

x	исходная	прочитанная	разница
0	1.000000	1.000000	0.000000e+00
1	2.718282	2.718282	0.000000e+00
2	7.389056	7.389056	0.000000e+00
3	20.085537	20.085537	0.000000e+00
4	54.598150	54.598150	7.105427e-15
5	148.413159	148.413159	0.000000e+00
6	403.428793	403.428793	0.000000e+00
7	1096.633158	1096.633158	0.000000e+00
8	2980.957987	2980.957987	0.000000e+00
9	8103.083928	8103.083928	0.000000e+00
10	22026.465795	22026.465795	3.637979e-12

5. ЗАДАНИЕ 8: БИНАРНЫЙ ФАЙЛОВЫЙ ВВОД/ВЫВОД (ЛОГАРИФМ)

Функция записана в бинарный файл: log_function.dat
Функция прочитана из бинарного файла

Сравнение значений с шагом 1:

x	исходная	прочитанная	разница
0.1	-2.302585	-2.302585	0.000000e+00
1.1	0.092705	0.092705	0.000000e+00
2.1	0.740233	0.740233	0.000000e+00
3.1	1.130147	1.130147	0.000000e+00
4.1	1.409999	1.409999	0.000000e+00
5.1	1.628429	1.628429	0.000000e+00
6.1	1.807603	1.807603	0.000000e+00
7.1	1.959502	1.959502	0.000000e+00
8.1	2.091344	2.091344	0.000000e+00
9.1	2.207812	2.207812	0.000000e+00

6. ЗАДАНИЕ 9: СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ

Создана функция ln(exp(x)) = x с 11 точками

6.1 Сериализация с использованием Serializable:
  Функция сериализована в файл: function_serializable.ser
  Функция десериализована из файла
  Сравнение значений:

x	исходная	восстановл.	разница
0	NaN	NaN	NaN
1	1.000000	1.000000	0.000000e+00
2	2.000000	2.000000	0.000000e+00
3	3.000000	3.000000	0.000000e+00
4	4.000000	4.000000	0.000000e+00
5	5.000000	5.000000	0.000000e+00
6	6.000000	6.000000	0.000000e+00
7	7.000000	7.000000	0.000000e+00
8	8.000000	8.000000	0.000000e+00
9	9.000000	9.000000	0.000000e+00
10	10.000000	10.000000	0.000000e+00

  Все значения совпадают: true
  Размер файла: 440 байт

6.2 Сериализация с использованием Externalizable:
Функция сериализована в файл: function_externalizable.ser
Функция десериализована из файла
Сравнение значений:

x	исходная	восстановл.	разница
0	NaN	NaN	NaN
1	1.000000	1.000000	0.000000e+00
2	2.000000	2.000000	0.000000e+00
3	3.000000	3.000000	0.000000e+00
4	4.000000	4.000000	0.000000e+00
5	5.000000	5.000000	0.000000e+00
6	6.000000	6.000000	0.000000e+00
7	7.000000	7.000000	0.000000e+00
8	8.000000	8.000000	0.000000e+00
9	9.000000	9.000000	0.000000e+00
10	10.000000	10.000000	0.000000e+00

Все значения совпадают: true
Размер файла: 440 байт

```
7. АНАЛИЗ ФОРМАТОВ ФАЙЛОВ И ВЫВОДЫ

Содержимое файлов и размеры:

1. Текстовый файл (exp_function.txt):
   Размер: 235 байт
   Первые 100 символов: 11 0.0 1.0 1.0 2.718281828459045 2.0 7.38905609893065 3.0 20.085536923187668 4.0 54.598150033144236 ...

2. Бинарный файл (log_function.dat):
   Размер: 180 байт

3. Файл сериализации Serializable (function_serializable.ser):
   Размер: 440 байт

4. Файл сериализации Externalizable (function_externalizable.ser):
   Размер: 440 байт

Выводы о форматах хранения:
1. Текстовый формат:
   + Человекочитаемый
   + Легко отлаживать
   - Больший размер файла
   - Медленнее чтение/запись

2. Бинарный формат (DataStreams):
   + Компактный размер
   + Быстрая работа
   - Не человекочитаемый

3. Сериализация Serializable:
   + Простая реализация
   + Автоматическая сериализация
   - Большой размер (метаданные)

4. Сериализация Externalizable:
   + Полный контроль
   + Компактный размер
   - Сложная реализация
```