

## Лабораторная работа №7

Выполнил работу  
Гапанюк Антон Андреевич  
Группа: 6204-010302D

Самара, 2025 г.

## Оглавление:

1. [Задание №1](#)
2. [Задание №2](#)
3. [Задание №3](#)
4. [Доработка лабораторной](#)

**Цель работы:** внести изменения в существующий набор типов табулированных функций, позволяющие обрабатывать точки функций по порядку (паттерн «Итератор»), а также выбирать тип объекта табулированной функции при его неявном создании (паттерн «Фабричный метод» и средства рефлексии).

## Ход работы

[Вернуться к оглавлению](#)

### Задание №1

Сделаем так, чтобы все объекты типа **TabulatedFunction** можно было использовать в качестве объекта-агрегата в «улучшенном цикле **for**» (вариант **for-each**), извлекаемые объекты при этом должны иметь тип **FunctionPoint**.

В интерфейсе **TabulatedFunction** добавим родительский тип **Iterable<FunctionPoint>**

В классах, реализующих интерфейс **TabulatedFunction**, добавим требующийся метод, возвращающий объект итератора. Классы итераторов сделаем анонимными.

Метод удаления в итераторах будет всегда выбрасывать исключение **UnsupportedOperationException**.

Метод получения следующего элемента будет выбрасывать исключение **NoSuchElementException**, если следующего элемента нет.

### ArrayTabulatedFunction

```
@Override
public Iterator<FunctionPoint> iterator(){
    return new Iterator<FunctionPoint>(){
        private int curIndex = 0;

        @Override
        public boolean hasNext(){
            return curIndex < points.length;
        }
    };
}
```

```

// Метод получения следующего элемента
@Override
public FunctionPoint next() {
    if (!hasNext()){
        throw new java.util.NoSuchElementException("Больше нет
доступных точек");
    }
    // Возвращаем копию точки для сохранения инкапсуляции
    return new FunctionPoint(points[curIndex++]);
}

// Метод удаления
@Override
public void remove(){
    throw new UnsupportedOperationException("Операция удаления
недоступна");
}
};

```

## LinkedListTabulatedFunction

```

@Override
public Iterator<FunctionPoint> iterator(){
    return new Iterator<FunctionPoint>() {
        private FunctionNode curNode = head.getNext();
        private FunctionNode endNode = head;

        @Override
        public boolean hasNext(){
            return curNode != endNode;
        }

        // Метод получения следующего элемента
        @Override
        public FunctionPoint next(){
            if (!hasNext()){
                throw new java.util.NoSuchElementException("Больше нет
доступных точек");
            }

```

```

        }

        FunctionPoint point = new FunctionPoint(curNode.getPoint());
        curNode = curNode.getNext();
        return point;
    }

    // Метод удаления
    @Override
    public void remove() {
        throw new UnsupportedOperationException("Операция удаления
недоступна");
    }
}

```

В методе **main()** программы проверим работу итераторов классов табулированных функций.

```

public static void testIterator() {
    System.out.println("== Тестирование итератора для
ArrayTabulatedFunction ==");

    // Создаем тестовую функцию для ArrayTabulatedFunction
    double[] xValues = {1.0, 2.0, 3.0, 4.0, 5.0};
    double[] yValues = {2.0, 4.0, 6.0, 8.0, 10.0};
    TabulatedFunction arrayFunc = new ArrayTabulatedFunction(xValues,
yValues);

    System.out.println("Итерация по точкам ArrayTabulatedFunction:");
    for (FunctionPoint point : arrayFunc) {
        System.out.println(point);
    }

    System.out.println("\n== Тестирование итератора для
LinkedListTabulatedFunction ==");

    // Создаем массив точек для LinkedListTabulatedFunction
    FunctionPoint[] points = {
        new FunctionPoint(1.0, 2.0),
        new FunctionPoint(2.0, 4.0),
    }
}

```

```

        new FunctionPoint(3.0, 6.0),
        new FunctionPoint(4.0, 8.0),
        new FunctionPoint(5.0, 10.0)
    };

    TabulatedFunction linkedListFunc = new
LinkedListTabulatedFunction(points);

    System.out.println("Итерация по точкам
LinkedListTabulatedFunction:");
    for (FunctionPoint point : linkedListFunc) {
        System.out.println(point);
    }
}

```

### **Вывод в консоли:**

```

==== Тестирование итератора для ArrayTabulatedFunction ====
Итерация по точкам ArrayTabulatedFunction:
(1.0, 2.0)
(2.0, 4.0)
(3.0, 6.0)
(4.0, 8.0)
(5.0, 10.0)

==== Тестирование итератора для LinkedListTabulatedFunction ====
Итерация по точкам LinkedListTabulatedFunction:
(1.0, 2.0)
(2.0, 4.0)
(3.0, 6.0)
(4.0, 8.0)
(5.0, 10.0)

```

[Вернуться к оглавлению](#)

### **Задание №2**

В пакете **functions** опишем базовый интерфейс фабрик табулированных функций **TabulatedFunctionFactory**. Интерфейс должен объявлять три перегруженных метода **TabulatedFunction createTabulatedFunction()**,

параметры которых соответствуют параметрам конструкторов классов табулированных функций.

```
package functions;

public interface TabulatedFunctionFactory {

    // Три перегруженных метода, соответствующих конструкторам
    TabulatedFunction create(double leftX, double rightX, int pointsCount);
    TabulatedFunction create(double leftX, double rightX, double[] values);
    TabulatedFunction create(FunctionPoint[] points);
}
```

Для удобства опишем в классах **ArrayTabulatedFunction** и **LinkedListTabulatedFunction** классы фабрик **ArrayTabulatedFunctionFactory** и **LinkedListTabulatedFunctionFactory**, реализующие интерфейс фабрики и порождающие объекты соответствующих классов табулированных функций. Сделаем классы фабрик вложенными и публичными.

```
// Класс фабрики
public static class ArrayTabulatedFunctionFactory implements
TabulatedFunctionFactory {

    @Override
    public TabulatedFunction create(double leftX, double rightX, int
pointsCount){
        return new ArrayTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override
    public TabulatedFunction create(double leftX, double rightX, double[]
values) {
        return new ArrayTabulatedFunction(leftX, rightX, values);
    }

    @Override
    public TabulatedFunction create(FunctionPoint[] points) {
        return new ArrayTabulatedFunction(points);
    }
}
```

```

// Класс фабрики

    public static class LinkedListTabulatedFunctionFactory implements
TabulatedFunctionFactory {
        @Override
        public TabulatedFunction create(double leftX, double rightX, int
pointsCount) {
            return new LinkedListTabulatedFunction(leftX, rightX, pointsCount);
        }

        @Override
        public TabulatedFunction create(double leftX, double rightX, double[]
values) {
            return new LinkedListTabulatedFunction(leftX, rightX, values);
        }

        @Override
        public TabulatedFunction create(FunctionPoint[] points) {
            return new LinkedListTabulatedFunction(points);
        }
    }
}

```

В классе **TabulatedFunctions** объявим приватное статическое поле типа **TabulatedFunctionFactory** и проинициализируем его объектом одного из описанных классов фабрик. Также объявим метод **setTabulatedFunctionFactory()**, позволяющий заменить объект фабрики.

Ещё в классе **TabulatedFunctions** опишем три перегруженных метода **TabulatedFunction createTabulatedFunction()**, возвращающих объекты табулированных функций, созданные с помощью текущей фабрики.

```

// Приватное статическое поле фабрики
private static TabulatedFunctionFactory factory = new
ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();

// Метод для замены фабрики
public static void setTabulatedFunctionFactory(TabulatedFunctionFactory
newFactory){
    factory = newFactory;
}

```

```

// Три перегруженных метода создания табулированных функций
public static TabulatedFunction createTabulatedFunction(double leftX,
double rightX, int pointsCount) {
    return factory.create(leftX, rightX, pointsCount);
}

public static TabulatedFunction createTabulatedFunction(double leftX,
double rightX, double[] values) {
    return factory.create(leftX, rightX, values);
}

public static TabulatedFunction createTabulatedFunction(FunctionPoint[]
points) {
    return factory.create(points);
}

```

В остальных методах класса, где требуется создание объектов табулированных функций, заменим явное создание объектов с помощью конструкторов на вызов соответствующего метода **createTabulatedFunction()**.

В методе **main()** проверим работу фабрик.

```

public static void testFactory() {
    System.out.println("==> Тестирование фабрик табулированных функций
==>");

    Function f = new Cos();
    TabulatedFunction tf;

    // Тестируем фабрику по умолчанию (должна быть
    ArrayTabulatedFunction)
    System.out.println("\n1. Фабрика по умолчанию:");
    tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
    System.out.println("    Тип: " + tf.getClass().getSimpleName());

    // Меняем на LinkedList фабрику
    System.out.println("\n2. LinkedList фабрика:");
    TabulatedFunctions.setTabulatedFunctionFactory(
        new
    LinkedListTabulatedFunction.LinkedListTabulatedFunctionFactory());

```

```
tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
System.out.println("    Тип: " + tf.getClass().getSimpleName());

// Возвращаем Array фабрику
System.out.println("\n3. Array фабрика:");
TabulatedFunctions.setTabulatedFunctionFactory(
    new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory());
tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
System.out.println("    Тип: " + tf.getClass().getSimpleName());

// Дополнительное тестирование других методов создания
System.out.println("\n4. Тестирование разных методов создания:");

// Тест создания через границы и количество точек
TabulatedFunction tf1 = TabulatedFunctions.createTabulatedFunction(0,
10, 5);
System.out.println("    create(0, 10, 5): " +
tf1.getClass().getSimpleName());

// Тест создания через границы и значения
double[] values = {1, 2, 3, 4, 5};
TabulatedFunction tf2 = TabulatedFunctions.createTabulatedFunction(0,
10, values);
System.out.println("    create(0, 10, values): " +
tf2.getClass().getSimpleName());

// Тест создания через массив точек
FunctionPoint[] points = {
    new FunctionPoint(0, 1),
    new FunctionPoint(2, 3),
    new FunctionPoint(4, 5)
};
TabulatedFunction tf3 =
TabulatedFunctions.createTabulatedFunction(points);
System.out.println("    create(points): " +
tf3.getClass().getSimpleName());

System.out.println("\n==== Тестирование завершено ===");
}
```

Вывод в консоли:

```
==== Тестирование фабрик табулированных функций ====
```

1. Фабрика по умолчанию:

    Тип: ArrayTabulatedFunction

2. LinkedList фабрика:

    Тип: LinkedListTabulatedFunction

3. Array фабрика:

    Тип: ArrayTabulatedFunction

4. Тестирование разных методов создания:

    create(0, 10, 5): ArrayTabulatedFunction

    create(0, 10, values): ArrayTabulatedFunction

    create(points): ArrayTabulatedFunction

```
==== Тестирование завершено ====
```

[Вернуться к оглавлению](#)

### Задание №3

В классе **TabulatedFunctions** добавим ещё три перегруженных версии метода **createTabulatedFunction()**. Их параметры должны повторять параметры трёх аналогичных методов, основанных на использовании фабрики, но также эти методы должны получать ссылку типа **Class** на описание класса, объект которого требуется создать. Сделаем так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс **TabulatedFunction**.

Новые методы создания объектов будут находить в предложенном классе конструктор с соответствующими типами параметров. С помощью найденного конструктора будет создан объект табулированной функции. Ссылка на этот объект будет возвращена из метода создания.

Если в ходе выполнения рефлексивных операций возникло исключение (не найден конструктор и т.д.), оно будет отловлено. Вместо него будет выброшено исключение **IllegalArgumentException**.

В классе **TabulatedFunctions** перегрузим методы, создающие объекты табулированных функций, добавив версии, принимающие также ссылку типа **Class** на описание класса, объект которого требуется создать. Сделаем так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс **TabulatedFunction**.

```
// ----- Задание №3: Методы с рефлексией -----  
  
    public static TabulatedFunction createTabulatedFunction(Class<?>  
functionClass, double leftX, double rightX, int pointsCount){  
        // Проверяем, что класс реализует TabulatedFunciton  
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)){  
            throw new IllegalArgumentException("Класс " +  
functionClass.getName() + " не реализует интерфейс TabulatedFunction");  
        }  
  
        try {  
            // Ищем конструктор с параметрами (double, double, int)  
            Constructor<?> constructor =  
functionClass.getConstructor(double.class, double.class, int.class);  
  
            // Создаем объект с помощью рефлексии  
            return (TabulatedFunction) constructor.newInstance(leftX, rightX,  
pointsCount);  
  
        } catch (NoSuchMethodException e) {  
            throw new IllegalArgumentException("Не найден конструктор  
(double, double, int) в классе " +  
functionClass.getName(), e);  
        } catch (Exception e) {  
            throw new IllegalArgumentException("Ошибка при создании объекта "  
+ functionClass.getName(), e);  
        }  
    }  
  
    public static TabulatedFunction createTabulatedFunction(Class<?>  
functionClass, double leftX, double rightX, double[] values) {  
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {  
            throw new IllegalArgumentException("Класс " +  
functionClass.getName() +  
                    " не реализует интерфейс  
TabulatedFunction");  
        }  
  
        try {  
            // Ищем конструктор с параметрами (double, double, double[])
```

```
        Constructor<?> constructor =
functionClass.getConstructor(double.class, double.class, double[].class);

        return (TabulatedFunction) constructor.newInstance(leftX, rightX,
values);

    } catch (NoSuchMethodException e) {
        throw new IllegalArgumentException("Не найден конструктор
(double, double, double[]) в классе " +
functionClass.getName(), e);
    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта "
+ functionClass.getName(), e);
    }
}

public static TabulatedFunction createTabulatedFunction(Class<?>
functionClass, FunctionPoint[] points) {
    if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Класс " +
functionClass.getName() +
" не реализует интерфейс
TabulatedFunction");
    }

    try {
        // Ищем конструктор с параметрами (FunctionPoint[])
        Constructor<?> constructor =
functionClass.getConstructor(FunctionPoint[].class);

        return (TabulatedFunction) constructor.newInstance((Object)
points);

    } catch (NoSuchMethodException e) {
        throw new IllegalArgumentException("Не найден конструктор
(FunctionPoint[]) в классе " +
functionClass.getName(), e);
    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта "
+ functionClass.getName(), e);
    }
}
```

```
    }

    // Метод создания объекта с помощью рефлексии
    public static TabulatedFunction tabulate(Class<?> functionClass, Function
function,
                                              double leftX, double rightX, int
pointsCount) {
        // Проверка, что отрезок табулирования находится в области
        // определения функции
        if (leftX < function.getLeftDomainBorder() || rightX >
function.getRightDomainBorder()) {
            throw new IllegalArgumentException(
                "Отрезок табулирования [" + leftX + ", " + rightX + "] " +
                "выходит за область определения функции [" +
                function.getLeftDomainBorder() + ", " +
                function.getRightDomainBorder() + "]");
        }

        // Проверка на область определения
        if (leftX >= rightX) {
            throw new IllegalArgumentException("Левая граница (" + leftX + ")")
должна быть меньше правой (" + rightX + ")");
        }

        // Проверка на кол-во точек
        if (pointsCount < 2) {
            throw new IllegalArgumentException("Кол-во точек должно быть не
меньше двух");
        }

        // Создаем массив значений функции
        double[] values = new double[pointsCount];
        double step = (rightX - leftX) / (pointsCount - 1);

        // Заполняем массив значениями
        for (int i = 0; i < pointsCount; ++i) {
            double x = leftX + i * step;
            values[i] = function.getFunctionValue(x);
        }
    }
}
```

```
// Используем рефлексивное создание
return createTabulatedFunction(functionClass, leftX, rightX, values);
}
```

Проверим в методе **main()** работу методов рефлексивного создания объектов, а также методов класса **TabulatedFunctions**, использующих создание объектов.

```
public static void testReflection() {
    System.out.println("== Тестирование рефлексивного создания объектов
==");

    TabulatedFunction f;

    // Тест 1: Создание ArrayTabulatedFunction через границы и количество
    точек
    System.out.println("\n1. ArrayTabulatedFunction через границы и
количество точек:");
    f = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, 0, 10, 3);
    System.out.println("    Тип: " + f.getClass().getSimpleName());
    System.out.println("    Функция: " + f);

    // Тест 2: Создание ArrayTabulatedFunction через границы и значения
    System.out.println("\n2. ArrayTabulatedFunction через границы и
значения:");
    f = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, 0, 10, new double[] {0, 5, 10});
    System.out.println("    Тип: " + f.getClass().getSimpleName());
    System.out.println("    Функция: " + f);

    // Тест 3: Создание LinkedListTabulatedFunction через массив точек
    System.out.println("\n3. LinkedListTabulatedFunction через массив
точек:");
    f = TabulatedFunctions.createTabulatedFunction(
        LinkedListTabulatedFunction.class,
        new FunctionPoint[] {
            new FunctionPoint(0, 0),
            new FunctionPoint(5, 25),
```

```

        new FunctionPoint(10, 100)
    }
);

System.out.println("    Тип: " + f.getClass().getSimpleName());
System.out.println("    Функция: " + f);

// Тест 4: Табулирование с рефлексивным созданием
System.out.println("\n4. Табулирование Sin с
LinkedListTabulatedFunction:");

f = TabulatedFunctions.tabulate(
    LinkedListTabulatedFunction.class, new Sin(), 0, Math.PI, 5);
System.out.println("    Тип: " + f.getClass().getSimpleName());
System.out.println("    Функция: " + f);

// Тест 5: Ошибочный случай - класс не реализует TabulatedFunction
System.out.println("\n5. Тест ошибки (класс не реализует
TabulatedFunction):");

try {
    f = TabulatedFunctions.createTabulatedFunction(
        String.class, 0, 10, 3);
} catch (IllegalArgumentException e) {
    System.out.println("    Ожидаемая ошибка: " + e.getMessage());
}

System.out.println("\n==== Тестирование рефлексии завершено ====");
}

```

Вывод в консоли:

==== Тестирование рефлексивного создания объектов ====

1. ArrayTabulatedFunction через границы и количество точек:

Тип: ArrayTabulatedFunction

Функция: {(0.0, 0.0), (5.0, 0.0), (10.0, 0.0)}

2. ArrayTabulatedFunction через границы и значения:

Тип: ArrayTabulatedFunction

Функция: {(0.0, 0.0), (5.0, 5.0), (10.0, 10.0)}

3. LinkedListTabulatedFunction через массив точек:

Тип: LinkedListTabulatedFunction

Функция: {(0.0, 0.0), (5.0, 25.0), (10.0, 100.0)}

4. Табулирование Sin с LinkedListTabulatedFunction:

Тип: LinkedListTabulatedFunction

Функция: {(0.0, 0.0), (0.7853981633974483, 0.7071067811865475), (1.5707963267948966, 1.0),  
(2.356194490192345, 0.7071067811865476), (3.141592653589793, 1.2246467991473532E-16)}

5. Тест ошибки (класс не реализует TabulatedFunction):

Ожидаемая ошибка: Класс java.lang.String не реализует интерфейс TabulatedFunction

==== Тестирование рефлексии завершено ===

[Вернуться к оглавлению](#)

## Доработка лабораторной

По запросу преподавателя для методов чтения добавим реализацию через рефлексию.

```
// Ввод табулированной функции из байтового потока с использованием рефлексии
    public static TabulatedFunction inputTabulatedFunction(Class<?>
functionClass, InputStream in) throws IOException {
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)){
            throw new IllegalArgumentException("Класс " +
functionClass.getName() +
            " не реализует интерфейс TabulatedFunction");
        }

        DataInputStream dataIn = new DataInputStream(in);
        // Читаем кол-во точек
        int pointsCount = dataIn.readInt();

        // Читаем координаты точек
        double[] xValues = new double[pointsCount];
        double[] yValues = new double[pointsCount];

        for (int i = 0; i < pointsCount; ++i) {
            xValues[i] = dataIn.readDouble();
            yValues[i] = dataIn.readDouble();
        }

        // Используем рефлексивное создание
        return createTabulatedFunction(functionClass, xValues, yValues);
    }
```

```

    // Чтение табулированной функции из символьного потока с использованием
    рефлексии

    public static TabulatedFunction readTabulatedFunction(Class<?>
functionClass, Reader in) throws IOException {
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
            throw new IllegalArgumentException("Класс " + functionClass.getName()
+
                    " не реализует интерфейс
TabulatedFunction");
        }

        StreamTokenizer tokenizer = new StreamTokenizer(in);

        // Читаем кол-во точек
        tokenizer.nextToken();
        int pointsCount = (int) tokenizer.nval;

        // Читаем координаты точек
        double[] xValues = new double[pointsCount];
        double[] yValues = new double[pointsCount];

        for (int i = 0; i < pointsCount; i++) {
            tokenizer.nextToken(); // x координата
            xValues[i] = tokenizer.nval;

            tokenizer.nextToken(); // y координата
            yValues[i] = tokenizer.nval;
        }

        // Используем рефлексивное создание
        return createTabulatedFunction(functionClass, xValues, yValues);
    }
}

```

```

// Вспомогательный метод для создания из массивов x и у через рефлексию
    private static TabulatedFunction createTabulatedFunction(Class<?>
functionClass,
                                                               double[] xValues,
double[] yValues) {
    // Проверяем, что массивы одинаковой длины
    if (xValues.length != yValues.length) {

```

```

        throw new IllegalArgumentException("Массивы x и y должны быть
одинаковой длины");
    }

    if (xValues.length < 2) {
        throw new IllegalArgumentException("Количество точек должно быть
не меньше 2");
    }

    // Проверяем упорядоченность x
    for (int i = 1; i < xValues.length; i++) {
        if (xValues[i] <= xValues[i - 1]) {
            throw new IllegalArgumentException("Точки должны быть
упорядочены по возрастанию x");
        }
    }

    // Создаем массив точек и используем рефлексивное создание
    FunctionPoint[] points = new FunctionPoint[xValues.length];
    for (int i = 0; i < xValues.length; i++) {
        points[i] = new FunctionPoint(xValues[i], yValues[i]);
    }
    return createTabulatedFunction(functionClass, points);
}

```

Проверим работу в main():

```

public static void testReflectionIO() {
    System.out.println("==== Тестирование рефлексивного чтения объектов
====");

    try {
        // Тест 1: Запись в байтовый поток и чтение через рефлексию
        System.out.println("\n1. Тест байтового потока:");

        // Создаем тестовую функцию и записываем в поток
        TabulatedFunction original = new ArrayTabulatedFunction(0, 10,
new double[][]{0, 5, 10});
        ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
        TabulatedFunctions.outputTabulatedFunction(original, byteOut);
    }
}

```

```

    // Читаем через рефлексию как LinkedListTabulatedFunction
    ByteArrayInputStream byteIn = new
ByteArrayInputStream(byteOut.toByteArray());
    TabulatedFunction readFunction =
TabulatedFunctions.inputTabulatedFunction(
        LinkedListTabulatedFunction.class, byteIn);

        System.out.println("    Оригинал: " +
original.getClass().getSimpleName());
        System.out.println("    Прочитано: " +
readFunction.getClass().getSimpleName());
        System.out.println("    Данные: " + readFunction);

    // Тест 2: Запись в символьный поток и чтение через рефлексию
    System.out.println("\n2. Тест символьного потока:");

    // Создаем другую функцию
    TabulatedFunction original2 = new LinkedListTabulatedFunction(
        new FunctionPoint[]{new FunctionPoint(1, 1), new
FunctionPoint(2, 4), new FunctionPoint(3, 9)}
    );
    StringWriter writer = new StringWriter();
    TabulatedFunctions.writeTabulatedFunction(original2, writer);

    // Читаем через рефлексию как ArrayTabulatedFunction
    StringReader reader = new StringReader(writer.toString());
    TabulatedFunction readFunction2 =
TabulatedFunctions.readTabulatedFunction(
        ArrayTabulatedFunction.class, reader);

        System.out.println("    Оригинал: " +
original2.getClass().getSimpleName());
        System.out.println("    Прочитано: " +
readFunction2.getClass().getSimpleName());
        System.out.println("    Данные: " + readFunction2);

    // Тест 3: Проверка корректности данных
    System.out.println("\n3. Проверка корректности данных:");
    boolean dataCorrect = true;
    for (int i = 0; i < original2.getPointsCount(); i++) {

```

```

        if (Math.abs(original2.getPointX(i) -
readFunction2.getPointX(i)) > 1e-10 ||
            Math.abs(original2.getPointY(i) -
readFunction2.getPointY(i)) > 1e-10) {
            dataCorrect = false;
            break;
        }
    }
    System.out.println("    Данные сохранены корректно: " +
dataCorrect);

} catch (IOException e) {
    System.out.println("    Ошибка ввода-вывода: " + e.getMessage());
} catch (Exception e) {
    System.out.println("    Неожиданная ошибка: " + e.getMessage());
}

System.out.println("\n==== Тестирование рефлексивного чтения завершено
====");
}

```

Вывод в консоли:

==== Тестирование рефлексивного чтения объектов ====

1. Тест байтового потока:

Оригинал: ArrayTabulatedFunction

Прочитано: LinkedListTabulatedFunction

Данные: {(0.0, 0.0), (5.0, 5.0), (10.0, 10.0)}

2. Тест символьного потока:

Оригинал: LinkedListTabulatedFunction

Прочитано: ArrayTabulatedFunction

Данные: {(1.0, 1.0), (2.0, 4.0), (3.0, 9.0)}

3. Проверка корректности данных:

Данные сохранены корректно: true

==== Тестирование рефлексивного чтения завершено ====