

# ЛАБОРАТОРНАЯ РАБОТА №7

Курс: Объектно-ориентированное  
программирование.

Студент: Кузнецов Эрик Витальевич  
Группа: 6204-010302D  
Преподаватель: Борисов Дмитрий Сергеевич

# Задание 1

Ход выполнения работы:

1)Расширен интерфейс TabulatedFunction для поддержки итераторов

2)Реализованы итераторы в ArrayTabulatedFunction

3)Реализованы итераторы в LinkedListTabulatedFunction

4)Протестирована работа итераторов в методе main

```
Users > user > OneDrive > Рабочий стол > Lab7 > functions > J TabulatedFunction.java
package functions;
import java.util.Iterator;

public interface TabulatedFunction extends Function, Iterable<FunctionPoint>, Cloneable {

    // Методы работы с точками
    int getPointsCount(); // Метод, возвращающий количество точек
    FunctionPoint getPoint(int index); // Метод, возвращающий ссылку на объект по индексу
    void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException; // Метод, заменяющий точку по индексу на заданную
    double getPointX(int index); // Возвращает X точки по указанному индексом
    void setPointX(int index, double x) throws InappropriateFunctionPointException; // Метод, изменяющий абсциссу точки по индексу
    double getPointY(int index); // Возвращает Y точки по указанному индексом
    void setPointY(int index, double y); // Метод, изменяющий ординату точки по индексу

    // Методы изменения количества точек
    void deletePoint(int index); // Удаление точки по индексу
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException; // Добавление точки

    // Метод вывода
    void printTabulatedFunction(); // Вывод в консоль

    Object clone();
}

// Методы интерфейса Function
@Override//№1
public Iterator<FunctionPoint> iterator() {
    // Создаем анонимный класс итератора
    return new Iterator<FunctionPoint>() {
        private int currentIndex = 0;

        // Проверяет, есть ли следующий элемент
        @Override
        public boolean hasNext() {
            return currentIndex < pointsCount;
        }

        // Возвращает следующий элемент
        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new NoSuchElementException("Нет следующего элемента");
            }
            // Возвращаем копию точки для защиты инкапсуляции
            return new FunctionPoint(points[currentIndex++]);
        }

        // Удаление не поддерживается - всегда бросаем исключение
        @Override
        public void remove() {
            throw new UnsupportedOperationException("Удаление не поддерживается");
        }
    };
}
```

```

@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.next; // Начинаем с первого элемента
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentNode != head && currentIndex < size;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new NoSuchElementException("Нет следующего элемента");
            }
            FunctionPoint point = new FunctionPoint(currentNode.point); // Копируем точку
            currentNode = currentNode.next; // Переходим к следующему узлу
            currentIndex++;
            return point;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Удаление не поддерживается");
        }
    };
}

```

#### ЗАДАНИЕ 1 - ТЕСТИРОВАНИЕ ИТЕРАТОРОВ

---

1. `ArrayTabulatedFunction` с `for-each` циклом:

Все точки функции через `for-each`:

- (0.0; 1.0)
- (2.0; 5.0)
- (4.0; 9.0)
- (6.0; 13.0)
- (8.0; 17.0)
- (10.0; 21.0)

2. `LinkedListTabulatedFunction` с `for-each` циклом:

Все точки функции через `for-each`:

- (0.0; 0.0)
- (2.0; 4.0)
- (4.0; 16.0)
- (6.0; 36.0)
- (8.0; 64.0)
- (10.0; 100.0)

3. Тестирование исключений итераторов:

a) `NoSuchElementException`:  
Поймано: Нет следующего элемента  
b) `UnsupportedOperationException`:  
Поймано: Удаление не поддерживается

4. Сравнение итератора с обычным циклом:

`ArrayTabulatedFunction` - обычный цикл:

- (0.0; 1.0)
- (2.0; 5.0)
- (4.0; 9.0)
- (6.0; 13.0)
- (8.0; 17.0)
- (10.0; 21.0)

`ArrayTabulatedFunction` - `for-each` цикл:

- (0.0; 1.0)
- (2.0; 5.0)
- (4.0; 9.0)
- (6.0; 13.0)
- (8.0; 17.0)
- (10.0; 21.0)

5. Проверка защиты инкапсуляции:

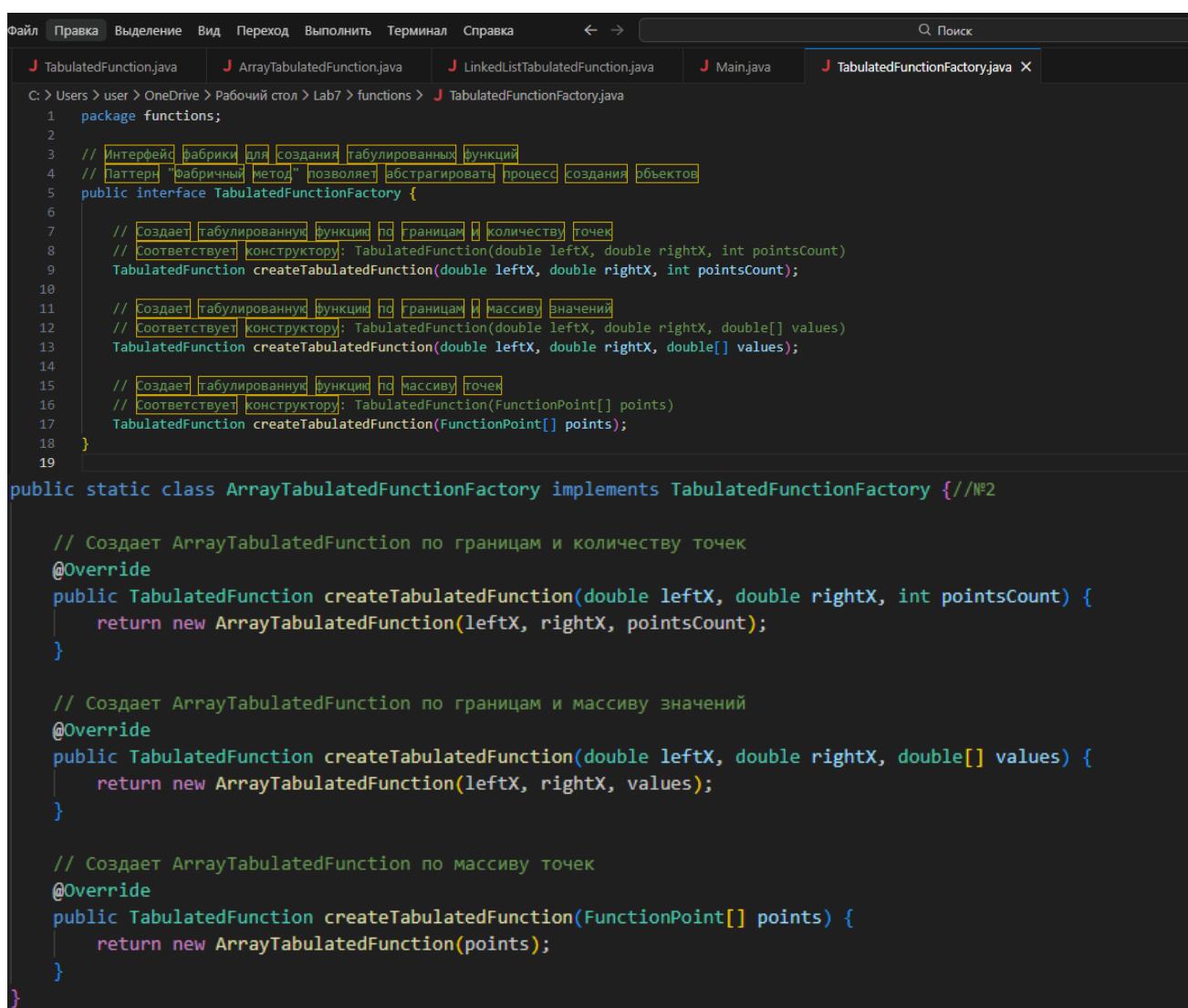
Получаем точку через итератор и пытаемся изменить:  
Инкапсуляция защищена: изменение копии не затронуло оригинал  
Оригинал: 1.0, Копия: 999.0

Рисунок 1-4-изображение с конечным результатом для задания 1.

## Задание 2.

Ход выполнения работы:

- 1) Создан интерфейс фабрики
- 2) Реализованы фабрики для каждого типа функций
- 3) Добавлено управление фабриками в TabulatedFunctions
- 4) Протестирована работа фабрик



The screenshot shows a Java code editor with the following tabs at the top: TabulatedFunction.java, ArrayTabulatedFunction.java, LinkedListTabulatedFunction.java, Main.java, and TabulatedFunctionFactory.java. The current file is TabulatedFunctionFactory.java. The code implements the TabulatedFunctionFactory interface, which provides methods for creating tabulated functions based on different criteria: boundaries and point count, boundaries and values array, or points array. It also implements the ArrayTabulatedFunctionFactory interface, providing implementations for each of the three creation methods.

```
Файл Правка Выделение Вид Переход Выполнить Терминал Справка ← → Q Поиск
J TabulatedFunction.java J ArrayTabulatedFunction.java J LinkedListTabulatedFunction.java J Main.java J TabulatedFunctionFactory.java ×
C: > Users > user > OneDrive > Рабочий стол > Lab7 > functions > J TabulatedFunctionFactory.java
1 package functions;
2
3 // Интерфейс фабрики для создания табулированных функций
4 // Паттерн Фабричный метод позволяет абстрагировать процесс создания объектов
5 public interface TabulatedFunctionFactory {
6
7     // Создает табулированную функцию по границам и количеству точек
8     // Соответствует конструктору: TabulatedFunction(double leftX, double rightX, int pointsCount)
9     TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount);
10
11    // Создает табулированную функцию по границам и массиву значений
12    // Соответствует конструктору: TabulatedFunction(double leftX, double rightX, double[] values)
13    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values);
14
15    // Создает табулированную функцию по массиву точек
16    // Соответствует конструктору: TabulatedFunction(FunctionPoint[] points)
17    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);
18 }
19
public static class ArrayTabulatedFunctionFactory implements TabulatedFunctionFactory { // №2
20
21     // Создает ArrayTabulatedFunction по границам и количеству точек
22     @Override
23     public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
24         return new ArrayTabulatedFunction(leftX, rightX, pointsCount);
25     }
26
27     // Создает ArrayTabulatedFunction по границам и массиву значений
28     @Override
29     public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
30         return new ArrayTabulatedFunction(leftX, rightX, values);
31     }
32
33     // Создает ArrayTabulatedFunction по массиву точек
34     @Override
35     public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
36         return new ArrayTabulatedFunction(points);
37     }
38 }
```

```
// Вложенный публичный класс фабрики для LinkedListTabulatedFunction
public static class LinkedListTabulatedFunctionFactory implements TabulatedFunctionFactory { //№2

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

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

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

#### ЗАДАНИЕ 2 - ТЕСТИРОВАНИЕ ФАБРИК

---

1. Фабрика по умолчанию:  
Тип созданного объекта: ArrayTabulatedFunction
2. Устанавливаем LinkedListTabulatedFunctionFactory:  
Тип созданного объекта: LinkedListTabulatedFunction
3. Прямое использование фабричных методов (должно создавать LinkedList):  
Создана функция с массивом значений: LinkedListTabulatedFunction  
Создана функция с массивом точек: LinkedListTabulatedFunction
4. Возвращаем ArrayTabulatedFunctionFactory:  
Тип созданного объекта: ArrayTabulatedFunction
5. Прямое использование фабричных методов:  
Создана функция с массивом значений: ArrayTabulatedFunction  
Создана функция с массивом точек: ArrayTabulatedFunction
6. Демонстрация полиморфизма:  
Все созданные функции поддерживают for-each:  
(0.0; 0.0)  
(1.0; 1.0)  
(2.0; 8.0)

Рисунок 5-8-Снимок Экрана с конечным результатом для задания 2

### Задание 3.

Ход выполнения работы:

- 1)Добавлены методы создания через рефлексию в TabulatedFunctions
- 2)Реализованы аналогичные методы для других конструкторов
- 3)Добавлен рефлексивный метод табулирования
- 4)Протестирована работа рефлексии

```
public static TabulatedFunction createTabulatedFunction(
    Class<?> functionClass, double leftX, double rightX, int pointsCount) {

    // Проверяем, что класс реализует TabulatedFunction
    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 через рефлексию
        return (TabulatedFunction) constructor.newInstance(leftX, rightX, pointsCount);
    } catch (NoSuchMethodException e) {
        throw new IllegalArgumentException(
            "Класс " + functionClass.getName() + " не имеет конструктора (double, double, int)", e);
    } catch (InstantiationException | IllegalAccessException | InvocationTargetException 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(
            "Класс " + functionClass.getName() + " не имеет конструктора (double, double, double[])", e);
    } catch (InstantiationException | IllegalAccessException | InvocationTargetException 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(
            "Класс " + functionClass.getName() + " не имеет конструктора (FunctionPoint[])", e);
    } catch (InstantiationException | IllegalAccessException | InvocationTargetException e) {
        throw new IllegalArgumentException(
            "Ошибка при создании объекта " + functionClass.getName(), e);
    }
}

*/
public static TabulatedFunction tabulate(
    Class<?> functionClass, Function function, double leftX, double rightX, int pointsCount) {

    if (leftX < function.getLeftDomainBorder() || rightX > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы табулирования выходят за область определения функции");
    }
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Количество точек должно быть не менее 2");
    }

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

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

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

```

==== ТЕСТИРОВАНИЕ ФУНКИЙ ЗАДАЧЕЙ № 3 =====

**ЗАДАНИЕ 3 - ТЕСТИРОВАНИЕ РЕФЛЕКСИИ**

=====

1. Создание ArrayTabulatedFunction через рефлексию:  
Тип созданного объекта: ArrayTabulatedFunction  
Функция:  $\{(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)\}$
2. Создание ArrayTabulatedFunction с массивом значений:  
Тип созданного объекта: ArrayTabulatedFunction  
Функция:  $\{(0.0; 0.0), (1.0; 1.0), (2.0; 4.0), (3.0; 9.0), (4.0; 16.0)\}$
3. Создание LinkedListTabulatedFunction через рефлексию:  
Тип созданного объекта: LinkedListTabulatedFunction  
Функция:  $\{(0.0; 0.0), (1.0; 1.0), (2.0; 8.0)\}$
4. Табулирование функции Sin с использованием рефлексии:  
Тип созданного объекта: LinkedListTabulatedFunction  
Первые 5 точек функции Sin:  
 $(0.0; 0.0)$   
 $(0.3141592653589793; 0.3090169943749474)$   
 $(0.6283185307179586; 0.5877852522942731)$   
 $(0.9424777960769379; 0.8090169943749475)$   
 $(1.2566370614359172; 0.9510565162951535)$
5. Тестирование обработки ошибок:  
а) Передача неправильного класса:  
Поймано исключение: Класс java.lang.String не реализует интерфейс TabulatedFunction  
б) Передача несуществующего класса:  
Поймано исключение: Класс не найден

Рисунок 9-13-Снимок Экрана с конечным результатом для задания 3