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)

ЛАБОРАТОРНАЯ РАБОТА № 3

«Разработка реализаций табулированной функции с единым интерфейсом и системой исключений»

по курсу
Объектно-ориентированное программирование

Выполнила: Элибекян
Анжелина, 6203-010302

Оглавление

Задание №1

Я изучила основные классы исключений Java, которые понадобятся для реализации безопасной работы с табулированными функциями.

Задание №2

Создаю два специализированных класса исключений. FunctionPointIndexOutOfBoundsException - это исключение я использую для случаев, когда программа пытается обратиться к несуществующей точке по индексу :

```
public class FunctionPointIndexOutOfBoundsException extends IndexOutOfBoundsException {

    // конструктор по умолчанию
    // создает исключение без сообщения об ошибке
    public FunctionPointIndexOutOfBoundsException(){
        super();
    }

    // конструктор с сообщением об ошибке
    // создает исключение с указанным сообщением, которое описывает причину ошибки
    public FunctionPointIndexOutOfBoundsException(String message){
        super(message);
    }
}
```

InappropriateFunctionPointException - это исключение я применяла при нарушении порядка точек или попытке добавить точку с уже существующей координатой x:

```
InappropriateFunctionPointException extends Exception{

    // конструктор по умолчанию
    // создает исключение без сообщения об ошибке
    public InappropriateFunctionPointException(){
        super();
    }

    // конструктор с сообщением об ошибке
    // создает исключение с указанным сообщением, которое описывает причину ошибки
    public InappropriateFunctionPointException(String message){
        super(message);
    }
}
```

Задание №3

В этом задании я модифицировала класс табулированной функции, добавив проверки и исключения для обеспечения безопасности данных.
Я добавляю проверки в конструкторы, чтобы предотвратить создание функции с некорректными параметрами:

```
public ArrayTabulatedFunction(double leftX, double rightX, int pointsCount) {
    // проверка корректности входных параметров
    if (leftX >= rightX) {
        throw new IllegalArgumentException("Левая граница должна быть меньше правой");
    }
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Количество точек должно быть не менее 2");
    }
```

Также я реализовала приватный метод checkIndex(), который проверяет валидность индекса перед доступом к точке:

```
private void checkIndex(int index) {
    if (index < 0 || index >= count) {
        throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " выходит за
границы [0, " + (count - 1) + "]");
    }
}
```

Для упорядоченности точек по координате x я создаю метод checkOrder():

```
private void checkOrder(int index, double newX) throws InappropriateFunctionPointException {
    if (index > 0 && newX <= points[index - 1].getX()) {
        throw new InappropriateFunctionPointException("Координата x должна быть больше
предыдущей точки");
    }
    if (index < count - 1 && newX >= points[index + 1].getX()) {
        throw new InappropriateFunctionPointException("Координата x должна быть меньше
следующей точки");
    }
}
```

Данный метод я применяю в setPoint() и setPointX(), чтобы гарантировать, что любое изменение координаты x не нарушит упорядоченность точек.
Особое внимание я уделила методу deletePoint(), так как удаление точек может привести к некорректному состоянию функии. В этом методе я использую IllegalStateException, потому что ошибка связана с текущим состоянием объекта. Эта проверка необходима, т.к. Функция с менее чем тремя точками не может обеспечить корректную интерполяцию.

```
public void deletePoint(int index) {
    // проверка корректности индекса
    checkIndex(index);
    // проверка, что после удаления останется минимум 2 точки
    if (count < 3) {
        throw new IllegalStateException("Невозможно удалить точку: количество точек должно
быть не менее 3");
```

```
    }

    // сдвиг элементов массива влево, начиная с позиции после удаляемой точки
    for (int i = index; i < count - 1; i++) {
        points[i] = points[i + 1];
    }
    // очистка последней позиции и уменьшение счетчика
    points[count - 1] = null;
    count--;
}
```

Далее, метод addPoint() включает комплексную проверку на уникальность координат x. Здесь же я использую метод equals() для сравнения вещественных чисел с учетом погрешности, что является достаточно важным аспектом при работе с числами с плавающей точкой.

```
// добавляем новую точку в табулированную функцию
public void addPoint(FunctionPoint point) throws InappropriateFunctionPointException {
    // поиск позиции для вставки (точки должны быть упорядочены по x)
    int insertIndex = 0;
    while (insertIndex < count && points[insertIndex].getX() < point.getX()) {
        insertIndex++;
    }

    // проверка, что точка с таким x еще не существует
    if (insertIndex < count && equals(points[insertIndex].getX(), point.getX())) {
        throw new InappropriateFunctionPointException("Точка с координатой x = " + point.getX()
+ " уже существует");
    }
```

Задание №4

Реализация LinkedListTabulatedFunction.

Для реализации альтернативной версии табулированной функции я выбрала двусвязный циклический список с выделенной головой. Эта структура данных обеспечивает эффективное выполнение операций вставки и удаления, что является преимуществом по сравнению с массивом. Я начала с создания внутреннего класса для узлов списка, тщательно продумав его инкапсуляцию:

```java
// внутренний класс для узла двусвязного списка
private static class FunctionNode {  20 usages
    private FunctionPoint point;    // данные точки, хранящейся в узле  2
    private FunctionNode prev;      // ссылка на предыдущий узел в списке
    private FunctionNode next;      // ссылка на следующий узел в списке

    // конструктор с точкой
    // создает узел с заданной точкой и null-ссылками
    public FunctionNode(FunctionPoint point) { this.point = point; }
}
```

Я реализовала метод initializeList() для создания пустого списка в корректном состоянии. Выделенная голова - это специальный узел, который не содержит полезных данных, но служит якорем для списка. В пустом списке голова ссылается сама на себя, что обеспечивает корректность операций даже при отсутствии реальных данных.

```java
// инициализация пустого циклического списка с головой
// создаем служебный узел head, который ссылается сам на себя
// в пустом списке head.next и head.prev указывают на head
private void initializeList() {  2 usages
    head = new FunctionNode( point: null);
    head.next = head;
    head.prev = head;
    count = 0;
}
```

Метод getNode() обеспечивает доступ к узлам списка по индексу:

```java
// получаем узел по индексу
// последовательно проходим по списку от головы до нужного узла
private FunctionNode getNode(int index) {  8 usages
    // проверка корректности индекса
    checkIndex(index);
    // начинаем с первого реального узла (после головы)
    FunctionNode current = head.next;
    // последовательно переходим к следующему узлу index раз
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current;
}
```

Далее я реализовала два метода для вставки узлов: addNodeToTail() для добавления в конец и addNodeByIndex() для вставки в произвольную позицию.

```
// добавляем узел в конец списка
// создаем новый узел и вставляем его перед головой (в циклическом списке)
private FunctionNode addNodeToTail() {  1 usage
    // создание нового узла, который будет вставлен между последним узлом и головой
    FunctionNode newNode = new FunctionNode( point: null);
    // установка ссылок нового узла
    newNode.prev = head.prev;
    newNode.next = head;
    // обновление ссылок соседних узлов
    head.prev.next = newNode;
    head.prev = newNode;
    // увеличение счетчика точек
    count++;

    // возвращаем ссылку на созданный узел
    return newNode;
}
```

Этот метод демонстрирует преимущества циклического списка - добавление в конец выполняется за константное время O(1), так как у нас есть непосредственный доступ к последнему узлу через head.prev.
Метод addNodeByIndex() более сложный, так как требует нахождения позиции для вставки:

```
// добавляем узел по указанному индексу
// вставляем новый узел перед узлом с заданным индексом
private FunctionNode addNodeByIndex(int index) {  1 usage
    // проверка корректности индекса (допустимы значения от 0 до count)
    if (index < 0 || index > count) {
        throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " выходит за границы [0, " + count + "]");
    }

    // если индекс равен количеству точек, добавляем в конец
    if (index == count) {
        return addNodeToTail();
    }

    // получение узла, перед которым нужно вставить новый узел
    FunctionNode nodeAtIndex = getNode(index);
    // создание нового узла
    FunctionNode newNode = new FunctionNode( point: null);
    // установка ссылок нового узла
    newNode.prev = nodeAtIndex.prev;
    newNode.next = nodeAtIndex;
    // обновление ссылок соседних узлов
    nodeAtIndex.prev.next = newNode;
    nodeAtIndex.prev = newNode;
    // увеличение счетчика точек
    count++;
    return newNode;
}
```

Особенность этой реализации в том, что она эффективно обрабатывает пограничные случаи - вставку в начало, конец и середину списка.
Также имеем метод deleteNodeByIndex(), который обеспечивает корректное удаление узлов с обновлением ссылок соседних элемеентов:

```java
// удаляем узел по указанному индексу
// извлекаем узел из списка и обновляем ссылки соседних узлов
private void deleteNodeByIndex(int index) {  1 usage
    // проверка корректности индекса
    checkIndex(index);
    // проверка, что после удаления останется минимум 2 точки
    if (count < 3) {
        throw new IllegalStateException("Невозможно удалить точку: количество точек должно быть не менее 3");
    }

    // получение узла для удаления
    FunctionNode nodeToDelete = getNode(index);
    // обход удаляемого узла: предыдущий узел ссылается на следующий и наоборот
    nodeToDelete.prev.next = nodeToDelete.next;
    nodeToDelete.next.prev = nodeToDelete.prev;
    // уменьшение счетчика точек
    count--;
}
```

Данная реализация довольно таки проста, создаем всего лишь две операции по обновлению ссылок, которые в свою очередь изолируют удаляемый узел от списка.

Задание №5

В этом задании я реализовала полный набор методов интерфейса табулированной функции для связного списка, обеспечив полную совместимость с массивной реализацией. Я создала конструкторы, аналогичные массивой реализации, но адаптированные для работы со связным списком. Особенность этой реализации в том, что я использую метод addPoint() для создания точек, что гарантирует соблюдение всех инвариантов и упорядоченности с самого начала.

```java
// конструктор для создания функции с равномерной сеткой и заданными значениями y
public LinkedListTabulatedFunction(double leftX, double rightX, double[] values) {  22 usages
    // проверка корректности входных параметров
    if (leftX >= rightX) {
        throw new IllegalArgumentException("Левая граница должна быть меньше правой");
    }
    if (values.length < 2) {
        throw new IllegalArgumentException("Количество точек должно быть не менее 2");
    }

    // инициализация пустого списка
    initializeList();
    // вычисление шага между соседними точками
    double step = (rightX - leftX) / (values.length - 1);
    // создание точек с координатами x и значениями y из массива values
    for (int i = 0; i < values.length; i++) {
        double x = leftX + i * step;
        // добавление точки через метод addPoint для упрощения
        try {
            addPoint(new FunctionPoint(x, values[i]));
        } catch (InappropriateFunctionPointException e) {
            // не должно происходить, так как точки создаются с уникальными x
        }
    }
}
```

Реализация метода getFunctionValue() для связного списка требует последовательного прохода по узлам. Этот метод демонстрирует преимущество упорядоченности точек - мы можем прекратить поиск, как только найдем интервал, содержащий x. Линейная интерполяция выполняется по стандартной формуле, обеспечивая плавность функции между точками.

```
// вычисляем значение функции в точке x с помощью линейной интерполяции
public double getFunctionValue(double x) {  3 usages
    // проверка, находится ли x в области определения функции
    if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
        return Double.NaN;
    }

    // поиск интервала, в котором находится x
    // начинаем с первого реального узла
    FunctionNode current = head.next;
    while (current != head) {
        FunctionNode next = current.next;
        // если next не голова и x находится в текущем интервале [current.x, next.x]
        if (next != head && x >= current.point.getX() && x <= next.point.getX()) {
            double x1 = current.point.getX();
            double x2 = next.point.getX();
            double y1 = current.point.getY();
            double y2 = next.point.getY();
            // линейная интерполяция
            return y1 + (y2 - y1) * (x - x1) / (x2 - x1);
        }
        current = current.next;
    }

    return Double.NaN;
}
```

Также, я реализовала полный набор методов для работы с точками, используя ранее созданные методы работы со списком. Важной особенностью является то, что метод getPoint() возвращает копию точки, а не оригинал. Это предотвращает несанкционированное изменение внутреннего состояния функции.

```java
// получаем точку по указанному индексу
public FunctionPoint getPoint(int index) {  3 usages
    // получение узла по индексу и возврат копии его точки
    return new FunctionPoint(getNode(index).point);
}

// устанавливаем новую точку по указанному индексу
public void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException {  2 usages
    // получение узла по индексу
    FunctionNode node = getNode(index);

    // проверка упорядоченности
    // если есть предыдущая точка, новая x должна быть больше ее x
    if (index > 0 && point.getX() <= node.prev.point.getX()) {
        throw new InappropriateFunctionPointException("Координата x должна быть больше предыдущей точки");
    }
    // если есть следующая точка, новая x должна быть меньше ее x
    if (index < count - 1 && point.getX() >= node.next.point.getX()) {
        throw new InappropriateFunctionPointException("Координата x должна быть меньше следующей точки");
    }

    // установка новой точки
    node.point = new FunctionPoint(point);
}
```

Реализация addPoint() для связного списка включает поиск позиции для вставки с сохранением упорядоченности. Этот метод эффективно использует упорядоченность списка для быстрого нахождения позиции вставки. Проверка на уникальность координаты x гарантирует целостность данных.

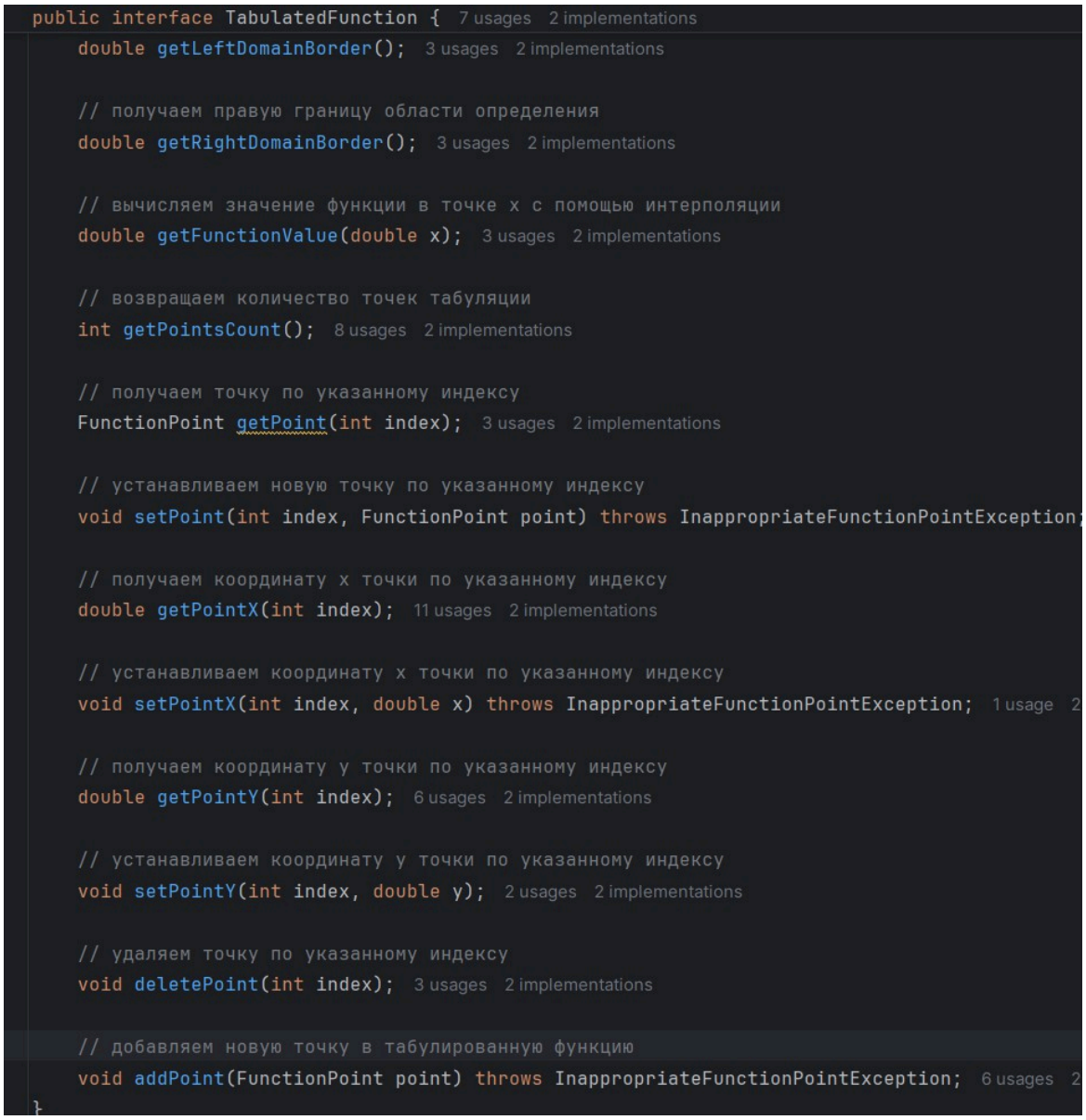
```java
// добавляем новую точку в табулированную функцию
public void addPoint(FunctionPoint point) throws InappropriateFunctionPointException {  6 usages
    // поиск позиции для вставки (точки должны быть упорядочены по x)
    int insertIndex = 0;
    // начинаем с первого реального узла
    FunctionNode current = head.next;
    // ищем позицию, где current.x >= point.x
    while (current != head && current.point.getX() < point.getX()) {
        current = current.next;
        insertIndex++;
    }

    // проверка, что точка с таким x еще не существует
    if (current != head && equals(current.point.getX(), point.getX())) {
        throw new InappropriateFunctionPointException("Точка с координатой x = " + point.getX() + " уже существует");
    }

    // вставка новой точки
    FunctionNode newNode = addNodeByIndex(insertIndex);
    newNode.point = new FunctionPoint(point);
}
```

Задание №6

Для обеспечения полиморфизма и возможности взаимозаменяемости реализаций я создала общий интерфейс TabulatedFunction.



```
public interface TabulatedFunction {  7 usages  2 implementations
    double getLeftDomainBorder();  3 usages  2 implementations

    // получаем правую границу области определения
    double getRightDomainBorder();  3 usages  2 implementations

    // вычисляем значение функции в точке x с помощью интерполяции
    double getFunctionValue(double x);  3 usages  2 implementations

    // возвращаем количество точек табуляции
    int getPointsCount();  8 usages  2 implementations

    // получаем точку по указанному индексу
    FunctionPoint getPoint(int index);  3 usages  2 implementations

    // устанавливаем новую точку по указанному индексу
    void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException;

    // получаем координату x точки по указанному индексу
    double getPointX(int index);  11 usages  2 implementations

    // устанавливаем координату x точки по указанному индексу
    void setPointX(int index, double x) throws InappropriateFunctionPointException;  1 usage  2

    // получаем координату y точки по указанному индексу
    double getPointY(int index);  6 usages  2 implementations

    // устанавливаем координату y точки по указанному индексу
    void setPointY(int index, double y);  2 usages  2 implementations

    // удаляем точку по указанному индексу
    void deletePoint(int index);  3 usages  2 implementations

    // добавляем новую точку в табулированную функцию
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;  6 usages  2
}
```

Я переименовала исходный класс TabulatedFunction в ArrayTabulatedFunction, что более точно отражает его реализацию. Такая структура обеспечивает четкое разделение ответственности и позволяет легко добавлять новые реализации в будущем.

```
public class ArrayTabulatedFunction implements TabulatedFunction {
    // реализация на основе массива
}

public class LinkedListTabulatedFunction implements TabulatedFunction {
    // реализация на основе связного списка
}
```

Задание №7

В ходе выполнения данного задания я провела комплексное тестирование разработанных реализаций табулированной функции. Основной целью было убедиться в корректности работы как массивой реализации ArrayTabulatedFunction, так и списковой реализации LinkedListTabulatedFunction, а также продемонстрировать принцип полиморфизма через использование единого интерфейса TabulatedFunction. В процессе работы я создала универсальную функцию testTabulatedFunction, которая тщательно проверяет все аспекты работы табулированных функций. Важнейшей частью тестирования стала проверка системы исключений. Для этого я создала отдельный метод testAllExceptions(), где последовательно проверяла все возможные ошибочные сценарии: некорректные параметры конструкторов, выход за границы индексов, нарушение порядка точек при модификации, попытки добавления точек с дублирующими координатами X и удаление точек при недостаточном их количестве.

Прикрепляю результат тестов:

```
==================================================
ТЕСТИРОВАНИЕ ArrayTabulatedFunction
==================================================

1. Создание функции y = x^2 для ArrayTabulatedFunction
Начальные точки функции:
(0,00, 0,00)
(1,00, 1,00)
(2,00, 4,00)
(3,00, 9,00)
(4,00, 16,00)

2. Интерполяция на исходной функции
f(0,50) = 0,50
f(1,20) = 1,60
f(2,50) = 6,50
f(3,20) = 10,40
f(6,80) = NaN

3. Область определения
Левая граница: 0,00
Правая граница: 4,00
Проверка точек вне области определения:
f(-1,00) = не определено
f(10,00) = не определено

4. Добавление точек
Добавляем точку (1.5, 2.25) - соответствует y = x²
Добавляем точку (3.5, 12.25) - соответствует y = x²
После добавления двух точек:
(0,00, 0,00)
(1,00, 1,00)
```

```
(1,50, 2,25)
(2,00, 4,00)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

5. Удаление точки
Удаляем точку с индексом 3
После удаления точки с индексом 3:
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

6. Изменение точки
Изменяем точку с индексом 2 на (3.0, 4.0)
Ошибка при изменении точки: Координата x должна быть меньше следующей точки
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

7. Финальная интерполяция
f(0,50) = 0,50
f(1,20) = 1,50
f(2,50) = 6,75
f(3,20) = 10,30
f(6,80) = NaN
```

```
8. Итоговое состояние функции
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

==================================================
ТЕСТИРОВАНИЕ LinkedListTabulatedFunction
==================================================

1. Создание функции y = x^2 для LinkedListTabulatedFunction
Начальные точки функции:
(0,00, 0,00)
(1,00, 1,00)
(2,00, 4,00)
(3,00, 9,00)
(4,00, 16,00)

2. Интерполяция на исходной функции
f(0,50) = 0,50
f(1,20) = 1,60
f(2,50) = 6,50
f(3,20) = 10,40
f(6,80) = NaN

3. Область определения
Левая граница: 0,00
Правая граница: 4,00
Проверка точек вне области определения:
f(-1,00) = не определено
f(10,00) = не определено
```

```
7. Финальная интерполяция
f(0,50) = 0,50
f(1,20) = 1,50
f(2,50) = 6,75
f(3,20) = 10,30
f(6,80) = NaN

8. Итоговое состояние функции
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

==================================================
ТЕСТИРОВАНИЕ ИСКЛЮЧЕНИЙ
==================================================

Тест 1: Некорректные параметры конструктора
Пытаемся создать функцию с leftX=10, rightX=0 (нарушение порядка)
:) IllegalArgumentException: Левая граница должна быть меньше правой
Пытаемся создать функцию с pointsCount=1 (слишком мало точек)
:) IllegalArgumentException: Количество точек должно быть не менее 2

Тест 2: Выход за границы индексов
Пытаемся получить точку с индексом -1
:) FunctionPointIndexOutOfBoundsException: Индекс -1 выходит за границы [0, 2]
Пытаемся получить точку с индексом 10 (вне границ)
:) FunctionPointIndexOutOfBoundsException: Индекс 10 выходит за границы [0, 2]
```