Отчет по лабораторной работе №3

Разработка иерархии классов для табулированных функций

Выполнила: Андреяшкина Мария
Группа: 6204-010302D

Оглавление

Задание 1

Создание интерфейса TabulatedFunction

Разработан интерфейс TabulatedFunction, определяющий основные операции для работы с табулированными функциями.

Основные методы интерфейса:

- getPointsCount() – получение количества точек
- getPoint(int index) – получение точки по индексу
- setPoint(int index, FunctionPoint point) – установка точки
- addPoint(FunctionPoint point) – добавление точки
- deletePoint(int index) – удаление точки
- getLeftDomainBorder(), getRightDomainBorder() – границы области определения
- getFunctionValue(double x) – значение функции в точке

Интерфейс обеспечивает единый контракт для всех реализаций табулированных функций, что позволяет использовать разные реализации через единый интерфейс.

Задание 2

Реализация класса ArrayTabulatedFunction

Создан класс ArrayTabulatedFunction, реализующий интерфейс TabulatedFunction на основе массива. Класс создан путем переименования существующего класса TabulatedFunction из второй лабораторной работы.

Конструкторы класса:

- ArrayTabulatedFunction(double leftX, double rightX, int pointsCount) - создает равномерную сетку
- ArrayTabulatedFunction(double leftX, double rightX, double[] values) - создает функцию с заданными значениями Y

Особенности реализации:

- Автоматическое расширение массива при добавлении точек
- Сохранение порядка точек по координате X
- Проверка корректности входных параметров с выбрасыванием исключений

Исходный код:

```
package functions;

public class ArrayTabulatedFunction implements TabulatedFunction {
    private FunctionPoint[] points;
    private int pointsCount;

    private static final double EPSILON = 1e-10;

    private boolean equals(double a, double b) {
        return Math.abs(a - b) < EPSILON;
    }

    private boolean lessOrEqual(double a, double b) {
        return a < b || equals(a, b);
    }

    public ArrayTabulatedFunction(double leftX, double rightX, int pointsCount) {
        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");
        }

        this.pointsCount = pointsCount;
        this.points = new FunctionPoint[pointsCount+10];

        double step = (rightX - leftX) / (pointsCount - 1);

        for (int i=0; i<pointsCount; i++) {
            double x = leftX + i*step;
            points[i] = new FunctionPoint(x,0.0);
        }
    }

    public ArrayTabulatedFunction(double leftX, double rightX, double[] values) {
        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");
        }

        this.pointsCount = values.length;
        this.points = new FunctionPoint[pointsCount+10];

        double step = (rightX - leftX) / (pointsCount - 1);

        for (int i=0; i<pointsCount; i++) {
            double x = leftX + i*step;
```

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

Задание 3

Модификация класса TabulatedFunction для обработки исключений

В существующий класс TabulatedFunction внесены изменения для корректной обработки исключительных ситуаций в соответствии с требованиями задания.

Реализованные проверки и исключения:

В конструкторах:

- Проверка, что leftX < rightX
- Проверка, что pointsCount ≥ 2
- При нарушении условий выбрасывается IllegalArgumentException

В методах работы с точками:

- Проверка корректности индекса в методах getPoint(), setPoint(), getPointX(), setPointX(), getPointY(), setPointY(), deletePoint()
- При выходе индекса за границы выбрасывается FunctionPointIndexOutOfBoundsException

В методах модификации точек:

- Проверка сохранения порядка точек в setPoint() и setPointX()
- Проверка отсутствия дублирования X в addPoint()
- При нарушении порядка выбрасывается InappropriateFunctionPointException

В методе deletePoint():

- Проверка, что количество точек ≥ 3
- При попытке удаления до менее 3 точек выбрасывается IllegalStateException

Задание 4

Реализация класса LinkedListTabulatedFunction

Создан класс для реализации табулированной функции на основе двусвязного циклического списка с выделенной головой.

Исходный код:

```
package functions;

public class LinkedListTabulatedFunction implements TabulatedFunction {
    // Внутренний класс для узла списка
    private static class FunctionNode {
        FunctionPoint point;
        FunctionNode prev;
        FunctionNode next;

        FunctionNode(FunctionPoint point) {
            this.point = point;
        }
    }

    private FunctionNode head; // Голова списка (не содержит данных)

    private FunctionNode lastAccessedNode; // Для оптимизации
    private int lastAccessedIndex;

    private int pointsCount;

    private static final double EPSILON = 1e-10;

    // Вспомогательные методы для сравнения
    private boolean equals(double a, double b) {
        return Math.abs(a - b) < EPSILON;
    }

    private boolean lessOrEqual(double a, double b) {
        return a < b || equals(a, b);
    }

    private void initializeList() {
        head = new FunctionNode(null);  // голова с null данными
        head.prev = head;                    // замыкаем на себя
        head.next = head;                    // замыкаем на себя
        pointsCount = 0;

        lastAccessedNode = head;
        lastAccessedIndex = -1;
    }
```

Поля класса LinkedListTabulatedFunction:

- private FunctionNode head - голова списка
- private int pointsCount - количество точек
- private FunctionNode lastAccessedNode - последний доступный узел для оптимизации

- private int lastAccessedIndex - индекс последнего доступного узла

Метод getNodeByIndex(int index)

```
    private FunctionNode getNodeByIndex(int index) {
        if (index < 0 || index >= pointsCount) {
            throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " вне
границ [0, " + (pointsCount - 1) + "]");
        }

        // Оптимизация: начинаем с последнего доступного узла если это ближе
        FunctionNode current;
        if (lastAccessedIndex != -1 && Math.abs(index - lastAccessedIndex) <
Math.min(index, pointsCount - index)) {
            current = lastAccessedNode;
            int currentIndex = lastAccessedIndex;

            if (index > currentIndex) {
                for (int i = currentIndex; i < index; i++) {
                    current = current.next;
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
            // Иначе начинаем с начала
            current = head.next;
            for (int i = 0; i < index; i++) {
                current = current.next;
            }
        }

        // Сохраняем для следующего вызова
        lastAccessedNode = current;
        lastAccessedIndex = index;

        return current;
    }
```

Метод addNodeToTail()

```
        private FunctionNode addNodeToTail() {

        return addNodeByIndex(pointsCount);
    }
```

```
    private FunctionNode getNodeByIndex(int index) {
        if (index < 0 || index >= pointsCount) {
```

```
            throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " вне
границ [0, " + (pointsCount - 1) + "]");
        }

        // Оптимизация: начинаем с последнего доступного узла если это ближе
        FunctionNode current;
        if (lastAccessedIndex != -1 && Math.abs(index - lastAccessedIndex) <
Math.min(index, pointsCount - index)) {
            current = lastAccessedNode;
            int currentIndex = lastAccessedIndex;

            if (index > currentIndex) {
                for (int i = currentIndex; i < index; i++) {
                    current = current.next;
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
            // Иначе начинаем с начала
            current = head.next;
            for (int i = 0; i < index; i++) {
                current = current.next;
            }
        }

        // Сохраняем для следующего вызова
        lastAccessedNode = current;
        lastAccessedIndex = index;

        return current;
    }
```

```
    private FunctionNode getNodeByIndex(int index) {
        if (index < 0 || index >= pointsCount) {
            throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " вне
границ [0, " + (pointsCount - 1) + "]");
        }

        // Оптимизация: начинаем с последнего доступного узла если это ближе
        FunctionNode current;
        if (lastAccessedIndex != -1 && Math.abs(index - lastAccessedIndex) <
Math.min(index, pointsCount - index)) {
            current = lastAccessedNode;
            int currentIndex = lastAccessedIndex;

            if (index > currentIndex) {
                for (int i = currentIndex; i < index; i++) {
                    current = current.next;
```

```
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
            // Иначе начинаем с начала
            current = head.next;
            for (int i = 0; i < index; i++) {
                current = current.next;
            }
        }

        // Сохраняем для следующего вызова
        lastAccessedNode = current;
        lastAccessedIndex = index;

        return current;
    }
```

```
    private FunctionNode getNodeByIndex(int index) {
        if (index < 0 || index >= pointsCount) {
            throw new FunctionPointIndexOutOfBoundsException("Индекс " + index + " вне
границ [0, " + (pointsCount - 1) + "]");
        }

        // Оптимизация: начинаем с последнего доступного узла если это ближе
        FunctionNode current;
        if (lastAccessedIndex != -1 && Math.abs(index - lastAccessedIndex) <
Math.min(index, pointsCount - index)) {
            current = lastAccessedNode;
            int currentIndex = lastAccessedIndex;

            if (index > currentIndex) {
                for (int i = currentIndex; i < index; i++) {
                    current = current.next;
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
            // Иначе начинаем с начала
            current = head.next;
            for (int i = 0; i < index; i++) {
                current = current.next;
            }
        }
```

```
        // Сохраняем для следующего вызова
        lastAccessedNode = current;
        lastAccessedIndex = index;

        return current;
    }
```

Задание 5

Реализовать в LinkedListTabulatedFunction конструкторы и методы, аналогичные TabulatedFunction.

Реализованы с теми же параметрами и исключениями:

- Проверка: левая граница < правой границы
- Проверка: количество точек ≥ 2
- Создание точек с равным шагом

Задание 6

1. Переименован класс: TabulatedFunction → ArrayTabulatedFunction
2. Создан интерфейс: TabulatedFunction.java с объявлениями всех методов
3. Оба класса реализуют интерфейс

Интерфейс TabulatedFunction:

```
package functions;

public interface TabulatedFunction {
    double getLeftDomainBorder();
    double getRightDomainBorder();
    double getFunctionValue(double x);
    int getPointsCount();
    FunctionPoint getPoint(int index);
    void setPoint(int index, FunctionPoint point) throws
InappropriateFunctionPointException;
    double getPointX(int index);
    void setPointX(int index, double x) throws InappropriateFunctionPointException;
    double getPointY(int index);
    void setPointY(int index, double y);
    void deletePoint(int index);
```

```
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;
}
```

Создана единая архитектура с интерфейсом TabulatedFunction.

Задание 7

Тестирование работы классов

Тестирование исключений:

1. Неверные границы - IllegalArgumentException
2. Недостаточное количество точек - IllegalArgumentException
3. Выход за границы индекса - FunctionPointIndexOutOfBoundsException
4. Нарушение порядка точек - InappropriateFunctionPointException
5. Дублирование X координаты - InappropriateFunctionPointException
6. Удаление при минимальном количестве точек – IllegalStateException

Вывод программы:

```
=== ТЕСТИРОВАНИЕ ARRAY TABULATED FUNCTION ===
ArrayTabulatedFunction создана успешно!
Границы: [0.0, 5.0]
Количество точек: 6

=== ТЕСТИРОВАНИЕ LINKED LIST TABULATED FUNCTION ===
LinkedListTabulatedFunction создана успешно!
Границы: [0.0, 5.0]
Количество точек: 6

=== ТЕСТИРОВАНИЕ ИСКЛЮЧЕНИЙ ===
1. Тест неверных границ:
Поймано исключение: Левая граница должна быть меньше правой
```

```
2. Тест недостаточного количества точек:
Поймано исключение: Количество точек должно быть не менее 2
3. Тест выхода за границы индекса:
Поймано исключение: Индекс 10 вне границ [0, 2]
4. Тест нарушения порядка точек:
Поймано исключение: X координата должна быть меньше следующей точки
5. Тест дублирования X координаты:
Поймано исключение: Точка с X=1.0 уже существует
6. Тест удаления при минимальном количестве точек:
Поймано исключение: Нельзя удалить точку: минимальное количество точек - 3
```

Все классы работают корректно, исключения обрабатываются правильно.

Выводы

1. Успешное освоение обработки исключений

В ходе работы были изучены и применены на практике различные типы исключений Java. Созданы пользовательские классы исключений, корректно интегрированные в иерархию стандартных исключений Java. Это позволило реализовать надежную систему обработки ошибок в классах работы с функциями.

2. Реализация двух различных структур данных

Были успешно разработаны две альтернативные реализации табулированных функций:

- ArrayTabulatedFunction - на основе массива (модифицированная версия из предыдущей работы)
- LinkedListTabulatedFunction - на основе двусвязного циклического списка

Обе реализации демонстрируют различные подходы к хранению и обработке данных, что расширяет понимание структур данных.

3. Создание гибкой архитектуры с использованием интерфейсов

Разработан интерфейс TabulatedFunction, который обеспечивает:

- Единый контракт для различных реализаций
- Возможность легкой замены реализации
- Соблюдение принципов объектно-ориентированного программирования

4. Оптимизация доступа к элементам списка

В LinkedListTabulatedFunction реализована эффективная система доступа к элементам через механизм кэширования последнего доступного узла, что значительно улучшает производительность при последовательном доступе.

5. Корректная работа с вещественными числами

Во всех классах реализовано корректное сравнение вещественных чисел через машинный эпсилон, что исключает ошибки округления и обеспечивает надежную работу с числами с плавающей точкой.

Практическая значимость

Полученные навыки:

- Работа с пользовательскими исключениями
- Реализация сложных структур данных (двусвязные списки)
- Принципы инкапсуляции и сокрытия реализации
- Оптимизация алгоритмов доступа
- Создание и использование интерфейсов

Проверка корректности:

Все разработанные классы прошли комплексное тестирование, включая:

- Тестирование нормальной работы
- Тестирование граничных случаев
- Тестирование обработки исключений
- Сравнительный анализ двух реализаций

Заключение

Лабораторная работа успешно завершена. Все поставленные задачи выполнены в полном объеме. Созданный программный комплекс демонстрирует корректную работу, надежную обработку ошибок и гибкую архитектуру, позволяющую легко расширять функциональность в будущем.

Разработанные классы могут быть использованы в дальнейшем для решения задач численного анализа, интерполяции функций и других математических вычислений, требующих работы с табулированными данными.