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)

ОТЧЕТ ПО
ЛАБОРАТОРНОЙ РАБОТЕ № 7

«Использование паттернов: “Итератор”, “Фабричный метод” и рефлексии»

по курсу
Объектно-ориентированное программирование

Выполнила: Элибекян Анжелина,
студентка группы 6203-010302D

Оглавление

Содержание

Задание №1

Я изменила интерфейс TabulatedFunction, чтобы он поддерживал циклы for-each. Для этого добавила наследование от Iterable. Итераторы должны были быть реализованы как анонимные классы, эффективно использовать внутреннюю структуру данных и обеспечивать защиту инкапсуляции.

```
// добавлен интерфейс Iterable для поддержки for-each циклов
public interface TabulatedFunction extends Function, Cloneable, Iterable<FunctionPoint> {
```

В классе ArrayTabulatedFunction я создала итератор, который работает с внутренним массивом точек:

```java
// реализация итератора для поддержки for-each циклов
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private int currentIndex = 0; // текущая позиция в массиве  2 usages

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

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                // бросаем исключение если элементов больше нет
                throw new NoSuchElementException("В табличной функции больше нет элементов");
            }
            // возвращаем копию точки для защиты инкапсуляции
            return new FunctionPoint(points[currentIndex++]);
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Операция удаления не поддерживается.");
        }
    };
```

Особенность этой реализации в том, что итератор непосредственно обращается к внутреннему массиву points, что обеспечивает высокую производительность. При этом для защиты инкапсуляции каждый раз возвращается копия точки, а не оригинальный объект.

Для LinkedListTabulatedFunction реализация итератора была более сложной из-за связной структуры данных:

```java
// реализация итератора для связного списка
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.next;  4 usages

        @Override
        public boolean hasNext() {
            // проверяем, не дошли ли до головы (конца списка)
            return currentNode != head;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                // бросаем исключение если элементов больше нет
                throw new NoSuchElementException("В табличной функции больше нет элементов");
            }
            // создаем копию точки и переходим к следующему узлу
            FunctionPoint point = new FunctionPoint(currentNode.point);
            currentNode = currentNode.next;
            return point;
        }

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

Здесь итератор проходит по узлам связного списка, начиная с первого реального узла (head.next) и доходит до головы списка, которая служит маркером конца.
Для проверки корректности работы итераторов я добавила тестовый код в метод main:

```
System.out.println("\n=== Задание 1: Проверка итераторов ===");

// проверяем работу итератора для ArrayTabulatedFunction
TabulatedFunction arrF = new ArrayTabulatedFunction( leftX: 0,  rightX: 10,  pointsCount: 8);
for (FunctionPoint p : arrF){
    System.out.println(p);  // используем for-each цикл
}
```

```
=== Задание 1: Проверка итераторов ===
(0.0;0.0)
(1.4285714285714286;0.0)
(2.857142857142857;0.0)
(4.285714285714286;0.0)
(5.714285714285714;0.0)
(7.142857142857143;0.0)
(8.571428571428571;0.0)
(10.0;0.0)
```

Задание №2

В данном задании требовалось устранить жесткую привязку к конкретным классам табулированных функций в методах класса TabulatedFunctions, реализовав механизм фабрик, позволяющий динамически выбирать тип создаваемых объектов. Первым шагом я создала интерфейс фабрики в пакете functions:

```
// интерфейс фабрики для создания табулированных функций
// позволяет создавать объекты разных типов динамически
public interface TabulatedFunctionFactory {  4 usages  2 implementations
    // создает функцию по границам и количеству точек
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount)
    // создает функцию по границам и массиву значений y
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values)
    // создает функцию по массиву точек
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);  1 usage  2 implementations
}
```

Далее я реализовала конкретные фабрики как вложенные статические классы в ArrayTabulatedFunction и LinkedListTabulatedFunction:

```
// фабрика для создания объектов ArrayTabulatedFunction
public static class ArrayTabulatedFunctionFactory implements TabulatedFunctionFactory {  2 usages
    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
        return new ArrayTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
        return new ArrayTabulatedFunction(leftX, rightX, values);
    }

    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
        return new ArrayTabulatedFunction(points);
    }
}
```

```
// фабрика для создания объектов LinkedListTabulatedFunction
public static class LinkedListTabulatedFunctionFactory implements TabulatedFunctionFactory {  1 usage
    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
        return new LinkedListTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
        return new LinkedListTabulatedFunction(leftX, rightX, values);
    }

    @Override  1 usage
    public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
        return new LinkedListTabulatedFunction(points);
    }
}
```

В классе TabulatedFunctions я добавила статическое поле для хранения текущей фабрики и методы для работы с ней:

```
private static TabulatedFunctionFactory factory = new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();

// метод для замены фабрики в runtime
public static void setTabulatedFunctionFactory(TabulatedFunctionFactory factory) {  2 usages
    TabulatedFunctions.factory = factory;
}

// методы создания функций через фабрику
public static TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {  1 usag
    return factory.createTabulatedFunction(leftX, rightX, pointsCount);
}

public static TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {  no usa
    return factory.createTabulatedFunction(leftX, rightX, values);
}

public static TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {  2 usages
    return factory.createTabulatedFunction(points);
}
```

Ключевым изменением была замена прямого создания объектов через конструкторы на вызовы фабричных методов. Например, в методе tabulate:
Было:
ArrayTabulatedFunction tabulatedFunc = new ArrayTabulatedFunction(leftX, rightX, pointsCount);

Стало:
TabulatedFunction tabulatedFunc = *createTabulatedFunction*(leftX, rightX, pointsCount);

```
System.out.println("\n=== Задание 2: Проверка фабрик ===");

// тестируем смену фабрик во время выполнения
Function f = new Cos();
TabulatedFunction tf;
// создаем функцию через фабрику по умолчанию (Array)
tf = TabulatedFunctions.tabulate(f, leftX: 0, Math.PI, pointsCount: 11);
System.out.println(tf.getClass()); // должен быть ArrayTabulatedFunction
// меняем фабрику на LinkedList
TabulatedFunctions.setTabulatedFunctionFactory(new LinkedListTabulatedFunction.LinkedListTabulatedFunctionFactory());
tf = TabulatedFunctions.tabulate(f, leftX: 0, Math.PI, pointsCount: 11);
System.out.println(tf.getClass()); // должен быть LinkedListTabulatedFunction
// возвращаем фабрику Array обратно
TabulatedFunctions.setTabulatedFunctionFactory(new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory());
tf = TabulatedFunctions.tabulate(f, leftX: 0, Math.PI, pointsCount: 11);
System.out.println(tf.getClass()); // снова ArrayTabulatedFunction
```

Этот тест наглядно демонстрирует, как можно динамически менять тип создаваемых функций во время выполнения программы.

```
=== Задание 2: Проверка фабрик ===
class functions.ArrayTabulatedFunction
class functions.LinkedListTabulatedFunction
class functions.ArrayTabulatedFunction
```

Задание №3

Требовалось реализовать альтернативный механизм создания табулированных функций с использованием рефлексии, позволяющий указывать конкретный класс для инстанцирования через объект Class.
Я добавила три перегруженных метода createTabulatedFunction, принимающих параметр Class<?> functionClass:

```java
// методы создания функций через рефлексию
public static TabulatedFunction createTabulatedFunction(Class<?> functionClass, double leftX, double rightX, int pointsCount) {
    try {
        // проверяем, что класс реализует TabulatedFunction
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
            throw new IllegalArgumentException("Класс должен реализовывать интерфейс TabulatedFunction");
        }

        // 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 (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}

public static TabulatedFunction createTabulatedFunction(Class<?> functionClass, double leftX, double rightX, double[] values) {
    try {
        // проверяем, что класс реализует TabulatedFunction
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
            throw new IllegalArgumentException("Класс должен реализовывать интерфейс TabulatedFunction");
        }
        // ищем конструктор с параметрами (double, double, double[])
        Constructor<?> constructor = functionClass.getConstructor(double.class, double.class, double[].class);

        // создаем объект через рефлексию
        return (TabulatedFunction) constructor.newInstance(leftX, rightX, values);

    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}

public static TabulatedFunction createTabulatedFunction(Class<?> functionClass, FunctionPoint[] points) {  1 usage
    try {
        // проверяем, что класс реализует TabulatedFunction
        if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
            throw new IllegalArgumentException("Класс должен реализовывать интерфейс TabulatedFunction");
        }

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

        // создаем объект через рефлексию
        return (TabulatedFunction) constructor.newInstance((Object) points);

    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}

// перегруженный метод табуляции с рефлексией
public static TabulatedFunction tabulate(Class<?> functionClass, Function function, double leftX, double rightX, int pointsCount) {
    // стандартные проверки параметров
```

```java
// перегруженный метод табуляции с рефлексией
public static TabulatedFunction tabulate(Class<?> functionClass, Function function, double leftX, double rightX, int pointsCount) {
    // стандартные проверки параметров
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Количество точек должно быть не менее 2");
    }
    if (leftX < function.getLeftDomainBorder() || rightX > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы табуляции выходят за область определения");
    }
    if (leftX >= rightX) {
        throw new IllegalArgumentException("Левая граница должна быть меньше правой");
    }

    // создаем табулированную функцию через рефлексию
    TabulatedFunction tabulatedFunc = createTabulatedFunction(functionClass, leftX, rightX, pointsCount);

    // заполняем значения y
    for (int i = 0; i < pointsCount; i++) {
        double x = tabulatedFunc.getPointX(i);
        double y = function.getFunctionValue(x);
        tabulatedFunc.setPointY(i, y);
    }
    return tabulatedFunc;
}
```

Теперь можно создавать функции, указывая конкретный класс:

```java
System.out.println("\n=== Задание 3: Проверка рефлексии ===");
// тестируем создание объектов через рефлексию

TabulatedFunction rf; // переиспользуем переменную для разных тестов

rf = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, leftX: 0, rightX: 10, pointsCount: 3);
System.out.println(rf.getClass());
System.out.println(rf);

rf = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, leftX: 0, rightX: 10, new double[] {0, 10});
System.out.println(rf.getClass());
System.out.println(rf);

rf = TabulatedFunctions.createTabulatedFunction(LinkedListTabulatedFunction.class, new FunctionPoint[] {
                new FunctionPoint( x: 0, y: 0),
                new FunctionPoint( x: 10, y: 10)
        }
);
System.out.println(rf.getClass());
System.out.println(rf);

rf = TabulatedFunctions.tabulate(LinkedListTabulatedFunction.class, new Sin(), leftX: 0, Math.PI, pointsCount: 11);
System.out.println(rf.getClass());
System.out.println(rf);
```

Результат:

```
=== Задание 3: Проверка рефлексии ===
class functions.ArrayTabulatedFunction
{(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)}
class functions.ArrayTabulatedFunction
{(0.0; 0.0), (10.0; 10.0)}
class functions.LinkedListTabulatedFunction
{(0.0; 0.0), (10.0; 10.0)}
class functions.LinkedListTabulatedFunction
{(0.0; 0.0), (0.3141592653589793; 0.3090169943749474), (0.6283185307179586; 0.5877852522924731),
```