Лабораторная работа 7

Выполнил: студент

Группы 6301-030301D

Дымченко В.Р.

Задание 1

Делаем так, чтобы все объекты типа TabulatedFunction можно было использовать в качестве объекта-агрегата в “улучшенном цикле for”В интерфейсе TabulatedFunction добавляем необходимый родительский тип.

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

В классах, реализующих интерфейс TabulatedFunction, добавляем требующийся метод, возвращающий объект итератора.

ArrayTabulatedFunction

```
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private int currentIndex = 0;  2 usages

        @Override
        public boolean hasNext() {
            return currentIndex < pointsCount;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new NoSuchElementException("Нет следующего элемента");
            }
            return new FunctionPoint(points[currentIndex++]);
        }

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

LinkedListTabulatedFunction

```
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.next;  4 usages

        @Override
        public boolean hasNext() {
            return currentNode != head;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new NoSuchElementException("Нет следующего элемента");
            }
            FunctionPoint point = new FunctionPoint(currentNode.point);
            currentNode = currentNode.next;
            return point;
        }

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

Задание 2

В пакете functions описываем 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 {  no usages  2 implementations
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount);
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values);
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);  no usages  2 implementations
}
```

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

```
private static TabulatedFunctionFactory factory = new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();  7 usa
```

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

```
public static void setTabulatedFunctionFactory(TabulatedFunctionFactory factory) {  no usages
    TabulatedFunctions.factory = factory;
}

public static TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {  no us
    return factory.createTabulatedFunction(leftX, rightX, pointsCount);
}

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

public static TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {  no usages
    return factory.createTabulatedFunction(points);
}
```

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

Задание 3

В классе TabulatedFunctions добавьте 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. Сделайте так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс TabulatedFunction.

Новые методы создания объектов должны найти в предложенном классе конструктор с соответствующими типами параметров (например, двумя параметрами типа double и одним параметром типа int для метода, создающего объект табулированной функции по левой и правой границе области определения и количеству точек). С помощью найденного конструктора (в него должны быть переданы фактические параметры) должен быть создан объект табулированной функции. Ссылка на этот объект и должна быть возвращена из метода создания.

Если в ходе выполнения рефлексивных операций возникло исключение (не найден конструктор и т.д.), оно должно быть отловлено (используйте блок try с отловом нескольких типов исключений). Вместо него должно быть выброшено исключение IllegalArgumentException, причём в его конструктор должно быть передано отловленное исключение из рефлексии. Это позволит в случае возникновения ошибок определить реальную причину ошибки.

В классе TabulatedFunctions перегрузите методы, создающие объекты табулированных функций, добавив версии, принимающие также ссылку типа Class на

описание класса, объект которого требуется создать. Сделайте так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс TabulatedFunction.

```
public static TabulatedFunction createTabulatedFunction(Class<? extends TabulatedFunction> functionClass, double leftX, double rightX, int pointsCount) {
    try {
        Constructor<? extends TabulatedFunction> constructor =
                functionClass.getConstructor(double.class, double.class, int.class);
        return constructor.newInstance(leftX, rightX, pointsCount);
    } catch (NoSuchMethodException | IllegalAccessException |
             InstantiationException | InvocationTargetException e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}
```

```
public static TabulatedFunction createTabulatedFunction(Class<? extends TabulatedFunction> functionClass, double leftX, double rightX, double[] values) {  1 usage
    try {
        Constructor<? extends TabulatedFunction> constructor =
                functionClass.getConstructor(double.class, double.class, double[].class);
        return constructor.newInstance(leftX, rightX, values);
    } catch (NoSuchMethodException | IllegalAccessException |
             InstantiationException | InvocationTargetException e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}
```

```
public static TabulatedFunction createTabulatedFunction(Class<? extends TabulatedFunction> functionClass, FunctionPoint[] points) {  2 usages
    try {
        Constructor<? extends TabulatedFunction> constructor =
                functionClass.getConstructor(FunctionPoint[].class);
        return constructor.newInstance((Object)points);
    } catch (NoSuchMethodException | IllegalAccessException |
             InstantiationException | InvocationTargetException e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}
```

Новый перегруженный метод

```
public static TabulatedFunction tabulate(Class<? extends TabulatedFunction> functionClass, Function function, doub
    if (leftX < function.getLeftDomainBorder() || rightX > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы табуляции выходят за область определения функции");
    }
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Требуется не менее 2 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);
    }

    return createTabulatedFunction(functionClass, points);
}
```

Тестирование

```
Тестирование:

1.Тестирование итератора

Тестирование итератора для ArrayTabulatedFunction:
Создана функция: f(x) = x² на [0, 10] с 6 точками
Точки функции:
   Точка 1: (0.0; 0.0)
   Точка 2: (2.0; 1.0)
   Точка 3: (4.0; 4.0)
   Точка 4: (6.0; 9.0)
   Точка 5: (8.0; 16.0)
   Точка 6: (10.0; 25.0)

Тестирование итератора для LinkedListTabulatedFunction:
Создана функция: f(x) = x на [0, 5] с 6 точками
Точки функции:
   Точка 1: (0.0; 0.0)
   Точка 2: (1.0; 1.0)
   Точка 3: (2.0; 2.0)
   Точка 4: (3.0; 3.0)
   Точка 5: (4.0; 4.0)
   Точка 6: (5.0; 5.0)

Итераторы работают(можно использовать for-each)
```

```
2.Тестирование фабричного метода

Демонстрация работы фабричного метода:

Фабрика по умолчанию (должна создавать ArrayTabulatedFunction):
Создана tabulate(f, 0, π, 11)
Тип созданного объекта: ArrayTabulatedFunction
Количество точек: 11

Меняем фабрику на LinkedListTabulatedFunctionFactory:
Создана tabulate(f, 0, π, 11) с новой фабрикой
Тип созданного объекта: LinkedListTabulatedFunction
Количество точек: 11

Возвращаем фабрику обратно на ArrayTabulatedFunctionFactory:
Создана tabulate(f, 0, π, 11) с восстановленной фабрикой
Тип созданного объекта: ArrayTabulatedFunction

Фабричный метод работает(можно динамически менять тип создаваемых объектов)
```

```
3.Тестирование рефлексии

Демонстрация работы рефлексии:


Создание ArrayTabulatedFunction через рефлексию:
Вызов: createTabulatedFunction(ArrayTabulatedFunction.class, 0, 10, 3)
Результат: {(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)}
Тип: ArrayTabulatedFunction

Создание ArrayTabulatedFunction с массивом значений:
Вызов: createTabulatedFunction(ArrayTabulatedFunction.class, 0, 10, new double[]{0, 10})
Результат: {(0.0; 0.0), (10.0; 10.0)}
Тип: ArrayTabulatedFunction

Создание LinkedListTabulatedFunction с массивом точек:
Вызов: createTabulatedFunction(LinkedListTabulatedFunction.class, points[])
Результат: {(0.0; 0.0), (10.0; 10.0)}
Тип: LinkedListTabulatedFunction

Использование tabulate() с рефлексией:
Вызов: tabulate(LinkedListTabulatedFunction.class, sin, 0, π, 11)
Тип: LinkedListTabulatedFunction
Количество точек: 11

Демонстрация обработки ошибок рефлексии:
Поймано исключение: Ошибка при создании объекта через рефлексию
Причина: NoSuchMethodException

Все тесты завершены успешно
```

Изменения:

1.Чтение из InputStream с рефлексией

```java
public static TabulatedFunction inputTabulatedFunction(Class<? extends TabulatedFunction> functionClass,
                                                        InputStream in) {
    try (DataInputStream dis = new DataInputStream(in)) {
        int pointsCount = dis.readInt();
        FunctionPoint[] points = new FunctionPoint[pointsCount];

        for (int i = 0; i < pointsCount; i++) {
            double x = dis.readDouble();
            double y = dis.readDouble();
            points[i] = new FunctionPoint(x, y);
        }

        return createTabulatedFunction(functionClass, points);
    } catch (IOException e) {
        throw new RuntimeException("Ошибка чтения табулированной функции из потока", e);
    }
}
```

2. Чтение из Reader с рефлексией

```
public static TabulatedFunction readTabulatedFunction(Class<? extends TabulatedFunction> functionClass,
                                                      Reader in) {
    try {
        StreamTokenizer tokenizer = new StreamTokenizer(in);
        tokenizer.parseNumbers();

        if (tokenizer.nextToken() != StreamTokenizer.TT_NUMBER) {
            throw new RuntimeException("Ожидалось количество точек");
        }
        int pointsCount = (int) tokenizer.nval;

        FunctionPoint[] points = new FunctionPoint[pointsCount];

        for (int i = 0; i < pointsCount; i++) {
            if (tokenizer.nextToken() != StreamTokenizer.TT_NUMBER) {
                throw new RuntimeException("Ожидалась координата X");
            }
            double x = tokenizer.nval;

            if (tokenizer.nextToken() != StreamTokenizer.TT_NUMBER) {
                throw new RuntimeException("Ожидалась координата Y");
            }
            double y = tokenizer.nval;

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

        return createTabulatedFunction(functionClass, points);
    } catch (IOException e) {
```

```
        return createTabulatedFunction(functionClass, points);
    } catch (IOException e) {
        throw new RuntimeException("Ошибка чтения табулированной функции из reader", e);
    }
```

И в результате получаем:

```
4. Тестирование чтения с рефлексией

Тестирование методов чтения с рефлексией:

Исходная функция: {(0.0; 0.0), (2.0; 1.0), (4.0; 4.0), (6.0; 9.0), (8.0; 16.0), (10.0; 25.0)}

Чтение из бинарного потока:
   Прочитан как ArrayTabulatedFunction: ArrayTabulatedFunction
   Результат: {(0.0; 0.0), (2.0; 1.0), (4.0; 4.0), (6.0; 9.0), (8.0; 16.0), (10.0; 25.0)}
   Прочитан как LinkedListTabulatedFunction: LinkedListTabulatedFunction
   Результат: {(0.0; 0.0), (2.0; 1.0), (4.0; 4.0), (6.0; 9.0), (8.0; 16.0), (10.0; 25.0)}

Чтение из текстового потока:
    Текстовые данные: "6 0.0 0.0 2.0 1.0 4.0 4.0 6.0 9.0 8.0 16.0 10.0 25.0"
   Прочитан из текста как LinkedListTabulatedFunction: LinkedListTabulatedFunction
   Результат: {(0.0; 0.0), (2.0; 1.0), (4.0; 4.0), (6.0; 9.0), (8.0; 16.0), (10.0; 25.0)}

Сравнение функций:
   Исходная и прочитанная (Array) идентичны: true
   Исходная и прочитанная (List) идентичны: true
   Array и List представления идентичны: true

Демонстрация обработки ошибок:
Поймано исключение (ожидаемо): RuntimeException

Методы чтения с рефлексией работают
Можно указать класс объекта при чтении из потока
```