Лабораторная работа по ООП №7

Студент: Тарасова Полина
Группа: 6204-010302D

Задание 1

Я добавила интерфейс Iterable<FunctionPoint> для TabulatedFunction, чтобы можно было использовать цикл for-each. В классах ArrayTabulatedFunction и LinkedListTabulatedFunction реализован метод iterator(), возвращающий анонимный итератор. Итератор проходит по внутренней структуре без лишних вызовов методов класса, метод remove() выбрасывает исключение UnsupportedOperationException, а метод next() — NoSuchElementException, если элементов больше нет. Итератор позволяет легко проходить по функциям, не заботясь о внутреннем устройстве списка или массива.

Итератор в ArrayTabulatedFunction:

```
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private int curIndex = 0;
        @Override
        public boolean hasNext() {
            return curIndex < pointsCount;
        }

        @Override
        public FunctionPoint next() {
            if (curIndex >= pointsCount) {
                throw new NoSuchElementException();
            }
            FunctionPoint fp = new FunctionPoint(points[curIndex]);
            curIndex ++;
            return fp;
        }
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
```

Итератор в LinkedListTabulatedFunction:

```
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode current =
LinkedListTabulatedFunction.this.head.next;
        @Override
        public boolean hasNext() {
            return current != LinkedListTabulatedFunction.this.head;
        }
```

```
        @Override
        public FunctionPoint next() {
            if (current == LinkedListTabulatedFunction.this.head) {
                throw new NoSuchElementException();
            }
            FunctionPoint fp = new FunctionPoint(current.point);
            current = current.next;
            return fp;
        }
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
```

Проверка в main:

```
System.out.println("Проверка итератора ArrayTabulatedFunction:");
TabulatedFunction func1 = new ArrayTabulatedFunction(new FunctionPoint[]{new
FunctionPoint(0,0), new FunctionPoint(1,1), new FunctionPoint(2,4)});
for (FunctionPoint p : func1) {
    System.out.println(p);
}
System.out.println();

System.out.println("Проверка итератора LinkedListTabulatedFunction:");
TabulatedFunction func2 = new LinkedListTabulatedFunction(new
FunctionPoint[]{new FunctionPoint(0,0), new FunctionPoint(1,1), new
FunctionPoint(2,4)});
for (FunctionPoint p : func2) {
    System.out.println(p);
}
System.out.println();
```

Задание 2

Я создала интерфейс TabulatedFunctionFactory с тремя методами createTabulatedFunction() для разных вариантов конструктора. В ArrayTabulatedFunction и LinkedListTabulatedFunction добавила вложенные публичные фабрики. В классе TabulatedFunctions сделала статическое поле текущей фабрики и методы setTabulatedFunctionFactory() и createTabulatedFunction(), которые используют фабрику для создания объектов. Фабрика помогает динамически менять тип создаваемого объекта. Её можно менять, не трогая остальной код.

TabulatedFunctionFactory:

```
package functions;

public interface TabulatedFunctionFactory {
    TabulatedFunction createTabulatedFunction(double leftX, double rightX,
```

```
double[] values);
    TabulatedFunction createTabulatedFunction(double leftX, double rightX,
int pointsCount);
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);
}
```

ArrayTabulatedFunction:

```
public static class ArrayTabulatedFunctionFactory implements
TabulatedFunctionFactory {
    @Override
    public TabulatedFunction createTabulatedFunction(double leftX, double
rightX, double[] values) {
        return new ArrayTabulatedFunction(leftX, rightX, values);
    }

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

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

LinkedListTabulatedFunction:

```
public static class LinkedListTabulatedFunctionFactory implements
TabulatedFunctionFactory {
    @Override
    public TabulatedFunction createTabulatedFunction(double leftX, double
rightX, double[] values) {
        return new LinkedListTabulatedFunction(leftX, rightX, values);
    }

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

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

TabulatedFunctions:

```
public static void setTabulatedFunctionFactory(TabulatedFunctionFactory
functionFactory) {
    TabulatedFunctions.functionFactory = functionFactory;
}
```

```
public static TabulatedFunction createTabulatedFunction(double leftX, double
rightX, double[] values) {
    return functionFactory.createTabulatedFunction(leftX, rightX, values);
}

public static TabulatedFunction createTabulatedFunction(double leftX, double
rightX, int pointsCount) {
    return functionFactory.createTabulatedFunction(leftX, rightX,
pointsCount);
}

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

Проверка в main:

```
System.out.println("Проверка работы Фабрики:");
Function sin = new Sin();
TabulatedFunction tabSin;
tabSin = TabulatedFunctions.tabulate(sin, 0, Math.PI, 11);
System.out.println(tabSin.getClass());
TabulatedFunctions.setTabulatedFunctionFactory(new
LinkedListTabulatedFunction.LinkedListTabulatedFunctionFactory());
tabSin = TabulatedFunctions.tabulate(sin, 0, Math.PI, 11);
System.out.println(tabSin.getClass());
TabulatedFunctions.setTabulatedFunctionFactory(new
ArrayTabulatedFunction.ArrayTabulatedFunctionFactory());
tabSin = TabulatedFunctions.tabulate(sin, 0, Math.PI, 11);
System.out.println(tabSin.getClass());
System.out.println();
```

Задание 3

В классе TabulatedFunctions я добавила новые перегруженные методы createTabulatedFunction(), которые принимают Class<? extends TabulatedFunction> и создают объекты через рефлексию. Исключения, возникающие при поиске конструктора или создании объекта, перехватываются и оборачиваются в IllegalArgumentException. Также я добавила метод tabulate() с указанием класса. С помощью рефлексии удобно создавать объекты динамически без привязки к конкретному классу. Это даёт большую гибкость.

```
public static TabulatedFunction createTabulatedFunction(Class<? extends
TabulatedFunction> clazz, double leftX, double rightX, double[] values) {
    if (clazz == null || values == null) {
        throw new IllegalArgumentException("Пустой параметр");
    }
    try {
        Constructor<? extends TabulatedFunction> constructor =
clazz.getConstructor(double.class, double.class, double[].class);
```

```
            return constructor.newInstance(leftX, rightX, values);
        }
        catch (NoSuchMethodException | InstantiationException |
IllegalAccessException | InvocationTargetException e) {
            throw new IllegalArgumentException("Ошибка при создании объекта через
рефлексию", e);
        }
    }

    public static TabulatedFunction createTabulatedFunction(Class<? extends
TabulatedFunction> clazz, double leftX, double rightX, int pointsCount) {
        if (clazz == null) {
            throw new IllegalArgumentException("Пустой параметр");
        }
        if (pointsCount < 2) {
            throw new IllegalArgumentException("Количество точек < 2");
        }
        try {
            Constructor<? extends TabulatedFunction> constructor =
clazz.getConstructor(double.class, double.class, int.class);
            return constructor.newInstance(leftX, rightX, pointsCount);
        }
        catch (NoSuchMethodException | InstantiationException |
IllegalAccessException | InvocationTargetException e) {
            throw new IllegalArgumentException("Ошибка при создании объекта через
рефлексию", e);
        }
    }

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

    public static TabulatedFunction tabulate(Class<? extends TabulatedFunction>
clazz, Function function, double leftX, double rightX, int pointsCount) {
        if (leftX >= rightX) {
            throw new IllegalArgumentException("Левая граница >= правой");
        }
        if (pointsCount < 2) {
            throw new IllegalArgumentException("Количество точек < 2");
        }
        double step = (rightX - leftX) / (pointsCount - 1);
        FunctionPoint[] points = new FunctionPoint[pointsCount];
        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(clazz, points);
    }
```

Проверка в main:

```
System.out.println("Проверка работы методов рефлексивного создания
объектов:");
TabulatedFunction func;
func =
TabulatedFunctions.createTabulatedFunction(ArrayTabulatedFunction.class, 0,
10, 3);
System.out.println(func.getClass());
System.out.println(func);
func =
TabulatedFunctions.createTabulatedFunction(ArrayTabulatedFunction.class, 0,
10, new double[] {0, 10});
System.out.println(func.getClass());
System.out.println(func);
func =
TabulatedFunctions.createTabulatedFunction(LinkedListTabulatedFunction.class,
new FunctionPoint[] {new FunctionPoint(0, 0), new FunctionPoint(10, 10)});
System.out.println(func.getClass());
System.out.println(func);
func = TabulatedFunctions.tabulate(LinkedListTabulatedFunction.class, new
Cos(), 0, Math.PI, 11);
System.out.println(func.getClass());
System.out.println(func);
```

Пример работы программы:

Проверка итератора ArrayTabulatedFunction:

(0.0; 0.0)

(1.0; 1.0)

(2.0; 4.0)

Проверка итератора LinkedListTabulatedFunction:

(0.0; 0.0)

(1.0; 1.0)

(2.0; 4.0)

Проверка работы Фабрики:

class functions.ArrayTabulatedFunction

class functions.LinkedListTabulatedFunction

class functions.ArrayTabulatedFunction

Проверка работы методов рефлексивного создания объектов:

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; 1.0), (0.3141592653589793; 0.9510565162951535), (0.6283185307179586; 0.8090169943749475), (0.9424777960769379; 0.5877852522924731), (1.2566370614359172; 0.30901699437494745), (1.5707963267948966; 6.123233995736766E-17), (1.8849555921538759; -0.30901699437494734), (2.199114857512855; -0.587785252292473), (2.5132741228718345; -0.8090169943749473), (2.827433388230814; -0.9510565162951535), (3.141592653589793; -1.0)}