# Лабораторная работа № 7

Выполнила: Оленина Арина Игоревна
группа 6204-010302D

Оглавление

# Задание 1

Я сделала так, чтобы все объекты типа TabulatedFunction можно было использовать в качестве объекта-агрегата в «улучшенном цикле for» (вариант for-each), извлекаемые объекты имеют тип FunctionPoint.
В интерфейсе TabulatedFunction добавила необходимый родительский тип, использовала при этом параметризованный тип (generic type).

```
package functions;
import java.util.Iterator;

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

В классах, реализующих интерфейс TabulatedFunction, добавила метод, возвращающий объект итератора.
Классы итераторов сделала анонимными. В соответствии с паттерном «Итератор» итераторы работают эффективно и используют знание о внутренней структуре объектов, а не вызывают публичные методы объекта табулированной функции.

Метод удаления всегда выбрасывает
исключение UnsupportedOperationException.
Метод получения следующего элемента выбрасывает
исключение NoSuchElementException, если следующего элемента нет.
Возвращаемый методом объект типа FunctionPoint не позволяет нарушить инкапсуляцию объекта табулированной функции.

```
public class ArrayTabulatedFunction implements
TabulatedFunction, Externalizable {
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private int currentIndex = 0;

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

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException("No
more elements");
            }
            // Возвращение копии точки для сохранения
инкапсуляции
            return new FunctionPoint(points[currentIndex++]);
        }
        @Override
        public void remove() {
```

```
            throw new UnsupportedOperationException("Remove
operation is not supported");
        }
    };
}
}
public class LinkedListTabulatedFunction implements
TabulatedFunction {
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.next;
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentNode != head && currentIndex <
pointsCount;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException("No
more elements");
            }
            FunctionPoint point = new
FunctionPoint(currentNode.point);
            currentNode = currentNode.next;
            currentIndex++;
            return point;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Remove
operation is not supported");
        }
    };
}
}
```

В методе main() программы проверила работу итераторов классов табулированных функций следующим образом.

```
public static void main(String[] args) {
    System.out.println("Задание 1");
    // Тест ArrayTabulatedFunction
    System.out.println("ArrayTabulatedFunction:");
    FunctionPoint[] points1 = {
            new FunctionPoint(0, 0),
            new FunctionPoint(1, 1),
            new FunctionPoint(2, 8),
            new FunctionPoint(3, 27)
```

```
        };
        TabulatedFunction arrayFunc = new
ArrayTabulatedFunction(points1);

        for (FunctionPoint p : arrayFunc) {
            System.out.println(p);
        }

        // Тест LinkedListTabulatedFunction
        System.out.println("\nLinkedListTabulatedFunction:");
        TabulatedFunction listFunc = new
LinkedListTabulatedFunction(points1);

        for (FunctionPoint p : listFunc) {
            System.out.println(p);
        }

        // Тест пустой функции
        System.out.println("\nEmpty function:");
        TabulatedFunction emptyFunc = new ArrayTabulatedFunction(0,
1, 2);
        for (FunctionPoint p : emptyFunc) {
            System.out.println(p);
        }
}
```

Вывод:

```
Задание 1
ArrayTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 8.0)
(3.0; 27.0)

LinkedListTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 8.0)
(3.0; 27.0)

Empty function:
(0.0; 0.0)
(1.0; 0.0)
```

## Задание 2

В классе TabulatedFunctions ряд методов в ходе своей работы порождает объекты табулированных функций. При этом реальный тип объектов оказывался фиксированным, и изменить его динамически в ходе работы программы было невозможно. Одним из способов решения этой проблемы является применение объектов фабрик с возможностью их замены.

Поскольку здесь требуется порождение только одного объекта, воспользовалась паттерном «Фабричный метод».
В пакете functions описала базовый интерфейс фабрик табулированных функций TabulatedFunctionFactory. Интерфейс объявляет три перегруженных метода TabulatedFunction createTabulatedFunction(), параметры которых соответствуют параметрам конструкторов классов табулированных функций.

```
package functions;

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

Для удобства описала в
классах ArrayTabulatedFunction и LinkedListTabulatedFunction классы фабрик ArrayTabulatedFunctionFactory и LinkedListTabulatedFunctionFactory, реализующие интерфейс фабрики и порождающие объекты соответствующих классов табулированных функций. Сделала классы фабрик вложенными и публичными.

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

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

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

```
    }

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

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

В классе TabulatedFunctions объявила приватное статическое поле типа TabulatedFunctionFactory и проинициализировала его объектом одного из описанных классов фабрик. Также объявила метод setTabulatedFunctionFactory(), позволяющий заменить объект фабрики. Ещё в классе TabulatedFunctions описала три перегруженных метода TabulatedFunction createTabulatedFunction(), возвращающих объекты табулированных функций, созданные с помощью текущей фабрики. Параметры методов соответствуют параметрам методов фабрики.

```
// Фабрика по умолчанию
private static TabulatedFunctionFactory factory = new
ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();

// Метод для установки фабрики
public static void
setTabulatedFunctionFactory(TabulatedFunctionFactory factory) {
    TabulatedFunctions.factory = factory;
}

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

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

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

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

```
public static TabulatedFunction tabulate(Function function,
double leftX, double rightX, int pointsCount) {
    if (leftX < function.getLeftDomainBorder() || rightX >
function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы выходят за
область определения функции");
    }
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Количество точек
должно быть не менее 2");
    }
    if (leftX >= rightX) {
        throw new IllegalArgumentException("Левая граница должна
быть меньше правой");
    }

    // Вызов фабрики вместо создания объекта
    TabulatedFunction tabulatedFunc =
createTabulatedFunction(leftX, rightX, pointsCount);

    // Заполнение значениями функции
    double step = (rightX - leftX) / (pointsCount - 1);
    for (int i = 0; i < pointsCount; i++) {
        double x = leftX + i * step;
        double y = function.getFunctionValue(x);
        tabulatedFunc.setPointY(i, y);
    }

    return tabulatedFunc;
}
```

Также в методах inputTabulatedFunction и readTabulatedFunction заменила retu rn new ArrayTabulatedFunction(points) на:

```
return createTabulatedFunction(points);
```

В методе main() проверила работу фабрик следующим образом.

```
System.out.println("\nЗадание 2");

Function f = new Cos();
TabulatedFunction tf;

// Первое создание - ArrayTabulatedFunction
tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
System.out.println("Default factory: " + tf.getClass());

// Замена на LinkedList фабрику
TabulatedFunctions.setTabulatedFunctionFactory(
        new
```

```
LinkedListTabulatedFunction.LinkedListTabulatedFunctionFactory()
);
tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
System.out.println("LinkedList factory: " + tf.getClass());

// Возврат на Array фабрику
TabulatedFunctions.setTabulatedFunctionFactory(
        new
ArrayTabulatedFunction.ArrayTabulatedFunctionFactory());
tf = TabulatedFunctions.tabulate(f, 0, Math.PI, 11);
System.out.println("Array factory: " + tf.getClass());
```

Результат:
Задание 2
Default factory: class functions.ArrayTabulatedFunction
LinkedList factory: class functions.LinkedListTabulatedFunction
Array factory: class functions.ArrayTabulatedFunction

## Задание 3

Ещё одним инструментом, позволяющим указывать типы порождаемых объектов, является рефлексия.

В классе TabulatedFunctions добавила ещё три перегруженных версии метода createTabulatedFunction(). Их параметры повторяют параметры трёх аналогичных методов, основанных на использовании фабрики, но также эти методы получают ссылку типа Class на описание класса, объект которого требуется создать. Сделала так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс TabulatedFunction. Новые методы создания объектов находят 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 передаются фактические параметры) создан объект табулированной функции. Ссылка на этот объект и возвращена из метода создания.
Если в ходе выполнения рефлексивных операций возникло исключение (не найден конструктор и т.д.), оно отловлено (использовала блок try с отловом нескольких типов исключений). Вместо него выброшено исключение IllegalArgumentException, причём в его конструктор передается отловленное исключение из рефлексии. Это позволяет в случае возникновения ошибок определить реальную причину ошибки.
В классе TabulatedFunctions перегрузила методы, создающие объекты табулированных функций, добавив версии, принимающие также ссылку типа Class на описание класса, объект которого требуется создать. Сделала

так, чтобы в эти методы можно было передать только ссылки на классы, реализующие интерфейс TabulatedFunction.

```
// Методы с рефлексией
public static TabulatedFunction createTabulatedFunction(Class<?>
functionClass,
                                                         double
leftX, double rightX, int pointsCount) {
    // Проверка, что класс реализует TabulatedFunction
    if
(!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Class must implement
TabulatedFunction interface");
    }

    try {
        // Поиск конструктора с параметрами (double, double,
int)
        Constructor<?> constructor =
functionClass.getConstructor(double.class, double.class,
int.class);
        return (TabulatedFunction)
constructor.newInstance(leftX, rightX, pointsCount);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error creating
tabulated function", e);
    }
}

public static TabulatedFunction createTabulatedFunction(Class<?>
functionClass,
                                                         double
leftX, double rightX, double[] values) {
    if
(!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Class must implement
TabulatedFunction interface");
    }

    try {
        // Поиск конструктора с параметрами (double, double,
double[])
        Constructor<?> constructor =
functionClass.getConstructor(double.class, double.class,
double[].class);
        return (TabulatedFunction)
constructor.newInstance(leftX, rightX, values);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error creating
tabulated function", e);
    }
}

public static TabulatedFunction createTabulatedFunction(Class<?>
```

```
functionClass, FunctionPoint[] points) {
    if
(!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Class must implement
TabulatedFunction interface");
    }

    try {
        // Поиск конструктора с параметрами (FunctionPoint[])
        Constructor<?> constructor =
functionClass.getConstructor(FunctionPoint[].class);
        return (TabulatedFunction)
constructor.newInstance((Object) points);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error creating
tabulated function", e);
    }
}
// Перегруженный метод tabulate с рефлексией
public static TabulatedFunction tabulate(Class<?> functionClass,
                                         Function function,
double leftX, double rightX, int pointsCount) {

    if (leftX < function.getLeftDomainBorder() || rightX >
function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы выходят за
область определения функции");
    }
    if (pointsCount < 2) {
        throw new IllegalArgumentException("Количество точек
должно быть не менее 2");
    }
    if (leftX >= rightX) {
        throw new IllegalArgumentException("Левая граница должна
быть меньше правой");
    }

    // Создание табулированной функции через рефлексию
    TabulatedFunction tabulatedFunc =
createTabulatedFunction(functionClass, leftX, rightX,
pointsCount);

    // Заполнение значениями функции
    double step = (rightX - leftX) / (pointsCount - 1);
    for (int i = 0; i < pointsCount; i++) {
        double x = leftX + i * step;
        double y = function.getFunctionValue(x);
        tabulatedFunc.setPointY(i, y);
    }

    return tabulatedFunc;
}
```

Добавила рефлексию для методов чтения в TabulatedFunctions

```
// Методы чтения с рефлексией
public static TabulatedFunction inputTabulatedFunction(Class<?>
functionClass, InputStream in) throws IOException {
    if
(!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Class must implement
TabulatedFunction interface");
    }

    DataInputStream dataIn = new DataInputStream(in);

    // Чтение количества точек
    int pointsCount = dataIn.readInt();

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

    // Создание объекта через рефлексию
    try {
        Constructor<?> constructor =
functionClass.getConstructor(FunctionPoint[].class);
        return (TabulatedFunction)
constructor.newInstance((Object) points);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error creating
tabulated function", e);
    }
}

public static TabulatedFunction readTabulatedFunction(Class<?>
functionClass, Reader in) throws IOException {
    if
(!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Class must implement
TabulatedFunction interface");
    }

    StreamTokenizer tokenizer = new StreamTokenizer(in);
    // Чтение количества точек
    tokenizer.nextToken();
    int pointsCount = (int) tokenizer.nval;
    // Чтение координат точек
    FunctionPoint[] points = new FunctionPoint[pointsCount];
    for (int i = 0; i < pointsCount; i++) {
        tokenizer.nextToken();
        double x = tokenizer.nval;
```

```
        tokenizer.nextToken();
        double y = tokenizer.nval;

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

    // Создание объекта через рефлексию
    try {
        Constructor<?> constructor =
functionClass.getConstructor(FunctionPoint[].class);
        return (TabulatedFunction)
constructor.newInstance((Object) points);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error creating
tabulated function", e);
    }
}
```

Проверила в методе main() работу методов рефлексивного создания объектов, а также методов класса TabulatedFunctions, использующих создание объектов. Сделала это следующим образом.

```
System.out.println("Задание 3");

TabulatedFunction f3;

f3 = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, 0, 10, 3);
System.out.println(f3.getClass());
System.out.println(f3);

f3 = TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, 0, 10, new double[] {0,
10});
System.out.println(f3.getClass());
System.out.println(f3);

f3 = TabulatedFunctions.createTabulatedFunction(
        LinkedListTabulatedFunction.class,
        new FunctionPoint[] {
                new FunctionPoint(0, 0),
                new FunctionPoint(10, 10)
        }
);
System.out.println(f3.getClass());
System.out.println(f3);

f3 = TabulatedFunctions.tabulate(
        LinkedListTabulatedFunction.class, new Sin(), 0,
Math.PI, 11);
System.out.println(f3.getClass());
System.out.println(f3);
```

Вывод:

Задание 3

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), (0.9424777960769379; 0.8090169943749475), (1.2566370614359172; 0.9510565162951535), (1.5707963267948966; 1.0), (1.8849555921538759; 0.9510565162951536), (2.199114857512855; 0.8090169943749475), (2.5132741228718345; 0.5877852522924732), (2.827433388230814; 0.3090169943749475), (3.141592653589793; 1.2246467991473532E-16)}