Лабораторная работа №3

Задание на лабораторную работу

Лабораторная работа является продолжением лабораторной работы №2 и знакомит с Java и ООП более подробно. В работе рассматриваются возможности создания и обработки исключений, а также чтение из потоков и запись данных в потоки, включая сериализацию. Изучаются такие понятия, как вложенные типы и анонимные классы. Проект пополняется функционалом, позволяющим осуществлять над табулированными функциями элементарные математические операции, а также генерировать производные от функции, которая может быть задана как аналитически, так и табулировано. Рассматриваются такие паттерны проектирования, как Абстрактная Фабрика, Итератор и Декоратор (Обёртка).

Задание 1

X	Перенести класс `Node` из пакета `functions` в класс `LinkedListTabulatedFunction`, сделав его статическим вложенным. Убедиться, что ничего не сломалось, запустив написанные ранее тесты.
Y	В пакете `functions` создать класс `Point`, содержащий только два публичных завершённых поля типа `double`: `x` и `y`. Данный класс будет необходим для того, чтобы предоставлять значения (`x`, `y`) из классов табулированных функций независимо от того, как они хранятся. Создать единственный конструктор, принимающий на вход оба аргумента `x` и `y`.
X	Пройтись по методам классов `ArrayTabulatedFunction` и `LinkedListTabulatedFunction` и исследовать их: что произойдёт, если в аргументе будет некорректное значение? Например, в метод `getX(int index)` у обоих классов теоретически может прийти значение index меньше нуля, или больше (либо равно) длине таблицы – значение такого аргумента может поломать всю логику работы программы. Однако, если в случае `ArrayTabulatedFunction` при некорректном аргументе будет выброшено исключение `ArrayIndexOutOfBoundsException`, так как реализация основана на массиве, то результат работы того же метода у связного списка непредсказуем. Требуется оценить последствия каждого метода, если значение индекса выходит за рамки допустимых значения, и в случае необходимости выбросить исключение `IllegalArgumentException`, В методах `floorIndexOfX()` и, если есть, `floorNodeOfX()` выбрасывать это исключение, если `x` меньше левой границы. В самом начале каждого конструктора запретить иметь длину таблицы меньше 2 точек, а иначе бросать это же исключение (в конструктор исключения передать сообщение о том, что длина меньше минимальной). Теперь можно удалить из интерполяционных методов условие, при котором длина таблицы равна 1. Запустить тесты и убедиться, что всё работает (удалить те, что перестали быть актуальными). Добавить тесты на исключения (см., как тестировать выбрасывание исключений, в другом файле).
Y	Создать в корневом пакете проекта пакет exceptions (рядом с пакетом functions). Создать в нём три класса исключений: `ArrayIsNotSortedException`, `DifferentLengthOfArraysException`, `InterpolationException`. Все исключения должны быть необъявляемые (т.е. наследоваться от класса `RuntimeException`) и иметь по два конструктора – без параметров и с параметром-сообщением, который просто вызывает конструктор своего класса-предка и передаёт в него это сообщение. В классе `AbstractTabulatedFunction` добавить два статических метода: `void checkLengthIsTheSame(double[] xValues, double[] yValues)`

	`void checkSorted(double[] xValues)` Реализовать эти методы. Первый метод должен проверять, одинакова ли длина данных массивов. Если она различна – выбрасывать исключение `DifferentLengthOfArraysException`. Во втором методе необходимо пробежаться по всему массиву `xValues` и проверить, что каждое следующее число больше предыдущего, в противном случае бросить исключение `ArrayIsNotSortedException`. Покрыть методы тестами. Добавить в конструкторы `ArrayTabulatedFunction` и `LinkedListTabulatedFunction` (в те, что на вход принимают два массива чисел) эти две проверки с помощью вызова написанных методов. Подумать с напарником, в каком порядке должны быть проведены проверки (включая ту, что должен добавить он в предыдущем пункте). Убедиться, что после этого тесты не сломались. В реализованных методах интерполяции (те, что с двумя аргументами) добавить проверку, что `x` находится внутри интервала интерполирования (т.е. между тем `x`, что получен по `floorIndex`, и следующим за ним). Если это не выполняется, бросать исключение `InterpolationException`. Покрыть тестами бросание всех исключений.
X или Y	Добавить к интерфейсу `TabulatedFunction` наследование от параметризованного интерфейса `Iterable<Point>`. Реализовать его метод `Iterator<Point> iterator()` и в классе `ArrayTabulatedFunction`, и в классе `LinkedListTabulatedFunction` одинаковым способом: бросанием исключения `UnsupportedOperationException`.

X	Переписать реализацию метода `iterator()` в классе `LinkedListTabulatedFunction` следующим образом: метод должен вернуть объект итератора с помощью создания анонимного класса, реализующего интерфейс `Iterator<Point>` (паттерн проектирования Итератор). Задача итератора – по запросу выдавать следующий элемент в контейнере. Контейнером в данном случае является табулированная функция, а её элементами будут служить точки `Point`. Иными словами, при обращении к итератору он будет возвращать каждую следующую точку табулированной функции.
	Внутри анонимного класса должно быть поле `Node node`, хранящее ссылку на текущий элемент в списке. Изначально node ссылается на head. Каждый раз, когда у итератора вызывается метод `next()`, внутри осуществляется проверка, остался ли хотя бы 1 элемент (`boolean hasNext()`). Если не осталось – итератор должен выбросить исключение `NoSuchElementException`. Иначе – создать объект `Point`, передав туда `x` и `y` из текущего узла `node`. Сдвинуть ссылку `node` на следующий элемент в списке, но, если текущий элемент был последним, можно, например, присвоить `null`. Тогда реализацию метода `hasNext()` можно осуществить с помощью проверки `node` на `null`.
	Добавить тесты, проверяющие работу метода (и, следовательно, итератора), двумя способами:
	1) с помощью цикла `while`:
	`while(iterator.hasNext()) { Point point = iterator.next(); … }`
	2) с помощью цикла for-each:
	`for (Point point : tabulatedFunction) { … }`

Y	Аналогичным образом переписать реализацию метода `iterator()` в классе `ArrayTabulatedFunction`. Отличия будут следующими: вместо поля `Node` в анонимном классе следует хранить индекс `i`. Когда у итератора запрашивают следующий элемент, он (помимо таких же проверок и бросания исключения) создаёт объект `Point` по элементам массивов `xValues[i], xValues[i]`. После

	чего `i` увеличивается на 1. Метод `hasNext()`, очевидно, возвращает `true`, пока `i < count`. Точно так же покрыть метод тестами с применением двух циклов.

X*	Создать класс `StrictTabulatedFunction`, реализующий интерфейс `TabulatedFunction`. Задачей класса является введение запрета на интерполяцию для табулированных функций. Было бы неплохо накладывать такой запрет на уже имеющиеся табулированные функции, чтобы не создавать новые. Но, поскольку у нас имеется уже две различных реализации, наследование здесь не подойдёт. Для динамического подключения дополнительного поведения к объекту без наследования используется паттерн проектирования Декоратор (Обёртка).
	В созданном классе должно быть определено поле `TabulatedFunction function`, хранящее ссылку на объект другой функции. Этот объект должен быть передан в конструкторе. Большинство методов реализовывается за счёт делегирования их поведения соответствующим методам внутреннего объекта `function`. Например:
	`public int getCount() {`
	`  return tabulatedFunction.getCount();`
	`}`
	Требуется реализовать все методы `TabulatedFunction` либо путём делегирования, либо в соответствии с целью запрета интерполяции. В данном случае единственный метод, который не следует делегировать – это `apply()`. Если в классе `AbstractTabulatedFunction` он вызывает методы интерполяции, то здесь их нужно исключить. Понадобиться найти `indexOfX(i)`: если он равен минус `1` (`x` не найден), то требуется бросить `UnsupportedOperationException`. В противном случае вернуть соответствующий ему `y`. Покрыть класс тестами путём оборачивания объектов классов `ArrayTabulatedFunction` и `LinkedListTabulatedFunction`.

Y*	Создать класс `UnmodifiableTabulatedFunction`, реализующий интерфейс `TabulatedFunction`. Задачей класса является внесение запрета на модификацию значений табулированной функции в соответствии с паттерном Декоратор. Все методы интерфейса следует реализовать по аналогии с классом `StrictTabulatedFunction`. Отличием тут будет то, что метод `apply()` вполне может передать своё поведению внутреннему объекту-функции, а вот `setY()` должен бросать исключение `UnsupportedOperationException`. Покрыть класс тестами путём оборачивания объектов классов `ArrayTabulatedFunction` и `LinkedListTabulatedFunction`.
X* и Y*	Добавить тесты, которые проверяют работу классов-обёрток `StrictTabulatedFunction` и `UnmodifiableTabulatedFunction` друг на друге. Т.е. если объект одного класса обернуть в другой (и наоборот), полученный объект не будет позволять ни модифицировать значения функции, ни использовать интерполяцию.

Задание 2

Данное задание предполагает добавление операций и операторов для функций. В качестве операторов выступает дифференциальный оператор, ставящий в соответствие функции её производную. Для реализации производных используются численные методы.

Левой разностной производной функции $f(x)$ называется функция $g(x)$:

$$g(x) = \frac{f(x) - f(x-h)}{h},$$

где h – шаг дифференцирования. Аналогично вычисляются правая разностная производная $h(x)$ и средняя разностная производная $s(x)$:

$$h(x) = \frac{f(x+h) - f(x)}{h},$$
$$s(x) = \frac{f(x+h) - f(x-h)}{2h}.$$

Однако, для табулированных функций удобнее вычислить производную на основе тех точек, которые заданы. Например, если табулированная функция имеет значения x_0, x_1, …, x_{n-1} и соответствующие им y_0, y_1, …, y_{n-1} , то в качестве производной функции можно построить новую табулированную функцию, у которой будут те же x_k, а $\hat{y}_k$ будут вычислены по формуле разностных производных. При этом первые n-1 значений можно определить по формуле правой производной:

$$\hat{y}_k = \frac{y_{k+1} - y_k}{x_{k+1} - x_k},$$

а последнее значение вычисляется по формуле левой производной, т.е. оно такое же, как и предпоследнее.

Операции будут рассматриваться только для табулированных функций и только примитивные: сложение, вычитание, умножение и деление. Каждая такая операция ставит в соответствие двум табулированным функциям одну новую табулированную функцию. Проблема состоит в том, чтобы определить, какую же реализацию должна иметь новая функция: `ArrayTabulatedFunction` или `LinkedListTabulatedFunction`? Можно выбрать одну из них, но в какой-то момент может понадобиться и другая. Что делать в этом случае – писать новый дублирующий функционал? Для того, чтобы переложить эту ответственность на программиста, вызывающего эти методы, а не реализующего их, можно использовать паттерн проектирования Абстрактная Фабрика.

X	Создать пакет `factory` внутри пакета `functions`. В нём создать интерфейс `TabulatedFunctionFactory` с единственным методом: `TabulatedFunction create(double[] xValues, double[] yValues)` Создать классы `ArrayTabulatedFunctionFactory` и `LinkedListTabulatedFunctionFactory`, реализующие этот интерфейс путём возвращения нового объекта соответствующего класса (`ArrayTabulatedFunction` или `LinkedListTabulatedFunction`) и передачей массивов-аргументов в его конструктор. Покрыть классы тестами путём проверки, будет ли удовлетворять соответствующему типу созданные объекты.
Y	Создать пакет `operations` внутри основного пакета проектов (рядом с `functions` и `exceptions`). Внутри пакета создать класс `TabulatedFunctionOperationService`, предоставляющий сервис для работы с табулированными функциями и их операции. Добавить в класс публичный статический метод: `Point[] asPoints(TabulatedFunction tabulatedFunction)` Метод должен осуществлять преобразование табулированной функции в массив точек. Для этого необходимо пробежаться по функции с помощью цикла for-each и записывать каждую следующую точку в массив. Отдельно придётся озаботиться инициализацией и инкрементированием индекса `i`, который не прописывается в заголовке цикла for-each. Покрыть метод тестами.
X	Перейти к созданию интерфейса дифференциального оператора. Произвольный дифференциальный оператор (`DifferentialOperator`) может быть применён как к произвольным функциям, так и к табулированным в частности. Но для табулированных функций можно написать ещё особый дифференциальный

	оператор (`TabulatedDifferentialOperator`), который будет возвращать именно табулированную функцию. Если у таких операторов будет общий интерфейс, то метод в обоих случаях должен будет вернуть тип `MathFunction`, и сузить тип будет нельзя – т.е. программист, применивший оператор `TabulatedDifferentialOperator`, должен будет использовать приведение типа, чтобы получить ссылку типа `TabulatedFunction` из `MathFunction`. А перед приведением типа – ещё и проверку `instanceof`, чтобы гарантировать отсутствие ошибки. Проблему можно обойти, воспользовавшись параметризацией типов. В пакете operations создать интерфейс `DifferentialOperator`, параметризованный типом `T`, который наследуется от типа `MathFunction`. В интерфейсе определить единственный метод, возвращающий производную входной функции: `T derive(T function)` Создать класс `TabulatedDifferentialOperator`, реализующий интерфейс `DifferentialOperator` и подставляющий вместо параметра `T` тип `TabulatedFunction`. Класс должен хранить ссылку на объект фабрики `TabulatedFunctionFactory factory`, которая передаётся в конструкторе. Также должен быть конструктор без аргументов, определяющий в качестве фабрики объект класса `ArrayTabulatedFunctionFactory`. Для фабрики должны быть определены геттер и сеттер. Реализовать метод `derive()`. Для начала неплохо получить все точки входной функции, используя метод `TabulatedFunctionOperationService.asPoints()`, и создать массивы `xValues` и `yValues` такой же длины. Затем определить значения этих массивов с помощью численного дифференцирования для табулированных функций, описанного выше. После этого создать новый экземпляр табулированной функции, вызвав соответствующий метод у фабрики. Протестировать написанные методы на простых табулированных функциях, у которых численная производная может быть легко вычислена. Во время тестирования использовать разные типы фабрик.
Y	Прежде чем перейти к реализации бинарных операций над табулированными функциями, необходимо определить, а к каким функциям они могут быть применимы. Очевидно, что количество записей в таблице должно быть одинаковым, и все x_k должны иметь одинаковые соответствующие значения. Иначе, например, складывать такие функции нельзя. В пакете `exceptions` создать класс `InconsistentFunctionsException` по аналогии с другими исключениями в этом пакете (тоже с двумя конструкторами). Добавить в начало класса `TabulatedFunctionOperationService` поле фабрики `TabulatedFunctionFactory factory`, задающееся с помощью конструктора. Второй конструктор должен быть без параметров, а в реализации поле `factory` должно инициализироваться объектом класса `ArrayTabulatedFunctionFactory`. Добавить геттер и сеттер для фабрики. После метода `asPoints()` нужно будет написать методы сложения и вычитания двух функций. Реализация методов описывается следующим образом (сначала просто прочитать). Вычислить количество записей в первой табулированной функции, сравнить его с количеством тех, что во второй. Если это не одно и то же число, то выбросить исключение `InconsistentFunctionsException`. Получить значения обеих функций в виде двух массивов точек `Point` с помощью метода `asPoints()`. Создать массивы `xValues` и `yValues`, в которые будут записываться результаты применения операции. Затем необходимо добавить цикл, пробегающийся по всем

	записям таблиц. В массив `xValues[i]` записывается значение соответствующего `x` из массива точек первой или второй функции. Если это значение у функций не совпадает, нужно бросить исключение `InconsistentFunctionsException`. В массив `yValues[i]` нужно записать результат операции сложения или вычитания (в зависимости от метода) соответствующих `y` из массива точек первой и второй функции. После цикла необходимо создать новый объект функции с помощью хранящейся в поле фабрики и передачей в метод созданных массивов. У такого подхода есть существенный недостаток – для двух операций сложения и умножения соответствующие методы будут различаться лишь одной строчкой: там, где эта операция выполняется. Весь остальной код будет дублирующим. Что будет, если понадобится добавить ещё операции умножения и деления? Для устранения недостатка можно воспользоваться возможностями ООП: выделить всю логику во вспомогательный метод, передать в качестве параметра метода операцию, которую нужно будет совершить со значениями `y`, и применить её там, где нужно. Создать вложенный в классе `TabulatedFunctionOperationService` приватный интерфейс `BiOperation` с единственным методом: `double apply(double u, double v)` Метод ставит в соответствие двум числам третье. Это может быть как сложение, так и вычитание. Так как каждая операция понадобится для передачи лишь один раз, для реализации интерфейса можно будет использовать анонимные классы. Создать в классе `TabulatedFunctionOperationService` недоступный другим классам метод `TabulatedFunction doOperation(TabulatedFunction a, TabulatedFunction b, BiOperation operation)`, принимающий на вход две табулированные функции и возвращающий новую табулированную функцию, полученную в результате применением переданной третьим аргументом операции поэлементно к их значениям `y`. Реализовать метод так, как было описано ранее. Добавить публичные методы сложения и вычитания, которые делегируют своё поведение методу `doOperation()`, передавая в качестве третьего параметра объект анонимного класса, реализующего интерфейс `BiOperation`. Примечание: в данном случае анонимный класс можно заменить на лямбда-выражение. Покрыть доступные методы тестами. Во время тестирования использовать разные типы фабрик. Также среди тестов должно быть сложение (или вычитание) функции одного типа и функции другого типа (например, `LinkedListTabulatedFunction + ArrayTabulatedFunction` или `ArrayTabulatedFunction – LinkedListTabulatedFunction`).
X	Добавить в класс `TabulatedFunctionOperationService` методы умножения и деления двух функций по аналогии с тем, как это сделано для сложения и вычитания. Аналогично покрыть методы тестами.
Y	Добавить в пакет operations абстрактный класс `SteppingDifferentialOperator`, реализующий интерфейс `DifferentialOperator` и подставляющий вместо параметра `T` тип `MathFunction`. Этот класс будет являться базовым для дифференциальных операторов различных функций с предопределённым шагом дифференцирования. Класс должен иметь защищённое поле `double step`, определяющееся в единственном конструкторе с таким же параметром. У поля должны быть геттер и сеттер. В конструкторе провести проверку, что если `step` отрицательный, либо равен нулю, положительной бесконечности или `NaN`, то выбросить исключение `IllegalArgumentException`. Создать класс `LeftSteppingDifferentialOperator`, который наследуется от класса `SteppingDifferentialOperator`. У него должен быть аналогичный

	конструктор, в котором происходит обращение к конструктору родителя. Переопределённый метод `derive()` должен возвращать объект анонимного класса, реализующего интерфейс `MathFunction` и представляющий собой функцию-производную. Метод `apply()` анонимного класса должен быть определён так, чтобы вычислялась левая разностная производная у функции, переданной в аргумент `derive()`. Аналогично добавить класс `RightSteppingDifferentialOperator` для правой разностной производной. При желании добавить класс `MiddleSteppingDifferentialOperator` для средней разностной производной. Покрыть неабстрактные классы тестами, используя простой шаг и простые функции, у которых разностную производную легко вычислить вручную (например, `SqrFunction`).
X*	Добавить абстрактной фабрике `TabulatedFunctionFactory` метод: `TabulatedFunction createStrict(double[] xValues, double[] yValues)` Метод должен иметь реализацию по умолчанию. В качестве возвращаемого значения должна быть новая табулированная функция, обёрнутая в `StrictTabulatedFunction`. Покрыть метод тестами для обеих реализаций интерфейса.
Y*	Добавить абстрактной фабрике `TabulatedFunctionFactory` метод: `TabulatedFunction createUnmodifiable (double[] xValues, double[] yValues)` Метод должен иметь реализацию по умолчанию. В качестве возвращаемого значения должна быть новая табулированная функция, обёрнутая в `UnmodifiableTabulatedFunction`. Покрыть метод тестами для обеих реализаций интерфейса.
X* или Y*	Добавить абстрактной фабрике `TabulatedFunctionFactory` метод: `TabulatedFunction createStrictUnmodifiable (double[] xValues, double[] yValues)` Метод должен иметь реализацию по умолчанию. В качестве возвращаемого значения должна быть новая табулированная функция, обёрнутая и в `UnmodifiableTabulatedFunction`, и в `StrictTabulatedFunction` (в любом порядке). Покрыть метод тестами. Убедиться, что полученная из метода функция действительно обладает свойствами обеих обёрток.

Задание 3

Y	Создать в основном пакете проекта пакет `io` (рядом с пакетами `exceptions`, `functions` и `operations`). Создать в этом пакете завершённый класс `FunctionsIO`, который не может иметь наследников и экземпляров. Для этого недостаточно объявить его `final` – нужно создать приватный конструктор, бросающий исключение `UnsupportedOperationException`.
Y	Создать в директории проекта папки `input` и `output` (рядом с файлом `pom.xml`). Папка output НЕ должна попадать в репозиторий – можно добавить её в файл «`.gitignore`». Папку `input` можно добавить туда же, а можно и не добавлять.
X	В классе `AbstractTabulatedFunction` переопределить метод `toString()` класса `Object` для того, чтобы строковое представление функции было таковым: первая строка – название класса и через пробел указание размера count, следующие строки – перечисление её точек, каждая точка на новой строке в квадратных скобках, причём `x` и `y` отделены друг от друга точкой с запятой и пробелом. Пример:

	`LinkedListTabulatedFunction size = 3`
	`[0.0; 0.0]`
	`[0.5; 0.25]`
	`[1.0; 1.0]`
	Для реализации необходимо создать экземпляр класса `StringBuilder` и использовать его метод `append()` для добавления новых элементов к строке. Для получения имени класса можно использовать метод `getClass()` класса `Object`, а у полученного класса – метод `getSimpleName()`. Для переноса строк можно использовать экранированный символ «`\n`». Для обхода всех точек необходимо использовать цикл for-each. Для получения результата необходимо вызывать у экземпляра `StringBuilder` метод `toString()`. Покрыть метод тестами (в тесте такая строка должна быть написана без каких-либо конкатенаций, за исключением случаев, когда строка очень длинная). Пример строки для теста:
	`LinkedListTabulatedFunction size = 3\n[0.0; 0.0]\n[0.5; 0.25]\n[1.0; 1.0]`

Y	В классе `FunctionsIO` добавить статический метод: `void writeTabulatedFunction(BufferedWriter writer, TabulatedFunction function)` Задача метода – записать представление функции function в буферизованный символьный поток `writer`. Формат записи должен быть следующим: первая строчка – число точек функции; каждая следующая строка – перечисление значений всех этих точек через пробел (одна строка – одна пара значений `x` и `y`). Для записи необходимо будет обернуть поток в `PrintWriter` и использовать его метод `println()` для записи числа точек `count`, а затем в цикле for-each метод `printf()` для форматированного вывода дробных чисел. Строка вывода в методе должна быть такой «`%f %f\n`». Вторым и третьим параметром в методе должны быть x и y текущей точки в цикле. Поток записи не должен быть закрыт в конце метода, так как не он его создал! Вместо этого для проброса всех данных из буфера следует использовать его метод `flush()`. Создать в пакете `io` класс `TabulatedFunctionFileWriter` с main-методом. Внутри метода должно создаваться два файловых символьных потока записи `FileWriter` с использованием одной конструкции try-with-resources. Первый поток должен указывать на файл «`output/array function.txt`», второй – «`output/linked list function.txt`». Оба потока следует обернуть в буферизованные потоки. Создать две табулированные функции – с реализацией в виде массива и с реализацией в виде связного списка. Записать каждую функцию в соответствующий поток с помощью реализованного ранее метода. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). Запустить метод, убедиться, что файлы появились в директории `output` и их содержание корректно.
X	Из двух папок `input` и `output` создать для себя те, что не попали в репозиторий (и не должны попасть).
X	В классе `FunctionsIO` добавить статический метод: `void writeTabulatedFunction(BufferedOutputStream outputStream, TabulatedFunction function)` Задача метода – записать представление функции `function` в буферизованный байтовый поток `outputStream`. Так как поток байтовый, не имеет смысла форматировать данные.

	Для записи необходимо будет обернуть поток в `DataOutputStream`. Первым нужно записать число значений таблицы `count`, а далее пробежаться по всем точкам функции с помощью цикла for-each и записать в поток все `x` и `y`. Для записи длины следует использовать метод `writeInt()`, для записи значений точек – `writeDouble` (методы обёртки `DataOutputStream`). Поток записи не должен быть закрыт в конце метода, так как не он его создал! Вместо этого для проброса всех данных из буфера следует использовать его метод `flush()`. Метод не должен обрабатывать исключение `IOException` – он должен добавить его в своё объявление. Создать в пакете `io` класс `TabulatedFunctionFileOutputStream` с main-методом. Внутри метода должно создаваться два файловых байтовых потока записи `FileOutputStream` с использованием одной конструкции try-with-resources. Первый поток должен указывать на файл «`output/array function.bin`», второй – «`output/linked list function.bin`». Оба потока следует обернуть в буферизованные потоки. Создать две табулированные функции – с реализацией в виде массива и с реализацией в виде связного списка. Записать каждую функцию в соответствующий поток с помощью реализованного ранее метода. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). Запустить метод, убедиться, что файлы появились в директории `output`. Попытаться их открыть.
Y	В классе `FunctionsIO` добавить статический метод: `TabulatedFunction readTabulatedFunction(BufferedReader reader, TabulatedFunctionFactory factory)` Задача метода – прочитать данные из буферизованного символьного потока `reader` и на их основе создать новую функцию с помощью фабрики `factory`. Формат файла должен быть следующим: первая строчка – число точек функции; каждая следующая строка – перечисление значений всех этих точек через пробел (одна строка – одна пара значений `x` и `y`). Значения `x` и `y` – числа с плавающей точкой, но разделителем целой и дробной части выступает не точка, а запятая. Файл можно читать построчно с помощью метода `readLine()` у `reader`. В первой строке лежит целое число, поэтому его следует извлечь из строки при помощи метода `Integer.parseInt()`. Полученное число будет количеством значений функции, поэтому сразу можно будет создать массивы `xValues` и `yValues` заданной длины. Помимо массивов понадобится объект форматирования дробных чисел, чтобы корректно читать данные с разделителями-запятыми – `NumberFormatter`. Получить такой объект можно с помощью статического метода этого класса `getInstance(…)`. В параметр метода передать русскую локализацию с помощью вызова `Locale.forLanguageTag("ru")`. После этого в цикле нужно будет бежать по строкам файла (`count` раз), считывать их (как это было сделано для первой строки) и извлекать из них значения. После чтения строку нужно разбить на две по пробелу – это легко сделать при помощи метода `split()` и передачей в него пробела. Он должен будет вернуть массив из двух строк – что было до пробела и после. Затем эти строки надо перевести в числа и записать в `xValues[i]` и `yValues[i]` соответственно. Для этого у объекта форматирования нужно использовать метод `parse()`, который вернёт объект-обёртку примитивного типа `Number`, у которого, в свою очередь, нужно вызвать метод `doubleValue()`. Важно: метод `parse()` может выбросить исключение `ParseException`. Наш

	метод не должен бросать это исключение! Его следует обработать в блоке `catch` конструкции try-catch следующим образом: при поимке оборачивать его в `IOException` и бросать дальше. У текущего метода в объявлении должно быть добавлено это исключение. После создания массивов получить и вернуть саму функцию, воспользовавшись методом фабрики. Создать в директории `input` файл `function.txt` и наполнить его содержимым по вышеописанному формату. Создать в пакете `io` класс `TabulatedFunctionFileReader` с main-методом. Внутри метода должно создаваться два файловых символьных потока чтения `FileReader` с использованием одной конструкции try-with-resources. Оба потока должны брать данные из файла «`input/function.txt`» и должны быть обёрнуты в буферизованные потоки. С помощью реализованного ранее метода требуется получить объекты функций `ArrayTabulatedFunction` и `LinkedListTabulatedFunction`, передавая в метод соответствующие фабрики. Вывести в консоль по очереди обе функции, используя их метод `toString()`. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`).
X	В классе `FunctionsIO` добавить статический метод: `TabulatedFunction readTabulatedFunction(BufferedInputStream inputStream, TabulatedFunctionFactory factory)` Задача метода – прочитать данные из буферизованного байтового потока `inputStream` и на их основе создать новую функцию с помощью фабрики `factory`. Данные в потоке передаются следующим образом: число `int`, задающее размер табулированной функции, после которого идут пары чисел `x` и `y` типа `double`, описывающие точки функции. Количество точек равно первому числу. При реализации в первую очередь следует обернуть входной поток в `DataInputStream`. С помощью метода `readInt()` (у объекта полученного потока) следует считать длину, после чего создать массивы `xValues` и `yValues` с этой длиной. Затем в одном цикле необходимо пробежаться сразу по обоим массивам и записать в них данные из потока с помощью метода `readDouble()`. После этого можно воспользоваться методом фабрики для создания новой функции и её возвращения. В директорию `input` скопировать один из бинарных файлов, записанных ранее в директорию `output`. Переименовать его в «`binary function.bin`». Создать в пакете `io` класс `TabulatedFunctionFileInputStream` с main-методом. Внутри метода должен создаваться один файловый байтовый поток чтения `FileInputStream` с использованием конструкции try-with-resources. Поток должен получать данные из файла «`input/binary function.bin`» и должен быть обёрнут в буферизованный поток. С помощью реализованного ранее метода требуется получить объект функции типа `ArrayTabulatedFunction`, передавая в метод соответствующий объект фабрики. После этого необходимо вывести в консоль функцию, используя метод `toString()`. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). Далее в этом же методе требуется считать ещё одну функцию из консоли. Для этого необходимо передать поток `System.in` в `InputStreamReader` и обернуть полученный объект в буферизованный поток.

	В консоли должно отобразиться сообщение «Введите размер и значения функции», после чего следует с помощью написанного ранее метода считать функцию из консоли. Функция должна иметь тип `LinkedListTabulatedFunction`, для чего в метод необходимо передать соответствующую фабрику. Затем требуется вывести в консоль производную считанной функции, полученную с помощью оператора `TabulatedDifferentialOperator`. Для вывода использовать метод `toString()`. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). Важно: в данном случае запрещено использовать конструкцию try-with-resources, так как она закрывает потоки, а поток `System.in` закрываться не должен. Запустить метод и добиться работоспособности считывания табулированной функции из консоли.
Y	Подготовить класс `ArrayTabulatedFunction` к сериализации. У всех сериализуемых классов должно быть явно задано поле `serialVersionUID`, которое должно быть сгенерировано средствами среды разработки (см. описание в другом файле).
X	Подготовить класс `LinkedListTabulatedFunction` к сериализации. У всех сериализуемых классов должно быть явно задано поле `serialVersionUID`, которое должно быть сгенерировано средствами среды разработки (см. описание в другом файле).
Y	Добавить в класс `FunctionsIO` статический метод: `void serialize(BufferedOutputStream stream, TabulatedFunction function)` Метод должен записывать в буферизованный байтовый поток `stream` сериализованную функцию `function`. Для этого поток следует обернуть в `ObjectOutputStream` и использовать метод `writeObject()`. Поток записи не должен быть закрыт в конце метода, так как не он его создал! Вместо этого для проброса всех данных из буфера следует использовать его метод `flush()`. Метод не должен обрабатывать исключение `IOException` – он должен добавить его в своё объявление.
X	Добавить в класс `FunctionsIO` статический метод: `TabulatedFunction deserialize(BufferedInputStream stream)` Метод должен считать из буферизованного байтового потока сериализованную функцию `TabulatedFunction`, т.е. десериализовать её. Для этого поток следует обернуть в `ObjectInputStream` и считать объект функции с помощью метода `readObject()` (а затем выполнить приведение типа). Метод не должен обрабатывать исключения `IOException` и `ClassNotFoundException` – он должен добавить их в своё объявление.
Y	Создать в пакете `io` класс `ArrayTabulatedFunctionSerialization` с main-методом. Внутри метода должен создаваться файловый байтовый поток записи `FileOutputStream` с использованием конструкции try-with-resources. Адрес записи – «`output/serialized array functions.bin`». Поток следует обернуть в буферизованный. Создать табулированную функцию типа `ArrayTabulatedFunction`. Найти её первую и вторую производные (такого же типа) с помощью `TabulatedDifferentialOperator`. Это будут ещё две функции. Произвести сериализацию всех трёх функций в созданный поток с помощью метода `FunctionsIO.serialize()`.

	Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). После записи и закрытия потока должен создаваться файловый байтовый поток записи `FileOutputStream`, адрес которого совпадает с предыдущим. Требуется обернуть поток в буферизованный, а затем произвести десериализацию всех трёх функций из созданного ранее файла. Это поможет сделать метод `FunctionsIO.deserialize()`. Необходимо вывести значения всех функций в консоль при помощи метода `toString()`. Исключения `IOException` и `ClassNotFoundException` должны быть пойманы и обработаны путём передачи стектрейса в поток ошибок (`printStackTrace()`).
X	Создать в пакете `io` класс `LinkedListTabulatedFunctionSerialization` с main-методом. Внутри метода должен создаваться файловый байтовый поток записи `FileOutputStream` с использованием конструкции try-with-resources. Адрес записи – «`output/serialized linked list functions.bin`». Поток следует обернуть в буферизованный. Создать табулированную функцию типа `LinkedListTabulatedTabulatedFunction`. Найти её первую и вторую производные (такого же типа) с помощью `TabulatedDifferentialOperator`. Это будут ещё две функции. Произвести сериализацию всех трёх функций в созданный поток с помощью метода `FunctionsIO.serialize()`. Исключение `IOException` должно быть поймано и обработано путём передачи стектрейса в поток ошибок (`printStackTrace()`). После записи и закрытия потока должен создаваться файловый байтовый поток записи `FileOutputStream`, адрес которого совпадает с предыдущим. Требуется обернуть поток в буферизованный, а затем произвести десериализацию всех трёх функций из созданного ранее файла. Это поможет сделать метод `FunctionsIO.deserialize()`. Необходимо вывести значения всех функций в консоль при помощи метода `toString()`. Исключения `IOException` и `ClassNotFoundException` должны быть пойманы и обработаны путём передачи стектрейса в поток ошибок (`printStackTrace()`).
X* и Y*	Покрыть тестами методы класса `FunctionsIO`. Для этого создать рядом с директориями `input` и `output` директорию `temp`. Эта директория может хранится в системе контроля версий, а вот файлы внутри неё не должны туда попадать. В неё можно записывать файлы, которые порождаются тестами. Самый очевидный способ тестирования – записывать во временные файлы информацию, тестируя методы записи. Для методов чтения придётся сначала воспользоваться методами записи, а уже потом непосредственно тестировать само чтение. Независимо от того, успешно или неуспешно завершился тест, папка temp должна быть очищена. Для очистки директории можно написать отдельный метод в тестирующем классе и добавить ему аннотацию `@AfterClass` (из библиотеки `testng`).
Y*	Для класса `ArrayTabulatedFunction` добавить возможность xml-сериализации. Для этого можно воспользоваться сторонней библиотекой «`XStream`» (добавить зависимости в `pom.xml`). Добавить в класс `FunctionsIO` статический метод: `void serializeXml(BufferedWriter writer, ArrayTabulatedFunction function)` Он должен обеспечивать функционал записи функции `function` в поток `writer` в формате xml. Использовать её следует по аналогии с тем, как это было осуществлено для других методов этого класса.

	Для записи требуется создать объект `XStream`, а затем вызвав у этого объекта метод `toXml()`, передав в него функцию. Полученную строку и нужно записать в поток `writer`. Добавить в класс `FunctionsIO` статический метод: `ArrayTabulatedFunction deserializeXml(BufferedReader reader)` Метод должен осуществлять xml-десериализацию из потока `reader`. Для этого так же нужно создать объект `XStream` и использовать его метод `fromXML()`. Создать соответствующий класс с main-методом для осуществления чтения/записи. При желании покрыть написанные методы тестами.

X*	Для класса `ArrayTabulatedFunction` добавить возможность json-сериализации. Для этого можно воспользоваться сторонней библиотекой «`FasterXML/jackson`» (в зависимости `pom.xml` придётся добавить три различных `artifactId` для `groupId com.fasterxml.jackson.core: jackson-core`, `jackson-annotations` и `jackson-databind`).
	После этого класс `ArrayTabulatedFunction` нужно будет подготовить в json-сериализации:
	1) Добавить у полей `xValues` и `yValues` аннотации `@JsonFormat(shape = JsonFormat.Shape.ARRAY)`, чтобы эти поля записывались как массивы.
	2) У конструктора с аргументами-массивами `xValues` и `yValues` добавить аннотацию `@JsonCreator`, чтобы при десериализации вызывался именно этот конструктор.
	3) У аргументов-массивов этого конструктора добавить аннотации `@JsonProperty(value = "xValues")` и `@JsonProperty(value = "yValues")` соответственно, чтобы при десериализации в конструктор передавались значения, записанные в этих полях.
	Добавить в класс `FunctionsIO` статический метод:
	`void serializeJson(BufferedWriter writer, ArrayTabulatedFunction function)`
	Он должен обеспечивать функционал записи функции `function` в поток `writer` в формате json. Использовать её следует по аналогии с тем, как это было осуществлено для других методов этого класса.
	Для записи требуется создать объект `ObjectMapper`, а затем вызвав у этого объекта метод `writeValueAsString()`, передав в него функцию. Полученную строку и нужно записать в поток `writer`.
	Добавить в класс `FunctionsIO` статический метод:
	`ArrayTabulatedFunction deserializeJson(BufferedReader reader)`
	Метод должен осуществлять json-десериализацию из потока `reader`. Для этого так же нужно создать объект `ObjectMapper`. У созданного объекта требуется вызвать метод `readerFor()`, передав туда класс `ArrayTabulatedFunction.class`, а у полученного объекта – метод `readValue()`, куда следует передать `reader`. Вернуть полученную функцию.
	Создать соответствующий класс с main-методом для осуществления чтения/записи.
	При желании покрыть написанные методы тестами.