Лабораторная работа №2

Задание на лабораторную работу

Разработать набор классов и интерфейсов для работы с математическими функциями. Функции могут быть задано как аналитически (с помощью формулы), так и с помощью таблицы. В последнем случае необходимо обеспечить поддержку получения значений функции, которые не были заданы таблично, с помощью интерполяции и экстраполяции.

Примечание: здесь и в следующих лабораторных работах предполагается использование собственного проекта, среды разработки, систем автоматического тестирования (см. практические задания, нужно подключить библиотеку тестов до начала работы) и совместное выполнение лабораторных вдвоём. Перед началом студент должен определить, какие задания ему выполнять – X или Y. Выбор осуществляется сразу для всех лабораторных работ. Каждая следующая лабораторная работа является продолжением предыдущей. Если студент делает лабораторную работу в одиночку, то он выполняет и X, и Y. Если задание помечено *, то оно является дополнительным, т.е. необязательным к выполнению, но позволяющее набрать больше баллов и получить больше опыта. Преподаватель может (и, скорее всего, будет) давать дополнительные индивидуальные задания, не приведённые здесь, во время проверки. Производить «коммиты» изменений необходимо не реже, чем после каждой выполненной строчки таблицы (в заданиях). Многие задания можно выполнять параллельно, поэтому не всегда обязательно совершать «коммиты» в том же порядке. Иными словами, не всегда один студент обязан ждать другого, чтобы произвести «коммит». Тем не менее, оба студента должны изучить все задания, чтобы понимать, что происходит на проекте и какие изменения могут произойти.

Задание 1

X или Y	Внутри созданного проекта (см., как настроить рабочее окружение и создать проект – для лабораторных) добавить пакет functions. Все классы и интерфейсы, созданные в рамках данной лабораторной, должны находиться в этом пакете. Создать интерфейс MathFunction, содержащий единственный метод apply(double x), принимающий на вход и возвращающий число с плавающей точкой. Так как это интерфейс, метод не должен содержать никакой реализации и будет автоматически являться публичным и абстрактным, поэтому будет лишним использовать соответствующие модификаторы.

X	Добавить класс IdentityFunction, реализующий интерфейс MathFunction, объекты которого должны выполнять тождественное преобразование, т.е. для каждого x метод apply() должен возвращать этот же x. С помощью комбинации клавиш Alt+Insert добавить тестирующий класс и покрыть класс тестами (в дальнейшем большинство новых классов тестов нужно создавать таким же образом).
Y	Добавить класс SqrFunction, реализующий интерфейс MathFunction следующим образом: для каждого x метод apply() должен возвращать квадрат x с помощью обращения к классу java.lang.Math. Покрыть класс тестами.
X и Y	Добавить ещё по одному классу математических функций, реализующему интерфейс MathFunction. Какую именно функцию добавить, необходимо уточнить у преподавателя. Покрыть классы тестами.
X	Добавить класс CompositeFunction, реализующий интерфейс MathFunction, объекты которого будут играть роль сложных математических функций. Такая функция должна применять к аргументу сначала одну «более простую» функцию, а затем другую – к результату первой функции. Важно понимать, что этими двумя функциями может быть одна и та же «более простая» функция. Более того, «более простой» функцией может оказаться сложная функция, т.е. объект класса CompositeFunction. Математически: h(x)=g(f(x)), где – h(x) сложная функция, f(x) – первая функция, может быть как простой, так и сложной, g(x) – вторая функция, может быть как простой, так и сложной. Именно в таком порядке, так как сначала к x будет применена f, а затем к результату – g. Для обеспечения такой функциональности необходимо, чтобы класс CompositeFunction имел два поля типа MathFunction. Нет смысла делать эти поля доступными извне, достаточно один раз задать их в конструкторе. Таким образом, поля должны быть приватными. После задания полей (firstFunction и secondFunction) и единственного публичного конструктора (с двумя аргументами) необходимо реализовать метод apply() так, как было описано выше. Покрыть этот класс тестами, используя в том числе другие классы функций. Обязательно в тестах проверить работу сложных функций от сложных функций.

Y	Добавить класс ConstantFunction, реализующий интерфейс MathFunction. Объекты этого класса должны всегда возвращать одно и то же число, не зависимо от того, какой x приходит в качестве аргумента метода apply(). Это число должно задаваться в публичном конструкторе в качестве аргумента. Поле, хранящее заданную константу, должно быть приватным и завершённым, чтобы не было желания его изменить. По желанию можно добавить публичный метод-геттер для этой константы. Добавить классы ZeroFunction и UnitFunction, расширяющие класс ConstantFunction, имеющие только по одному конструктору без аргументов, никаких новых полей и методов. Этот конструктор должен быть реализован так, чтобы метод apply() возвращал только 0 и 1 соответственно для ZeroFunction и UnitFunction. Покрыть все 3 класса тестами.
Y	Добавить в интерфейс MathFunction метод: CompositeFunction andThen(MathFunction afterFunction) – с реализацией по умолчанию. Метод должен возвращать сложную функцию, у которой первой функцией является текущий объект, а второй – afterFunction. Данный метод позволяет быстро создавать сложные функции из других функций. Например, если f(x), g(x), h(x) – математические функции, f, g, h – соответствующие им ссылки на объекты (имеют тип MathFunction), то для создания сложной функции f(g(h(x))) можно будет воспользоваться конструкцией: MathFunction composite = f.andThen(g).andThen(h); Или можно сразу применить, если не нужна ссылка на объект: double result = f.andThen(g).andThen(h).apply(x); Необходимо покрыть метод тестами с различными цепочками функций.

Задание 2

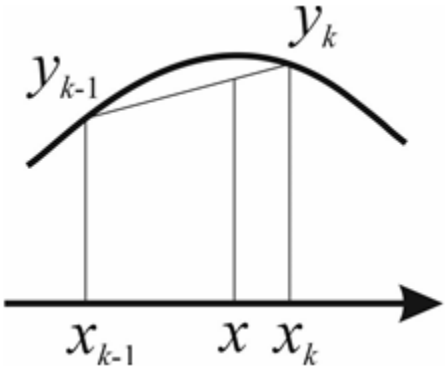
Реализованный механизм функций удобен для отложенных вычислений (значение математической функции не будет вычислено, пока не применить метод apply()), но имеет и свои недостатки.

Так, например, если метод требует много ресурсов/времени на вычисление, то его повторный вызов вновь потратит столько же ресурсов/времени. Более того, существуют функции, которые нельзя записать аналитически, либо в виде простого вычисляемого выражения. Во многих таких случаях используется дискретизация функции или табулирование её значений:

x_0	y_0
x_1	y_1
...	...
x_{n-1}	y_{n-1}

Здесь *n* – это номер аргумента и значения в таблице (индекс).

Если же требуется получить значение функции между табулированными точками, то используется интерполяция. Если значение функции выходит за интервал таблицы, то используется экстраполяция. В данной работе допускается использовать линейную интерполяцию (в том числе и для экстраполяции):



Так, значение y в точке x, находящейся внутри интервала (x_{k-1}, x_k) может быть вычислено по формуле:

$$y = y_{k-1} + \frac{y_k - y_{k-1}}{x_k - x_{k-1}}(x - x_{k-1}).$$

Для экстраполяции слева допускается использовать эту же формулу, взяв для неё самый левый интервал (между нулевым и первым индексом). Для экстраполяции справа – самый правый интервал (между предпоследним и последним индексами).

X	Создать интерфейс TabulatedFunction, расширяющий интерфейс MathFunction и добавляющий следующие методы без реализации.
	
	Метод получения количества табулированных значений:
	
	int getCount();
	
	Метод, получающий значение аргумента x по номеру индекса:
	
	double getX(int index);
	
	Метод, получающий значение y по номеру индекса:
	
	double getY(int index);
	
	Метод, задающий значение y по номеру индекса:
	
	void setY(int index, double value);
	
	Значение аргумента x задавать не рекомендуется, поскольку все x будут уникальные и отсортированы, а изменение может сломать всю сортировку. Поэтому и соответствующего метода не будет.
	
	Метод, возвращающий индекс аргумента x. Предполагается, что все x различны. Если такого x в таблице нет, то необходимо вернуть -1:
	
	int indexOfX(double x);
	
	Метод, возвращающий индекс первого вхождения значения y. Если такого y в таблице нет, то необходимо вернуть -1:
	
	int indexOfY(double y);
	
	Метод, возвращающий самый левый x:
	
	double leftBound();
	
	Метод, возвращающий самый правый x:
	
	double rightBound();
	
	Методы интерполирования и экстраполирования не нужны, потому что они в интерфейсе они автоматически станут публичными, а в нашем случае доступ извне к интерполяции ни к чему.

Y	Прежде чем переходить к реализации интерфейса TabulatedFunction, необходимо задуматься о том, каким образом таблица будет храниться. Основных способов два – либо два массива (для значений x и y), либо связный список, каждый узел которого хранит в себе один x и один y.
	
	Тем не менее, есть некоторый общий функционал, который не должен быть доступен всем. Это методы интерполяции. Сюда же можно отнести (по желанию) поле count, которое в любом случае должно быть – как у массива, так и у списка – это приватное поле отвечает за количество строк в таблице и должно быть возвращено соответствующим методом.
	
	Для того чтобы не дублировать защищённые методы и/или функционал, нужно для начала создать абстрактный класс AbstractTabulatedFunction, добавляющий следующие абстрактные protected-методы.
	
	Метод поиска индекса x, который, в отличие от обычного indexOfX(), не должен возвращать -1 (если x не найден), а должен вернуть индекс максимального значения x, которое меньше заданного x. Так, для набора значений x [-3., 4., 6.] – индексация начинается с нуля – метод, применённый к 4.5, должен вернуть 1, так как 4 – максимальный x из всего массива, который меньше 4.5, и имеет индекс 1. Если все x больше заданного, то метод должен вернуть 0; если все x меньше заданного, то метод должен вернуть count. Метод будет полезен для поиска интервала, в который попадает x (напоминание, что в таблице x предполагаются упорядоченными).
	
	int floorIndexOfX(double x);
	
	Метод экстраполяции слева:
	
	double extrapolateLeft(double x);
	
	Метод экстраполяции справа:
	
	double extrapolateRight(double x);
	
	Метод интерполяции с указанием индекса интервала:
	
	double interpolate(double x, int floorIndex);
	
	Также можно добавить защищённый метод с реализацией интерполяции, поскольку формула будет одна и та же, независимо от того, какой способ хранения данных используется:

	double interpolate(double x, double leftX, double rightX, double leftY, double rightY) {…} Наконец, можно реализовать метод apply(), определённый ещё в интерфейсе MathFunction, поскольку теперь имеется весь набор необходимых методов, несмотря на то, что их реализация ещё не готова. Метод apply() принимает на вход x. Если этот x меньше левой границы, то нужно использовать левую интерполяцию. Если он больше правой границы, то нужно использовать правую интерполяцию. Если он внутри интервала, можно попытаться найти, а есть ли он в таблице, использовав метод indexOf() – если вернулось не -1, то вернуть соответствующее y через метод getY(). В противном случае вызвать метод интерполяции с указанием индекса интервала, предварительно отыскав его с помощью метода floorIndexOfX(double x).
X* или Y*	Создать класс тестирования для реализованных методов AbstractTabulatedFunction. В отличии от предыдущих тестов, здесь есть особенность: так как класс абстрактный, он не может иметь экземпляров, и, следовательно, возникает вопрос, как протестировать написанные методы. В таких случаях создаются mock-объекты – объекты, предназначенные исключительно для тестирования и замены реальных объектов. В данном случае можно создать класс MockTabulatedFunction, расширяющий класс AbstractTabulatedFunction и реализующий все его нереализованные методы каким-нибудь очень простым способом. Например, поместив в него неизменяемые поля x0, x1, y0, y1, проинициализированные сразу же (но x0 и x1 должны быть упорядочены!). В этом случае метод getCount() должен всегда возвращать 2. После того, как mock будет дописан, можно будет доделать и тест, создавая, где необходимо mock-объекты. Класс-тест должен тестировать только методы interpolate() (тот, который был реализован) и apply().

Y	Реализовать класс AbstractTabulatedFunction на основе хранения данных в массиве. Назвать новый класс ArrayTabulatedFunction. Класс должен иметь два конструктора: 1) Конструктор с двумя параметрами типа double[]: xValues и yValues. Такие же приватные/защищённые поля должны быть и у самого класса. Предполагается, что значения xValues не повторяются и упорядочены. Также подразумевается, что длина этих массивов совпадает – приватное/защищенное поле count должно иметь значение этой длины. Есть одна особенность реализации: так как массивы являются ссылочными типами, программист за пределами этого класса может поменять их значения, что может плохо сказаться на работоспособности класса (особенно если xValues вдруг перестанут быть упорядоченными). Поэтому необходимо вместо присваивания ссылки сделать копию массива в поле. Для этого можно воспользоваться методом Arrays.copyOf(…). 2) Конструктор с четырьмя параметрами: MathFunction source, double xFrom, double xTo, int count, – данный конструктор должен принимать на вход другую функцию source, интервал xFrom – xTo (включительно) и количество точек count. Если xFrom > xTo, то надо поменять их значения местами. Необходимо заполнить поля xValues и yValues массивами размером count с помощью равномерной дискретизации исходной функции. При этом первый элемент xValues должен равняться xFrom, а последний – xTo. Важно: количество точек на 1 больше количество интервалов разбиения, что нужно учитывать при расчёте шага. Если xFrom равняется xTo, то нужно все элементы каждого массива заполнить одним и тем же значением (xFrom для xValues, source.apply(xFrom) для yValues). О том, как проводить дискретизацию, можно уточнить у преподавателя. С реализацией методов getCount(), getX(), getY(), setY(), leftBound(), rightBound() проблем возникнуть не должно. Методы indexOfX(), indexOfY() должны возвращать -1, если элемент не найден. Метод floorIndexOfX() должен быть реализован в соответствии с тем, как было описано ранее для абстрактного класса. Методы интерполирования и экстраполирования должны быть реализованы так, как было описано ранее (за исключением случае, когда размер массивов равен 1 – должно вернуться единственно заданное значение).

	После реализации все методы класса необходимо покрыть тестами.

X	Реализовать класс AbstractTabulatedFunction на основе хранения данных в двусвязном циклическом списке. Назвать новый класс LinkedListTabulatedFunction. Прежде чем переходить к непосредственной реализации, нужно создать вспомогательный класс узла списка – Node. За пределами пакета никто не должен знать о существовании этого класса. Класс должен иметь 4 публичных поля: 2 поля типа Node (next и prev) и 2 – типа double: x и y. Поля next и prev ссылаются соответственно на следующий и предыдущий узел в списке. Перейти к реализации LinkedListTabulatedFunction. Если в абстрактном классе не было защищённого поля count, то добавить его. Также должно быть приватное поле Node head, которое ссылается на голову списка. Если длина списка равна 0, то head == null. Перед написанием конструкторов необходимо добавить приватный метод addNode(double x, double y), добавляющий новый узел в конец списка. В новом узле должны быть записаны входные значения x и y. Если при этом список пустой (голова не задана), то новый узел должен стать головой списка. Его поля next и prev должны ссылаться на самого себя. Если же список не пустой, то нужно добавить новый узел к уже имеющимся. Для этого надо получить ссылку last на последний узел списка, коим является head.prev (так как список циклический). Поля last.next и head.prev теперь должны ссылаться на новый узел. У нового узла поля prev и next должны соответственно ссылаться на узлы last и head. Вне зависимости от того, был список пустой или нет, count должен увеличиться на 1. Класс должен иметь два конструктора (такие же, как у класса ArrayTabulatedFunction). Реализация будет отличаться. 1) Конструктор с двумя параметрами типа double[]: xValues и yValues. Предполагается, что значения xValues не повторяются и упорядочены. Также подразумевается, что длина этих массивов совпадает. Эту длину записывать в поле count не нужно! Вместо этого нужно обойти в цикле сразу оба массива, на каждой итерации вызывая метод addNode(x, y). 2) Конструктор с четырьмя параметрами: MathFunction source, double xFrom, double xTo, int count, – данный конструктор должен принимать на вход другую функцию source, интервал xFrom – xTo (включительно) и количество точек count. Если xFrom > xTo, то надо поменять их значения местами.

	Необходимо заполнить список с помощью равномерной дискретизации исходной функции. При этом у головы списка x должен равняться xFrom, а у последнего узла – xTo. Важно: количество точек на 1 больше количество интервалов разбиения, что нужно учитывать при расчёте шага. Если xFrom равняется xTo, то нужно, чтобы все элементы списка были заполнены одним и тем же значением (xFrom для всех node.x, source.apply(xFrom) для всех node.y). О том, как проводить дискретизацию, можно уточнить у преподавателя. С реализацией методов getCount(),leftBound(), rightBound() проблем возникнуть не должно. Для остальных методов нужно добавить вспомогательный приватный метод Node getNode(int index), возвращающий ссылку на узел номер index. Предполагается, что индекс всегда корректный. Необходимо в цикле бежать по элементам списка, пока не будет найден нужный по счёту. Его и нужно будет вернуть из метода. Дополнительно можно реализовать, чтобы в случае, когда индекс больше половины count, бежать с хвоста списка. После этого с методами getX(), getY(), setY() проблем возникнуть не должно. Методы indexOfX(), indexOfY() должны возвращать -1, если элемент не найден. Метод floorIndexOfX() должен быть реализован в соответствии с тем, как было описано ранее для абстрактного класса. Методы интерполирования и экстраполирования должны быть реализованы так, как было описано ранее (за исключением случае, когда размер массивов равен 1 – должно вернуться единственно заданное значение). После реализации все методы класса необходимо покрыть тестами. Если есть желание покрыть тестами и приватные методы, можно сделать их защищёнными.
X*	Реализация метода apply() для LinkedListTabulatedFunction, написанная в классе AbstractTabulatedFunction, не очень удачна, поскольку сначала ищет индекс узла методом floorIndexOfX(), а затем во время интерполяции происходит поиск узла с таким индексом. Иными словами, дважды выполняется проход по части списка, что не очень хорошо. Можно переопределить метод apply(), который будет использовать вместо метода floorIndexOfX() новый приватный/защищённый метод floorNodeOfX(), возвращающий не

	индекс, а узел. После переопределения необходимо будет покрыть методы тестами.
X и Y	Написать дополнительные тесты для сложных функций с учётом комбинаций табулированных функций друг с другом (реализованных как с помощью массива, так и с помощью списка), а также табулированных и других функций.

Задание 3

X или Y	Создать интерфейс Insertable, содержащий единственный метод void insert(x, y), вставляющий значение в табулированную функцию. При этом предполагается, что если x уже содержится в таблице, то его значение y заменяется на новое. Если входное x находится между двумя другими значениями x, то в таблицу добавляется значение между ними. Если входное x левее всех имеющихся x, то значение добавляется в начало таблицы. Если правее – то в конец.
X	У класса ArrayTabulatedFunction реализовать интерфейс Insertable. Если x уже есть в массиве xValues, то просто заменить соответствующие значение y в массиве yValues и закончить выполнение метода. Если же такого x нет, то нужно увеличить размер массива и добавить x и y в соответствующее место. Так как изменять размеры массивов нельзя, то придётся создать новый и скопировать туда значения текущих массивов (не забыв добавить новое), а затем присвоить ссылки полей на соответствующие новые массивы. Для копирования значений массивов рекомендуется пользоваться методом System.arraycopy(). Не забыть увеличить поле count на 1. Дополнительно можно реализовать, чтобы массивы xValues[] и yValues[] были с запасом, т.е. их реальный размер был больше, чем значение count. Тогда создавать массив можно будет не каждый раз, поскольку такая операция обходится весьма дорого. Покрыть новый метод тестами.

Y	У класса LinkedListTabulatedFunction реализовать интерфейс Insertable. Если список пустой, то просто делегировать выполнение методу addNode() и завершить выполнение метода insert(). Затем аккуратно пробежаться по списку и найти место, в которое нужно добавить новый узел (если вдруг такое x уже есть, то просто заменить у соответствующего узла y и завершить выполнение метода). Добавление в середину и в конец списка не должны вызывать проблем. Если же узел добавляется в начало списка (когда во всех узлах x больше заданного), то необходимо после этого обновить ещё и головную ссылку списка. Не забыть обновить поле count. Покрыть новый метод тестами.
X или Y	Создать интерфейс Removable с единственным методом void remove(int index). Догадаться, для чего нужен этот метод и как его следует реализовывать.
Y	У класса ArrayTabulatedFunction реализовать интерфейс Removable. Покрыть новый метод тестами.
X	У класса LinkedListTabulatedFunction реализовать интерфейс Removable. Покрыть новый метод тестами.