Лабораторная работа №4

Задание на лабораторную работу

Лабораторная работа является продолжением лабораторной работы №3. В работе изучаются основы многопоточных приложений, в том числе возможности запуска нескольких потоков, которые работают с общими ресурсами (функциями). Также в ходе работы студенты должны ознакомиться с блокировками и научиться их использовать.

Задание 1

X или Y	Рядом с пакетами `exceptions`, `functions`, `io` и `operations` создать пакет `concurrent`.
X	В пакете `concurrent` создать класс `ReadTask`, реализующий интерфейс `Runnable`. Класс должен иметь приватное поле, содержащее ссылку на табулированную функцию, которая передаётся в конструкторе. Реализовать метод `run`() следующим образом: В цикле `for` (не for-each) пробежаться по всем записям табулированной функции и для каждой из них в консоли на отдельной строке вывести следующую информацию: `After read: i = %d, x = %f, y = %f` В параметры должны быть подставлены соответственно: индекс и значения `(x, y)` из табулированной функции.
X	В пакете `concurrent` создать класс `WriteTask`, реализующий интерфейс `Runnable`. Класс должен иметь два приватных поля: первое – это ссылка на табулированную функцию; второе – значение с плавающей точкой `value`, которое будет записываться в эту функцию. Оба параметра должны передаваться в конструкторе. Реализовать метод `run`() следующим образом: В цикле `for` (не for-each) пробежаться по всем записям табулированной функции и задать в качестве значений у значение `value` (у всех одно и то же). После этого внутри цикла вывести в консоль сообщение: `Writing for index %d complete` Вместо параметра `%d` должен подставляться индекс.
X	В пакете `concurrent` создать класс `ReadWriteTaskExecutor` с main-методом. Внутри метода должна создаваться табулированная функция (с реализацией в виде связного списка) с помощью конструктора, который создаёт функцию на основе другой функции. В конструктор требуется передать тождественно равную отрицательному числу (например, `-1`) константную функцию `ConstantFunction` и интервал от `1` до какого-нибудь большого числа (например, `1000`). Столько же точек должна иметь функция. Далее требуется создать два потока исполнения `Thread`, один из которых принимает объект `ReadTask`, другой – `WriteTask`. При создании обоих объектов следует передать в их конструкторы ранее созданную табулированную функцию. Для `WriteTask` вторым параметром в конструктор должно передаваться какое-либо положительное число, например, `0,5`. После создания потоков требуется их стартовать. Запустить метод на исполнение. Изучить выведенную в консоль информацию. Если для заданного индекса чтение данных из функции выполнилось раньше, чем запись значения в функцию, то в консоль (в идеальном случае) для него

	должно выводиться отрицательное `y`. Иначе – положительное. Убедитесь, что это не так. Возможно, придётся запустить приложение несколько раз.
	Пример:
	`After read: i = 1, x = 2,000000, y = 0,500000`
	`Writing for index 1 complete`
	Для индекса `i = 1` вывелось, что значение `y` положительно, хотя информация о записи появилась только следующей строчкой. И наоборот:
	`Writing for index 3 complete`
	`After read: i = 3, x = 4,000000, y = -1,000000`
	Записи сообщают, что для индекса `i = 3` значение `y` уже стало положительным, однако следующая строка пишет, что оно якобы ещё отрицательно.
	Это связано с тем, что вывод в консоль не всегда успевает осуществляться вместе с чтением/записью.

X	С помощью блоков синхронизации исправить проблему вывода в консоль у задач `ReadTask` и `WriteTask`. Блоки должны быть внутри циклов `for`, иначе поток исполнения одной из задач чтения/записи не передаст управление потоку другой задачи до тех пор, пока полностью не выполнится.
Y	В пакете `concurrent` создать класс `MultiplyingTask`, реализующий интерфейс `Runnable`. Класс должен иметь приватное поле, содержащее ссылку на табулированную функцию, которая передаётся в конструкторе. Реализовать метод `run()` следующим образом: В цикле `for` (не for-each) пробежаться по всем записям табулированной функции и для каждой из них увеличить значение `y` в 2 раза (перезаписать его). После цикла вывести в консоль информацию о том, что текущий поток (указать имя потока) закончил выполнение задачи.
Y	В пакете `concurrent` создать класс `MultiplyingTaskExecutor` с main-методом. Внутри метода должна создаваться табулированная функция (с реализацией в виде связного списка) с помощью конструктора, который создаёт функцию на основе другой функции. В конструктор требуется передать тождественно равную единице функцию `UnitFunction` и интервал от `1` до какого-нибудь большого числа (например, `1000`). Далее требуется создать список `List` потоков. В цикле `for` выполнить небольшое число (например, `10`) итераций, в ходе каждой из которых создаётся задача `MultiplyingTask` с передачей в конструктор ранее созданной функции. Для каждой задачи создаётся поток, её исполняющий, который добавляется в список потоков. После цикла требуется пройтись по всему списку и стартовать все потоки. Затем необходимо усыпить текущий поток (например, на пару секунд), дав шанс выполниться остальным. После пробуждения текущий поток должен вывести в консоль табулированную функцию. Запустить написанный метод и внимательно изучить результат. Если потоков исполнения было `10`, то в ходе выполнения каждый из них должен был умножить значения `y` на `2`, в результате чего начальное `y` (которое было равно `1`) должно было увеличиться в `1024` раза. Убедитесь, что в общем случае это не так. Возможно, придётся запустить приложение несколько раз. Проблема связана с тем, что иногда один поток исполнения не успевает записать свой результат в функцию, в то время как другой поток успевает извлечь из неё неактуальное значение.

Y	С помощью блоков синхронизации исправить проблему некорректных значений результата у `MultiplyingTaskExecutor`. Блок должен быть внутри цикла `for`, иначе один поток исполнения будет блокировать все остальные потоки до тех пор, пока полностью не выполнит свою задачу.
Y*	Ожидание главным потоком остальных фиксированное время – не лучшая идея. Она является причинами чрезмерного простоя потока и, напротив, преждевременного возобновления исполнения. Требуется заменить время ожидания на проверку, а не закончили ли другие потоки исполняться. Для этого задачам `MultiplyingTask` понадобится добавить информацию о том, что они были выполнены / ещё не были выполнены. Все задачи нужно будет хранить в отдельном наборе `Collection` (выбрать самостоятельно, какую именно коллекцию взять). Главный поток должен постоянно проверять все задачи и смотреть, не выполнилась ли очередная. Если да – то удалять её из набора. Как только в наборе не осталось невыполненных задач, продолжить выполнение далее. Описанное решение тоже не является идеальным, так как заставляет один из потоков постоянно следить за другими и тратить на это ресурсы. Важно: если использовать итератор при обходе коллекции, то удаление элемента из этой коллекции может приводить к непредсказуемым результатам работы итератора.

Задание 2

X или Y	В пакете `concurrent` создать класс `SynchronizedTabulatedFunction`, реализующий интерфейс `TabulatedFunction` и представляющий собой потокобезопасную обёртку над объектом табулированной функции в соответствии с паттерном проектирования «Декоратор». Внутри созданного класса должна храниться ссылка на объект `TabulatedFunction`, которая приходит в качестве параметра конструктора. Все методы класса должны делегировать своё поведение агрегируемому объекту, но должны быть синхронизованы. Т.е. при работе в многопоточной среде до тех пор, пока один из потоков выполняет какой-либо метод обёртки, другие потоки не могут выполнять любой другой (или этот же) метод. Реализовать такое поведение с помощью блокировок: можно предложить своё решение, а можно подсмотреть идею реализации, например, у метода `Collections.synchronizedCollection()`. После реализации добавить тесты для проверки работоспособности созданного класса в однопоточной среде. Тесты для многопоточной среды создавать не требуется.
X	Переписать метод `iterator()` у `SynchronizedTabulatedFunction`. Метод итератора, хотя и запускается в синхронизированном блоке, тем не менее возвращает итератор, который не является потокобезопасным и не содержит никаких блокировок. Один из способов достижения потокобезопасности заключается в том, что в блоке синхронизации все данные из функции копируются, а итератор пробегается по сделанной копии, которая больше никому не доступна. Тем самым гарантируется, что исключение `ConcurrentModificationException` не будет брошено в случае изменения данных (ведь итератор проходит по копии старых данных). Такое поведение реализовано, например, в классе `CopyOnWriteArrayList`. Для реализации поведения можно в блоке синхронизации воспользоваться методом `TabulatedFunctionOperationService.asPoints()` для получения

	копии данных внутренней функции в виде массива точек `Point` и после этого создать объект анонимного итератора, который должен будет пробегаться по этому массиву. Покрыть итератор тестами в однопоточной среде.
Y	Объекты класса `SynchronizedTabulatedFunction` обладают тем недостатком, что вызов каждого метода производится в своём блоке синхронизации. Т.е., например, между вызовами двух методов одним потоком исполнения может успеть «влезть» другой поток. Требуется написать метод, который бы позволял совершать набор действий с объектом этого класса внутри одного блока синхронизации. Для этого нужно создать внутренний публичный интерфейс операции `Operation` с типом-параметром `T`, характеризующий возвращаемое значение операции (если возвращаемое значение не нужно, можно будет подставлять вместо `T` тип `Void`, а в конце реализации возвращать `null`). Интерфейс должен содержать единственный метод `apply()`, принимающий в качестве аргумента `SynchronizedTabulatedFunction` и возвращающий `T`. Добавить в класс `SynchronizedTabulatedFunction` метод `doSynchronously()` с типом-параметром `T`, принимающий на вход операцию `Operation` и возвращающий `T`. Подумать, чем именно приходится операция для параметра `T` (тип возвращаемого значения) – производителем или потребителем – и на основании этого записать в `Operation<...>` правильный подстановочный символ (wildcard). Метод должен выполнять указанную операцию, возвращать значение, и всё это должно быть внутри блока синхронизации. Покрыть метод тестами в однопоточной среде для различных типов `Operation<T>` (`T` в том числе `Void`). Для этого можно использовать анонимные классы или лямбда-выражения.
Y	Добавить в класс `TabulatedDifferentialOperator` метод `deriveSynchronously()` по аналогии с методом `derive()`. Метод должен вычислять производную и возвращать результат в условиях работы в многопоточной среде. В методе в первую очередь следует обернуть входную функцию в синхронизированную обёртку `SynchronizedTabulatedFunction`. Однако это не нужно делать, если входная функция уже удовлетворяет этому типу (т.е. сама является синхронизированной обёрткой). У полученного объекта следует вызвать метод `doSynchronously()` и передать туда операцию, вычисляющую производную, т.е. метод реализации должен делегировать своё поведению методу `derive()` текущего класса. Для создания операции можно использовать анонимный класс или лямбда-выражение, в том числе в виде ссылки на метод. Покрыть работу метода тестами в однопоточной среде.

Задание 3*

X* и / или Y*	Добавить в проект новый функционал, вычисляющий интеграл от табулированной функции на всей области задания. Численный метод интегрирования выбрать самостоятельно. Интеграл должен быть рассчитан параллельно: каждый поток считает свою порцию данных (участок интегрирования), а затем результаты расчётов суммируются для получения правильного ответа. При реализации следует использовать пакет `java.util.concurrent`, в том числе классы исполнителей потоков и задач, а также очереди. Тщательно продумать, какие классы и интерфейсы требуется создать. Можно реализовать стратегию «разделяй и властвуй».