Домашние задания

Домашнее задание 1. Обход файлов

1. Разработайте класс `Walk`, осуществляющий подсчет хэш-сумм файлов.
    1. Формат запуска:
       ```
       java Walk <входной файл> <выходной файл>
       ```
    2. Входной файл содержит список файлов, которые требуется обойти.
    3. Выходной файл должен содержать по одной строке для каждого файла. Формат строки:
       ```
       <шестнадцатеричная хэш-сумма> <путь к файлу>
       ```
    4. Для подсчета хэш-суммы используйте последние 64 бита SHA-256 (поддержка есть в стандартной библиотеке).
    5. Если при чтении файла возникают ошибки, укажите в качестве его хэш-суммы все нули.
    6. Кодировка входного и выходного файлов — UTF-8.
    7. Размеры файлов могут превышать размер оперативной памяти.
    8. Пример

Входной файл
samples/1
samples/12
samples/123
samples/1234
samples/1
samples/binary
samples/no-such-file

Выходной файл
6b86b273ff34fce1 samples/1
6b51d431df5d7f14 samples/12
a665a45920422f9d samples/123
03ac674216f3e15c samples/1234
6b86b273ff34fce1 samples/1
40aff2e9d2d8922e samples/binary
0000000000000000 samples/no-such-file

2. Сложный вариант:
    1. Разработайте класс `RecursiveWalk`, осуществляющий подсчет хэш-сумм файлов в директориях.
    2. Входной файл содержит список файлов и директорий, которые требуется обойти. Обход директорий осуществляется рекурсивно.
    3. Пример:

Входной файл
samples/binary
samples
samples/no-such-file

Выходной файл
40aff2e9d2d8922e samples/binary
6b86b273ff34fce1 samples/1
6b51d431df5d7f14 samples/12
a665a45920422f9d samples/123
03ac674216f3e15c samples/1234
40aff2e9d2d8922e samples/binary
0000000000000000 samples/no-such-file

3. При выполнении задания следует обратить внимание на:
    - Дизайн и обработку исключений, диагностику ошибок.
    - Программа должна корректно завершаться даже в случае ошибки.
    - Корректная работа с вводом-выводом.
    - Отсутствие утечки ресурсов.
    - Возможность повторного использования кода.
4. Требования к оформлению задания.
    - Проверяется исходный код задания.
    - Весь код должен находиться в пакете `info.kgeorgiy.ja.фамилия.walk`.

Репозиторий курса

Домашнее задание 2. Множество на массиве

1. Разработайте класс `ArraySet`, реализующий неизменяемое упорядоченное множество.
    - Класс `ArraySet` должен реализовывать интерфейс SortedSet (простой вариант) или NavigableSet (сложный вариант).
    - Все операции над множествами должны производиться с наилучшей асимптотической эффективностью.
2. При выполнении задания следует обратить внимание на:
    - Применение стандартных коллекций.
    - Избавление от повторяющегося кода.
    - Отсутствие unchecked warnings при компиляции.
    - Отсутствие излишних подавленных unchecked warnings.

Домашнее задание 3. Студенты

1. Разработайте класс `StudentDB`, осуществляющий поиск по базе данных студентов.
    - Класс `StudentDB` должен реализовывать интерфейс `StudentQuery` (простой вариант) или `GroupQuery` (сложный вариант).
    - Каждый метод должен состоять из ровно одного оператора. При этом длинные операторы надо разбивать на несколько строк.
2. При выполнении задания следует обратить внимание на:
    - применение лямбда-выражений и потоков;
    - избавление от повторяющегося кода.

Домашнее задание 4. Сплитераторы и коллекторы

1. Разработайте класс `Lambda`, реализующий сплитераторы для деревьев и дополнительные коллекторы.
    - *Простой вариант* (интерфейс `EasyLambda`) — реализуйте:
        - сплитераторы для двоичных деревьев, двоичных деревьев с известным размером, k-ичных деревьев;
        - коллекторы первого, последнего, среднего элементов;
        - коллекторы общего префикса и суффикса строк.
    - *Сложный вариант* (интерфейс `HardLambda`) — дополнительно реализуйте:
        - сплитераторы всех видов деревьев над списками элементов;
        - коллектор *n*-ого элемента;
        - коллекторы первых и последних *n* элементов.
2. При выполнении задания следует обратить внимание на:
    - характеристики создаваемых сплитераторов;
    - избавление от повторяющегося кода.

Домашнее задание 5. Implementor

1. Реализуйте класс `Implementor`, генерирующий реализации классов и интерфейсов.
    - Аргумент командной строки: полное имя класса/интерфейса, для которого требуется сгенерировать реализацию.
    - В результате работы должен быть сгенерирован java-код класса с суффиксом `Impl`, расширяющий (реализующий) указанный класс (интерфейс).
    - Сгенерированный класс должен компилироваться без ошибок.
    - Сгенерированный класс не должен быть абстрактным.
    - Методы сгенерированного класса должны игнорировать свои аргументы и возвращать значения по умолчанию.
2. В задании выделяются три варианта:
    - *Простой* — `Implementor` должен уметь реализовывать только интерфейсы (но не классы). Поддержка generics не требуется.
    - *Сложный* — `Implementor` должен уметь реализовывать и классы, и интерфейсы. Поддержка generics не требуется.
    - *Бонусный* — `Implementor` должен уметь реализовывать generic-классы и интерфейсы. Сгенерированный код должен иметь корректные параметры типов и не порождать unchecked warnings.

Домашнее задание 6. Jar Implementor

Это домашнее задание **связано** с предыдущим и будет приниматься только с ним. Предыдущее домашнее задание отдельно сдать будет нельзя.

1. Создайте `.jar`-файл, содержащий скомпилированный `Implementor` и сопутствующие классы.
    - Созданный `.jar`-файл должен запускаться командой `java -jar`.
    - Запускаемый `.jar`-файл должен принимать те же аргументы командной строки, что и класс `Implementor`.
2. Модифицируйте `Implementor` так, чтобы при запуске с аргументами `-jar имя-класса файл.jar` он генерировал `.jar`-файл с реализацией соответствующего класса (интерфейса). Для компиляции можно использовать код из тестов.
3. Вы можете создавать файлы и директории в текущем каталоге, но не за его пределами.
4. Для проверки, кроме исходного кода, также должны быть представлены:
    - скрипт для создания запускаемого `.jar`-файла, в том числе, исходный код манифеста;
    - запускаемый `.jar`-файл.
5. **Сложный вариант**. Решение должно быть модуляризовано.

Домашнее задание 7. Javadoc

Это домашнее задание **связано** с двумя предыдущими и будет приниматься только с ними. Предыдущие домашнее задание отдельно сдать будет нельзя.

1. Документируйте класс `Implementor` и сопутствующие классы с применением Javadoc.
    - Должны быть документированы все классы и все члены классов, в том числе `private`.
    - Документация должна генерироваться без предупреждений.
    - Сгенерированная документация должна содержать корректные ссылки на классы стандартной библиотеки и модулей `info.kgeorgiy.java.advanced.*`.
2. Для проверки, кроме исходного кода, также должны быть представлены:
    - скрипт для генерации документации (он может рассчитывать, что рядом с вашим репозиторием склонирован репозиторий курса);
    - сгенерированная документация.

В **последующих** домашних заданиях все `public` и `protected` сущности должны быть документированы.

Домашнее задание 8. Итеративный параллелизм

1. Реализуйте класс `IterativeParallelism`, который будет обрабатывать списки в несколько потоков.
2. В простом варианте должны быть реализованы следующие методы:
    - `argMax(threads, list, comparator)` — индекс первого максимума;
    - `argMin(threads, list, comparator)` — индекс первого минимума;
    - `indexOf(threads, list, predicate)` — индекс первого элемента, удовлетворяющего предикат;
    - `lastIndexOf(threads, list, predicate)` — индекс последнего элемента, удовлетворяющего предикат;
    - `sumIndices(threads, list, predicate)` — сумма индексов элементов, удовлетворяющих предикат;
3. В сложном варианте должны быть дополнительно реализованы следующие методы:
    - `indices(threads, list, predicate)` — индексы элементов, удовлетворяющих предикат;
    - `filter(threads, list, predicate)` — вернуть список, содержащий элементы удовлетворяющие предикату;
    - `map(threads, list, function)` — вернуть список, содержащий результаты применения функции;
4. Во все функции передается параметр `threads` — сколько потоков надо использовать при вычислении. Вы можете рассчитывать, что число потоков относительно мало.
5. Не следует рассчитывать на то, что переданные компараторы, предикаты и функции работают быстро.
6. Можно сделать O(`threads`), но не O(`list.size()`) действий без распараллеливания.
7. При выполнении задания **нельзя** использовать *Concurrency Utilities* и *Parallel Streams*.

Домашнее задание 9. Параллельный запуск

1. Напишите класс `ParallelMapperImpl`, реализующий интерфейс `ParallelMapper`.
   ```
   public interface ParallelMapper extends AutoCloseable {
       <T, R> List<R> map(
           Function<? super T, ? extends R> f,
           List<? extends T> args
       ) throws InterruptedException;

       @Override
       void close();
   }
   ```
    - Метод `map` должен параллельно вычислять функцию `f` на каждом из указанных аргументов (`args`).
    - Конструктор `ParallelMapperImpl(int threads)` должен создавать `threads` рабочих потоков, которые используются для распараллеливания.
    - Метод `close` должен останавливать все рабочие потоки.
    - К одному `ParallelMapperImpl` могут одновременно обращаться несколько клиентов.
    - При недостатке потоков для распараллеливания, задания на исполнение должны накапливаться в очереди и обрабатываться в порядке поступления.
    - В реализации не должно быть активных ожиданий.
    - Код должен находиться в пакете `iterative`.
    - Обратите внимание на обработку исключений, кидаемых функцией `f`.
        1. Исключения не должны приводить к сокращению числа рабочих потоков.
        2. **Сложный вариант.** Исключения должны выкидываться из метода `map`.
2. Доработайте класс `IterativeParallelism` так, чтобы он мог использовать `ParallelMapper`.
    - Добавьте конструктор `IterativeParallelism(ParallelMapper)`.
    - Методы класса должны делить работу на `threads` фрагментов и исполнять их при помощи `ParallelMapper`.
    - При наличии `ParallelMapper` сам `IterativeParallelism` новые потоки создавать не должен.
    - Должна быть возможность одновременного запуска и работы нескольких клиентов, использующих один `ParallelMapper`.
3. При выполнении задания всё ещё **нельзя** использовать *Concurrency Utilities* и *Parallel Streams*.

Домашнее задание 10. Web Crawler

1. Напишите потокобезопасный класс `WebCrawler`, который будет рекурсивно обходить сайты.
    1. Класс `WebCrawler` должен иметь конструктор
       ```
       public WebCrawler(Downloader downloader, int downloaders, int extractors, int perHost)
       ```
        - `downloader` позволяет скачивать страницы и извлекать из них ссылки;
        - `downloaders` — максимальное число одновременно загружаемых страниц;
        - `extractors` — максимальное число страниц, из которых одновременно извлекаются ссылки;
        - `perHost` — максимальное число страниц, одновременно загружаемых с одного хоста. Для определения хоста следует использовать метод `getHost` класса `URLUtils` из тестов.
    2. Класс `WebCrawler` должен реализовывать интерфейс `Crawler`
       ```
       public interface Crawler extends AutoCloseable {
           Result download(String url, int depth);

           void close();
       }
       ```
        - Метод `download` должен рекурсивно обходить страницы, начиная с указанного URL, на указанную глубину и возвращать список загруженных страниц и файлов. Например, если глубина равна 1, то должна быть загружена только указанная страница. Если глубина равна 2, то указанная страница и те страницы и файлы, на которые она ссылается, и так далее.
        - Метод `download` может вызываться параллельно в нескольких потоках.
        - Загрузка и обработка страниц (извлечение ссылок) должна выполняться максимально параллельно, с учетом ограничений на число одновременно загружаемых страниц (в том числе с одного хоста) и страниц, с которых загружаются ссылки.
        - Для распараллеливания разрешается создать `downloaders + extractors` вспомогательных потоков.
        - Повторно загружать и/или извлекать ссылки из одной и той же страницы в рамках одного обхода (`download`) запрещается.
        - Метод `close` должен завершать все вспомогательные потоки.
    3. Для загрузки страниц должен применяться `Downloader`, передаваемый первым аргументом конструктора.
       ```
       public interface Downloader {
           public Document download(final String url) throws IOException;
       }
       ```
        - Метод `download` загружает документ по его адресу (URL).
        - Документ позволяет получить ссылки по загруженной странице:
          ```
          public interface Document {
              List<String> extractLinks() throws IOException;
          }
          ```
          Ссылки, возвращаемые документом, являются абсолютными и имеют схему `http` или `https`.
    4. Должен быть реализован метод `main`, позволяющий запустить обход из командной строки
        - Командная строка
          ```
          WebCrawler url [depth [downloaders [extractors [perHost]]]]
          ```
        - Для загрузки страниц требуется использовать реализацию `CachingDownloader` из тестов.
2. Версии задания
    1. *Простая* — не требуется учитывать ограничения на число одновременных закачек с одного хоста (`perHost >= downloaders`).
    2. *Полная* — требуется учитывать все ограничения.
    3. *Бонусная* — сделать параллельный обход в ширину.
3. Задание подразумевает активное использование Concurrency Utilities, в частности, в решении не должно быть «велосипедов», аналогичных/легко сводящихся к классам из Concurrency Utilities.

Домашнее задание 11. HelloUDP

1. Реализуйте клиент и сервер, взаимодействующие по UDP.
2. Класс `HelloUDPClient` должен отправлять запросы на сервер, принимать результаты и выводить их на консоль.
    - Аргументы командной строки:
        1. имя или ip-адрес компьютера, на котором запущен сервер;
        2. номер порта, на который отсылать запросы;
        3. префикс запросов (строка);
        4. число параллельных потоков запросов;
        5. число запросов в каждом потоке.
    - Запросы должны одновременно отсылаться в указанном числе потоков. Каждый поток должен ожидать обработки своего запроса и выводить сам запрос и результат его обработки на консоль. Если запрос не был обработан, требуется послать его заново.
    - Запросы должны формироваться по схеме <префикс запросов><номер запроса в потоке>_<номер потока>.
3. Класс `HelloUDPServer` должен принимать задания, отсылаемые классом `HelloUDPClient` и отвечать на них.
    - Аргументы командной строки:
        1. номер порта, по которому будут приниматься запросы;
        2. число рабочих потоков, которые будут обрабатывать запросы.
    - Ответом на запрос должно быть Hello, <текст запроса>.
    - Несмотря на то, что текущий способ получения ответа по запросу очень прост, сервер должен быть рассчитан на ситуацию, когда этот процесс может требовать много ресурсов и времени.
    - Если сервер не успевает обрабатывать запросы, прием запросов может быть временно приостановлен.

Домашнее задание 12. Физические лица

1. Добавьте к банковскому приложению возможность работы с физическими лицами.
    1. У физического лица (`Person`) можно запросить имя, фамилию и номер паспорта.
    2. Удалённые физические лица (`RemotePerson`) должны передаваться при помощи удалённых объектов.
    3. Локальные физические лица (`LocalPerson`) должны передаваться при помощи механизма сериализации, и при последующем использовании не требовать связи с сервером.
    4. Должна быть возможность поиска физического лица по номеру паспорта, с выбором типа возвращаемого лица.
    5. Должна быть возможность создания записи о физическом лице по его данным.
2. У физического лица может быть несколько счетов, к которым должен предоставляться доступ через `Person`.
    1. Счёту физического лица с идентификатором *subId* должен соответствовать банковский счёт с *id* вида *personId*:*subId*.
    2. Изменения, производимые со счётом в банке (создание и изменение баланса), должны быть видны всем соответствующим `RemotePerson`, и только тем `LocalPerson`, которые были созданы после этого изменения.
    3. Изменения в счетах, производимые через `RemotePerson`, должны сразу применяться глобально, а производимые через `LocalPerson` – только локально для этого конкретного `LocalPerson`.
3. Реализуйте приложение, демонстрирующее работу с физическим лицами.
    1. Аргументы командной строки: имя, фамилия, номер паспорта физического лица, номер счёта, изменение суммы счёта.
    2. Если информация об указанном физическом лице отсутствует, то оно должно быть добавлено. В противном случае – должны быть проверены его данные.
    3. Если у физического лица отсутствует счёт с указанным номером, то он создается с нулевым балансом.
    4. После обновления суммы счёта новый баланс должен выводиться на консоль.
4. **Сложный вариант**
    1. На каждом счету всегда должно быть неотрицательное количество денег.
5. Приложение должно находиться в пакете `info.kgeorgiy.ja.*.bank` и его подпакетах.

Домашнее задание 13. Физические лица (тесты)

1. Напишите тесты, проверяющие поведение банка и приложения из домашнего задания Физические лица.
    - Для реализации тестов рекомендуется использовать JUnit (Tutorial). Множество примеров использования можно найти в тестах.
    - Если вы знакомы с другим тестовым фреймворком (например, TestNG), то можете использовать его.
    - Добавьте jar-файлы используемых библиотек в каталог `lib` вашего репозитория.
    - *Нельзя* использовать самописные фреймворки и тесты, запускаемые через `main`.
2. **Сложный вариант**
    1. Тесты не должны рассчитывать на наличие запущенного RMI Registry.
    2. Создайте класс `BankTests`, запускающий тесты.
    3. Создайте скрипт, запускающий `BankTests` и возвращающий код (статус) `0` в случае успеха и `1` в случае неудачи.
    4. Создайте скрипт, запускающий тесты с использованием стандартного подхода для вашего тестового фреймворка. Код возврата должен быть как в предыдущем пункте.
3. Тесты должны находиться в пакете `info.kgeorgiy.ja.*.bank` и его подпакетах.
4. При сдаче вам нужно будет продемонстрировать работу тестов на вашем компьютере.

Это домашнее задание **связано** с домашним заданием Физические лица и будет приниматься только с ним.

Домашнее задание 14. HelloNonblockingUDP

1. Реализуйте клиент и сервер, взаимодействующие по UDP, используя только неблокирующий ввод-вывод.
2. Класс `HelloUDPNonblockingClient` должен иметь функциональность аналогичную `HelloUDPClient`, но без создания новых потоков.
3. Класс `HelloUDPNonblockingServer` должен иметь функциональность аналогичную `HelloUDPServer`, но все операции с сокетом должны производиться в одном потоке.
4. В реализации не должно быть активных ожиданий, в том числе через `Selector`.
5. Обратите внимание на выделение общего кода старой и новой реализации.
6. Можно использовать неблокирующий или асинхронный ввод-вывод. Нельзя использовать виртуальные потоки.
7. *Бонусный вариант*. Клиент и сервер могут перед началом работы выделить *O(число рабочих потоков)* памяти. Выделять дополнительную память во время работы запрещено.

Домашнее задание 15. Статистика текста

1. Создайте приложение `TextStatistics`, анализирующее тексты на различных языках.
    1. Аргументы командной строки:
        - локаль текста,
        - локаль вывода,
        - файл с текстом,
        - файл отчета.
    2. Поддерживаемые локали текста: все локали, имеющиеся в системе.
    3. Поддерживаемые локали вывода: русская и английская.
    4. Файлы имеют кодировку UTF-8.
    5. Подсчет статистики должен вестись по следующим категориям:
        - предложения,
        - слова,
        - числа,
        - деньги,
        - даты.
    6. Для каждой категории должна собираться следующая статистика:
        - число вхождений,
        - число различных значений,
        - минимальное значение,
        - максимальное значение,
        - минимальная длина,
        - максимальная длина,
        - среднее значение/длина.
    7. Пример отчета:
       ```
       Анализируемый файл "quine.ru-RU.in".
       Сводная статистика
           Число предложений: 30.
           Число слов: 117.
           Число чисел: 35.
           Число сумм: 3.
           Число дат: 3.
       Статистика по предложениям
           Число предложений: 30 (30 различных).
           Минимальное предложение: "Анализируемый файл "quine.ru-RU.in".".
           Максимальное предложение: "Число чисел: 35.".
           Минимальная длина предложения: 13 ("Число дат: 3.").
           Максимальная длина предложения: 109 ("GK: если сюда поставить реальное предложение, то процесс не сойдётся").
           Средняя длина предложения: 37,7.
       Статистика по словам
           Число слов: 117 (48 различных).
           Минимальное слово: "GK".
           Максимальное слово: "языках".
           Минимальная длина слова: 1 ("с").
           Максимальная длина слова: 18 ("TextStatisticsTest").
           Средняя длина слова: 6,752.
       Статистика по числам
           Число чисел: 35 (21 различных).
           Минимальное число: -12345,67.
           Максимальное число: 12345,67.
           Среднее число: 221,381.
       Статистика по суммам денег
           Число сумм: 3 (3 различных).
           Минимальная сумма: 100,00 ₽.
           Максимальная сумма: 345,67 ₽.
           Средняя сумма: 222,83 ₽.
       Статистика по датам
           Число дат: 3 (3 различных).
           Минимальная дата: 23 мая 2025 г..
           Максимальная дата: 30 мая 2025 г..
           Средняя дата: 26 мая 2025 г..
       ```
2. Вы можете рассчитывать на то, что весь текст помещается в память.
3. При выполнении задания следует обратить внимание на:
    1. Декомпозицию сообщений для локализации.
    2. Согласование сообщений по роду и числу.
4. Приложение должно находиться в пакете `info.kgeorgiy.ja.*.i18n` и его подпакетах.