**1.**

Давай напишем архиватор. **Архиватор**, как минимум, должен уметь архивировать и разархивировать файлы.

**Начнем с первого.**  
Нам потребуется менеджер архива. Он будет совершать операции над файлом архива (*файлом, который будет храниться на диске и иметь расширение****zip***).  
Класс, который будет этим заниматься, назовем **ZipFileManager**.  
А главный класс приложения "***Архиватор***" будет называться **Archiver**.  
В программировании и не только, есть понятие полного (*абсолютного*) и относительного пути.  
Для начала, разберемся что-же такое путь вообще.  
Путь (*англ. Path*) - это набор символов, который показывает, где в операционной системе находится какой-то файл или папка.  
Полный или абсолютный путь - это путь, начинающийся с корневой директории.  
В операционной системе ***Windows***, корневой директорией принято считать диск.  
Пример полного пути в Windows: *C:\user\zips\Test1.zip*.  
**Относительный путь** - это путь относительно какой-то директории.  
***zips\Test1.zip*** - это пример относительного пути файла ***Test1.zip*** относительно директории (*папки*) ***C:\user***.  
Относительный путь, относительно директории ***C:\user\zips*** будет просто ***Test1.zip*** и совпадать с именем файла.  
Обрати внимание, что по умолчанию, и полный, и относительный путь к файлу, включают в себя имя файла.

1. Создай класс менеджер **ZipFileManager**  
2. Добавь в класс приватную переменную **Path** zipFile.  
В ней мы будем хранить полный путь к архиву, с которым будем работать.  
3. Добавь конструктор ZipFileManager(Path **zipFile**).  
Проинициализируй поле класса zipFile.  
4. Объяви публичный метод createZip(Path **source**) throws Exception, пока с пустой реализацией.  
**Path** source - это путь к чему-то, что мы будем архивировать.  
5. Создай класс **Archiver** и добавь в него метод main.  
6. В методе main:  
6.1 Запроси пользователя ввести полный путь архива с клавиатуры.  
Не забудь, что имя тоже входит в состав полного пути.  
6.2 Создай объект класса **ZipFileManager**, передав в него имя файла архива.  
Разберись, как из String получить **Path**.

**Подсказка:** *изучи метод*get()*класса***Paths***.*

6.3 Запроси пользователя ввести путь к файлу, который будем архивировать.  
Не путай это с файлом архива, который мы уже ввели.  
На этот раз нам нужен файл, который мы будем сжимать, а не в котором хранить сжатые данные.  
6.4 Вызови метод createZip у объекта **ZipFileManager**, передав в него путь для архивации.

**Требования:**

* •

Создай класс ZipFileManager.

* •

Внутри класса ZipFileManager должно быть создано приватное поле Path zipFile.

* •

Внутри класса ZipFileManager должен быть создан конструктор, который будет инициализировать поле zipFile.

* •

Внутри класса ZipFileManager должен быть создан публичный метод void createZip(Path source) throws Exception.

* •

Создай класс Archiver и добавь в него метод main. Реализуй его согласно заданию.

**2.**

Сейчас мы напишем реализацию метода createZip(Path **source**), в котором мы будем архивировать файл, заданный переменной source.  
В **Java** есть специальный класс **ZipOutputStream** из пакета *java.util.zip*, который сжимает (*архивирует*) переданные в него данные.  
Чтобы несколько файлов, сжимаемые в один архив, не слиплись вместе, для каждого из них создается специальная сущность - элемент архива **ZipEntry**.  
Т.е. в **ZipOutputStream** мы сначала кладем **ZipEntry**, а затем уже записываем содержимое файла.  
При записи файл автоматически сжимается, а при чтении - автоматически восстанавливается.  
**ZipEntry** может быть не только файлом, но и папкой.

**Чтобы заархивировать файл (*создать новый архив и добавить в него файл*):**  
1. Создай новый поток архива **ZipOutputStream** используя переменную класса **zipFile**, с помощью метода newOutputStream класса **Files**.  
2. Создай новый элемент архива **ZipEntry**.  
В конструктор **ZipEntry** передай строку, содержащую имя новой записи.  
Имя нужно получить из полного пути source, взять только имя файла и сконвертировать его в String.  
3. Добавь в поток архива созданный элемент архива.  
4. Перепиши данные из файла, который архивируем в поток архива. **Для этого:**  
4.1. Создай поток **InputStream** для добавляемого файла **source**, используя метод newInputStream класса **Files**  
4.2. Сделай цикл, который будет читать данные из **InputStream** (*созданного в п.4.1*), пока они там есть и записывать их в **ZipOutputStream** (*созданный в п.1*)  
4.3. Закрой **InputStream**, сделай это с помощью **try-with-resources**  
5. Закрой элемент архива у потока архива  
6. Закрой поток архива, сделай это также с помощью **try-with-resources**  
7. Запусти программу и проверь, что файл архивируется

**Требования:**

* •

Метод createZip должен создавать ZipOutputStream используя поле zipFile и метод Files.newOutputStream.

* •

Метод createZip должен создавать элемент архива ZipEntry c именем файла, полученным из параметра source.

* •

Созданный ZipEntry нужно добавить в ZipOutputStream.

* •

Для переменной source должен быть создан InputStream с помощью метода Files.newInputStream.

* •

Данные из InputStream нужно переписать в ZipOutputStream.

* •

Закрой текущий Entry у объекта ZipOutputStream.

* •

InputStream для source должен быть закрыт.

* •

ZipOutputStream должен быть закрыт.

**3.**

Как видишь, архивировать оказалось не так уж и сложно. Но наш архиватор получился каким-то уж слишком примитивным.  
Настоящий архиватор должен уметь гораздо больше: ***распаковку архива, добавление нового файла в существующий архив, удаление файла из архива, просмотр содержимого архива***.  
Сейчас мы будем улучшать наш архиватор. А для этого придется написать несколько новых классов.  
Сначала давай создадим enum Operation, который будет содержать все команды, которые поддерживает наш архиватор.  
Так же было бы удобно использовать **ConsoleHelper** для работы с консолью, чтобы все что касается консоли было собрано в одном классе.  
В дальнейшем, если не указано обратного, то весь ввод и вывод должен происходить через **ConsoleHelper**.

1. Объяви enum**Operation**, в него добавить команды:  
1.1. Создать архив CREATE  
1.2. Добавить файл в архив ADD  
1.3. Удалить файл из архива REMOVE  
1.4. Извлечь содержимое архива EXTRACT  
1.5. Просмотреть содержимое архива CONTENT  
1.6. Выйти из программы EXIT  
2. Создай класс **ConsoleHelper** и реализуй в нем статические публичные методы:  
2.1. Вывести сообщение в консоль void writeMessage(String **message**)  
2.2. Прочитать строку с консоли String readString()  
2.3. Прочитать число с консоли int readInt()  
Методы чтения с консоли могут бросать исключение **IOException** в случае ошибки ввода, учти это при их объявлении.

**Требования:**

* •

Создай enum Operation. Он должен содержать следующие значения: CREATE, ADD, REMOVE, EXTRACT, CONTENT, EXIT.

* •

Создай класс ConsoleHelper.

* •

В классе ConsoleHelper нужно реализовать статический метод void writeMessage(String message), который будет выводить message в консоль.

* •

В классе ConsoleHelper нужно реализовать статический метод String readString(), который будет считывать с консоли строку.

* •

В классе ConsoleHelper нужно реализовать статический метод int readInt(), который будет считывать с консоли число.

**4.**

Каждая команда подразумевает выполнение каких-то действий. Создадим интерфейс **Command**, с методом execute() (**execute** - "*выполнить" по-английски*).  
Для каждой команды мы создадим свой собственный класс. Все классы команд должны реализовывать (*быть унаследованы*) интерфейс **Command**.  
Так как команд будет много, отведем для них отдельный пакет command.  
Все интерфейсы и реализации команд будем хранить именно в нем.  
Самая простая команда - это выход EXIT, с нее и начнем.  
1. Создай пакет **command**  
2. В нем объявить интерфейс **Command**  
3. Добавь метод void execute() throws Exception в интерфейс **Command**  
4. Объяви класс **ExitCommand**, реализующий интерфейс **Command**  
5. Реализуй метод execute() в классе **ExitCommand**, он должен выводить "*До встречи!*" с помощью метода из класса **ConsoleHelper**  
6. В самом конце метода main в класса **Archiver** добавь код, который создает объект типа **ExitCommand** и вызывает у него метод execute()  
7. Попробуй, как это все работает  
Обрати внимание, что все файлы проекта должны быть в кодировке **UTF-8**.  
Кодировку в **IntelliJ IDEA** можно задать через пункты меню Settings -> Editor -> File Encodings.  
Проверь, что все три поля отвечающие за кодировку выставлены в **UTF-8**.

**Требования:**

* •

В пакете command должен быть создан интерфейс Command.

* •

В Command должен быть объявлен метод void execute() throws Exception.

* •

Создай класс ExitCommand в пакете command. Он должен реализовывать Command.

* •

Метод execute в классе ExitCommand должен выводить в консоль "До встречи!" с использованием ConsoleHelper.

* •

В конце main нужно добавить создание объекта ExitCommand и вызов у него метода execute.

**5.**

**Разделим команды на два типа**: те, которые работают непосредственно с архивом и вспомогательные (например EXIT).  
Все команды первого типа, будут иметь общий функционал, его удобно вынести в какой-то их общий базовый класс.  
Назовем этот класс **ZipCommand**.  
Он, как и все классы команд, должен реализовывать интерфейс **Command**.  
Все команды, которые работают с архивом, должны быть унаследованы от класса **ZipCommand**.  
Мы не будем создавать объекты класса **ZipCommand**, поэтому сделаем его абстрактным.

1. Создай абстрактный класс **ZipCommand**, реализующий интерфейс **Command**  
2. Создай по одному классу для каждой команды. Все перечисленные команды должны быть унаследованы от **ZipCommand** и содержать пустую реализацию метода execute(), его реализацию мы будем писать для каждой команды отдельно по мере реализации нашего архиватора.  
2.1. Команда создания архива (*упаковки файлов в архив*) - **class ZipCreateCommand**  
2.2. Команда просмотра содержимого архива - **class ZipContentCommand**  
2.3. Команда распаковки архива - **class ZipExtractCommand**  
2.4. Команда добавления файла в архив - **class ZipAddCommand**  
2.5. Команда удаления файла из архива - **class ZipRemoveCommand**

**Требования:**

* •

В пакете command должен быть создан абстрактный класс ZipCommand реализующий интерфейс Command.

* •

В пакете command должен быть создан класс ZipCreateCommand унаследованный от ZipCommand.

* •

В пакете command должен быть создан класс ZipContentCommand унаследованный от ZipCommand.

* •

В пакете command должен быть создан класс ZipExtractCommand унаследованный от ZipCommand.

* •

В пакете command должен быть создан класс ZipAddCommand унаследованный от ZipCommand.

* •

В пакете command должен быть создан класс ZipRemoveCommand унаследованный от ZipCommand.

**6.**

Представь, что пользователь сообщает нам с помощью переменной **Operation** operation, что он хочет сделать.  
Тогда мы должны проверить значение этой переменной, создать объект соответствующего класса команды и вызвать у него метод execute().  
Чтобы не создавать объект класса нужной команды каждый раз, его нужно где-то хранить.  
Создадим для этих целей класс **CommandExecutor**, пусть у него будет открытым только один публичный статический метод execute(Operation **operation**), который найдет нужную команду и вызовет у нее метод execute.

1. Создай класс **CommandExecutor**.  
Он должен быть в корне задачи, не стоит добавлять его в пакет **command**  
2. Запрети явный вызов конструктора этого класса  
3. Добавь в класс приватное статическое константное хранилище команд Map<Operation, Command> **allKnownCommandsMap**  
4. Проинициализируй переменную allKnownCommandsMap так, чтобы каждому значению из **Operation** соответствовала правильная команда.  
5. Реализуй публичный статический метод execute(Operation **operation**) throws Exception, который должен брать нужную команду из allKnownCommandsMap и вызывать у нее метод execute.  
Догадался ты или нет, но ты только что реализовал паттерн Команда!

**Требования:**

* •

В корне задачи должен быть создан класс CommandExecutor.

* •

Запрети явный вызов конструктора CommandExecutor.

* •

В классе CommandExecutor нужно создать приватное статическое константное поле Map<Operation, Command> allKnownCommandsMap.

* •

Поле allKnownCommandsMap должно быть проинициализировано всеми значениями Operation и соответствующими Command-классами.

* •

В классе CommandExecutor нужно реализовать метод execute(Operation operation), который будет вызывать execute у нужной команды из allKnownCommandsMap.

**7.**

Мы много всего сделали и можно немного передохнуть.  
Создадим, пакет exception и добавим в него два класса **PathIsNotFoundException** и **WrongZipFileException**.  
Исключение **PathIsNotFoundException** будем кидать, если не сможем найти путь, в который нужно распаковать архив, или путь к файлу, который хотим запаковать, или любой другой путь. Исключение **WrongZipFileException** будем кидать, если будет попытка сделать что-нибудь с архивом, который не существует.

1. Создай пакет **exception**  
2. Добавь в него класс **PathIsNotFoundException** унаследованный от **Exception**  
3. Добавь класс **WrongZipFileException**, также унаследованный от **Exception**

**Требования:**

* •

В пакете exception должен быть создан класс PathIsNotFoundException унаследованный от Exception.

* •

В пакете exception должен быть создан класс WrongZipFileException унаследованный от Exception.

**8.**

Чтобы узнать какую команду сейчас хочет выполнить пользователь, добавим метод Operation askOperation() в класс **Archiver**.  
Этот метод должен вывести в консоль список доступных команд и попросить выбрать одну из них.  
Для удобства будем просить ввести номер команды, где номер - это порядковый номер команды в enum **Operation**.  
Получить порядковый номер значения в enum'е можно с помощью метода ordinal().  
Теперь все готово чтобы переписать main, используя последние достижения науки и техники, а именно класс **CommandExecutor** и метод askOperation().  
1. Добавь публичный статический метод Operation askOperation() throws IOException в класс **Archiver**.  
**Он должен:**  
1.1. Использовать методы класса **ConsoleHelper**  
1.2. Запрашивать у пользователя номер операции, которую он хочет совершить.

**Подсказка:**  
*чтобы вывести номер операции "Создать архив", используй:*Operation.CREATE.ordinal()

1.3. Возвращать выбранную операцию.

**Пример вывода метода askOperation():**  
Выберите операцию:  
0 - упаковать файлы в архив  
1 - добавить файл в архив  
2 - удалить файл из архива  
3 - распаковать архив  
4 - просмотреть содержимое архива  
5 - выход  
2. Перепиши метод main():  
2.1. Объяви локальную переменную типа Operation  
2.2. В цикле запрашивай новое значение для переменной **п.2.1.** с помощью метода askOperation() и вызывай выполнение операции с помощью CommandExecutor.execute()  
2.3. Обеспечь выход из цикла, если пользователь выбрал операцию Operation.EXIT  
2.4. Оберни вызов askOperation() и execute(operation) в блок **try-catch**.  
Если произойдет исключение **WrongZipFileException** выведи сообщение "*Вы не выбрали файл архива или выбрали неверный файл.*" с помощью ConsoleHelper, при любых других исключениях выводи "*Произошла ошибка. Проверьте введенные данные."*.  
2.5. Проследи, чтобы программа продолжила свою работу (*перешла на новый шаг цикла*), после обработки исключений.  
3. Запусти программу и проверь, что команда "*выход*" работает.

**Требования:**

* •

В классе Archiver нужно создать публичный статический метод Operation askOperation().

* •

Метод askOperation должен вывести в консоль все возможные операции и их номер.

* •

Метод askOperation должен считать с клавиатуры номер выбранной операции и вернуть ее.

* •

Перепиши метод main согласно заданию.

**9.**

Теперь займемся другой, не менее важной частью нашего архиватора.  
Очень часто пользователь хочет создать архив не из одного файла, а из целой папки.  
Тогда архивация сводится к поочередному добавлению элемента **ZipEntry** для каждого файла в архив.  
Нам нужно написать класс **FileManager** для получения списка всех файлов в какой-то папке.  
У него должен быть конструктор, который будет принимать путь **Path** rootPath, указывающий на корень (*папку, файлы в которой нас интересуют*) и метод List<Path> getFileList(), который должен возвращать список относительных путей всех файлов, которые находятся по пути rootPath, включая файлы в подпапках.  
Директория, в которой будем искать, может задаваться только один раз в конструкторе.  
Поэтому, прямо в конструкторе мы сразу можем сформировать список файлов (их относительных путей), сохранив его в переменную класса List<Path>**fileList**.  
Осталось только решить, как нам собрать все файлы, даже те, которые могут быть в папке, которая в папке, которая в папке, которая в интересующей нас папке. Фух, чуть не вошел рекурсию.  
Хотя нам-то она и поможет! Сделаем метод collectFileList(Path **path**), который будет складывать в переменную класса **fileList** все файлы, обнаруженные внутри переданного пути path, вызывая сам себя для всех объектов, в обнаруженных директориях.

1. Создай класс **FileManager** с конструктором FileManager(Path **rootPath**) throws IOException  
2. Объяви и проинициализируй приватные переменные класса:  
2.1. Path rootPath - корневой путь директории, файлы которой нас интересуют  
2.2. List<Path> **fileList** - список относительных путей файлов внутри rootPath  
3. Создай **геттер** для fileList  
4. Реализуй метод void collectFileList(Path path) throws IOException, который должен:  
4.1. Проверить, если переданный путь path является обычным файлом (*используй метод*Files.isRegularFile), то получить его относительный путь относительно rootPath и добавить его в список fileList.  
4.2. Если переданный путь path, является директорией (*узнать это поможет метод*Files.isDirectory), то пройтись по всему содержимому директории и вызвать collectFileList(Path **path**), передав в path обнаруженные элементы.  
Пройтись по всему содержимому директории можно предварительно получив DirectoryStream с помощью метода newDirectoryStream класса **Files**.  
Не забудь закрыть созданный **DirectoryStream**.  
5. Добавь вызов метода collectFileList(**rootPath**) в конструкторе **FileManager**.  
6. Примени все свои знания об инкапсуляции к этому классу.  
Выполняя это задание, ты написал алгоритм, который обходит дерево файлов.  
Но в **Java** есть специальный интерфейс **FileVisitor** для этих целей.  
Очень рекомендую разобраться как им пользоваться.

**Требования:**

* •

В корне задачи должен быть создан класс FileManager c конструктором FileManager(Path rootPath) throws IOException.

* •

В классе FileManager должны быть созданы приватные поля Path rootPath и List<Path> fileList, которые инициализируются в конструкторе.

* •

В классе FileManager должен быть создан getter для поля fileList.

* •

В классе FileManager нужно реализовать приватный метод void collectFileList(Path path) throws IOException согласно заданию.

* •

Нужно добавить вызов метода collectFileList(rootPath) в конструктор FileManager.

**10.**

Давай наконец попробуем нормально реализовать метод execute() класса **ZipCreateCommand**, который мы добавляли раньше.

**Для этого нужно:**  
1. В начале метода добавить вывод сообщения "*Создание архива.*" Не забудь, что мы работаем с консолью через методы класса **ConsoleHelper**.  
2. Создай новый объект класса **ZipFileManager**. Т.к. создание этого объекта будет необходимо и другим командам, вынеси создание в отдельный метод ZipFileManager getZipFileManager() throws Exception в класс **ZipCommand**.  
**Этот метод должен:**  
2.1. Просить пользователя ввести полный путь файла архива  
2.2. Считывать введенный путь в переменную типа String  
2.3. Используя введенный String формировать путь **Path**  
2.4. Создать объект **ZipFileManager**, передав в конструктор полученный путь  
2.5. Вернуть созданный объект  
3. Попроси пользователя ввести полное имя файла или директории для архивации  
4. Создай путь **Path**, используя введенную строку  
5. У объекта класса **ZipFileManager** вызови метод createZip(), передав в него путь из п.4  
6. Выведи в консоль сообщение "*Архив создан.*"  
7. Оберни содержимое метода execute() класса **ZipCreateCommand** в блок **try-catch** и отлавливай  
исключение типа **PathIsNotFoundException**. Если оно произошло, выведи сообщение "*Вы неверно указали имя файла или директории.*"  
8. Запусти программу и проверь, что команда "*упаковать файлы в архив*" работает

**Требования:**

* •

В классе ZipCommand нужно добавить public метод ZipFileManager getZipFileManager() throws Exception.

* •

Метод getZipFileManager должен считывать с консоли путь к архиву и возвращать для него объект ZipFileManager.

* •

Метод execute в классе ZipCreateCommand должен быть реализован следуя описанию в задании, используя метод getZipFileManager.

* •

В случае возникновения исключения PathIsNotFoundException, нужно вывести в консоль сообщение "Вы неверно указали имя файла или директории.".

**11.**

Давай наконец попробуем нормально реализовать метод execute() класса **ZipCreateCommand**, который мы добавляли раньше.

**Для этого нужно:**  
1. В начале метода добавить вывод сообщения "*Создание архива.*" Не забудь, что мы работаем с консолью через методы класса **ConsoleHelper**.  
2. Создай новый объект класса **ZipFileManager**. Т.к. создание этого объекта будет необходимо и другим командам, вынеси создание в отдельный метод ZipFileManager getZipFileManager() throws Exception в класс **ZipCommand**.  
**Этот метод должен:**  
2.1. Просить пользователя ввести полный путь файла архива  
2.2. Считывать введенный путь в переменную типа String  
2.3. Используя введенный String формировать путь **Path**  
2.4. Создать объект **ZipFileManager**, передав в конструктор полученный путь  
2.5. Вернуть созданный объект  
3. Попроси пользователя ввести полное имя файла или директории для архивации  
4. Создай путь **Path**, используя введенную строку  
5. У объекта класса **ZipFileManager** вызови метод createZip(), передав в него путь из п.4  
6. Выведи в консоль сообщение "*Архив создан.*"  
7. Оберни содержимое метода execute() класса **ZipCreateCommand** в блок **try-catch** и отлавливай  
исключение типа **PathIsNotFoundException**. Если оно произошло, выведи сообщение "*Вы неверно указали имя файла или директории.*"  
8. Запусти программу и проверь, что команда "*упаковать файлы в архив*" работает

**Требования:**

* •

В классе ZipCommand нужно добавить public метод ZipFileManager getZipFileManager() throws Exception.

* •

Метод getZipFileManager должен считывать с консоли путь к архиву и возвращать для него объект ZipFileManager.

* •

Метод execute в классе ZipCreateCommand должен быть реализован следуя описанию в задании, используя метод getZipFileManager.

* •

В случае возникновения исключения PathIsNotFoundException, нужно вывести в консоль сообщение "Вы неверно указали имя файла или директории.".

**12.**

Сегодня мы подготовимся к реализации команды ZipContentCommand. Она будет заниматься получением содержимого архива.  
Содержимое архива - это упакованные файлы и папки, но нам было бы интересно узнать не только имена объектов архива, но и их размер до и после сжатия, степень сжатия и метод сжатия.  
Создадим класс **FileProperties**, который будет отвечать за свойства каждого файла в архиве.  
Свойства - это набор, состоящий из: имя файла, размер файла до и после сжатия, метод сжатия.  
1. Создай класс **FileProperties**  
2. Добавь в него приватные переменные класса:  
2.1. Имя String name  
2.2. Размер в байтах long size  
2.3. Размер после сжатия в байтах long compressedSize  
2.4. Метод сжатия int compressionMethod  
3. Добавь гетеры для них  
4. Добавь конструктор FileProperties(String **name**, long **size**, long **compressedSize**, int **compressionMethod**)  
5. Добавь метод long getCompressionRatio(), который будет считать степень сжатия по формуле: 100 - ((compressedSize \* 100) / size)  
6. Переопредели метод String toString(), чтобы он возвращал строку по шаблону:  
"***name size Kb (compressedSize Kb) сжатие: compressionRatio%***", если размер **size** больше нуля, иначе он должен вернуть только имя файла.  
Нулевой размер может быть, например, у директории. Не забудь перевести байты в килобайты, а их не столько же, сколько граммов в килограмме, и даже не столько, сколько блинов у меня на столе... Хм, похоже мне пора перекусить...

**Требования:**

* •

В корне задачи нужно создать класс FileProperties.

* •

Класс FileProperties должен содержать поля String name, long size, long compressedSize, int compressionMethod.

* •

Класс FileProperties должен содержать конструктор FileProperties(String name, long size, long compressedSize, int compressionMethod).

* •

Для всех полей в FileProperties должны быть созданы геттеры.

* •

Класс FileProperties должен содержать публичный метод long getCompressionRatio(), который считает степень сжатия файла.

* •

Метод toString у класса FileProperties должен быть реализован следуя описанию в задании.

**13.**

Продолжим наш путь к получению содержимого файла архива. Напишем метод getFilesList() в классе **ZipFileManager**.  
Он будет возвращать список файлов в архиве, вернее список свойств этих файлов (*класс свойств***FileProperties** *мы уже реализовали*).  
**Итак:**  
1. Добавь метод List<FileProperties> getFilesList() throws Exception в класс **ZipFileManager**  
2. Внутри метода проверь является ли содержимое **zipFile** обычным файлом с помощью подходящего метода класса **Files**. Если это не файл, брось исключение WrongZipFileException().  
3. Создай список с элементами типа **FileProperties**, в него мы будем складывать свойства файлов  
4. Создай входящий поток **ZipInputStream**, для файла из переменной **zipFile**.  
Как и в прошлые разы, оберни его создание в **try-with-resources**  
5. Пройдись по всем элементам **ZipEntry** потока **ZipInputStream**  
6. Для каждого элемента **ZipEntry** вычитай его содержимое, иначе у нас не будет информации о его размере.  
Нельзя узнать размер файла в архиве, не вычитав его. Это очень легко сделать с помощью функции copyData, используя временный буфер типа **ByteArrayOutputStream**.  
7. Получи имя, размер, сжатый размер и метод сжатия элемента архива. Посмотри, что еще можно узнать о нем.  
8. Создай объект класса **FileProperties**, используя полученные данные о файле.  
9. Добавь созданный объект из **п.8** в список из **п.3**  
10. После выхода из цикла верни собранную информацию вызвавшему методу.

**Требования:**

* •

В классе ZipFileManager нужно создать публичный метод List<FileProperties> getFilesList() throws Exception.

* •

Если Path zipFile не является файлом, метод getFilesList должен кинуть исключение WrongZipFileException.

* •

Метод getFilesList должен пройтись по всем файлам архива и вернуть данные о них. Реализация должна быть выполнена согласно описанию в задании.

* •

Поток для чтения из архива должен быть закрыт.

**14.**

Все готово, чтобы реализовать метод execute() класса **ZipContentCommand**:  
1. Выведи сообщение "***Просмотр содержимого архива.***"  
2. Создай объект класса **ZipFileManager** с помощью метода getZipFileManager()  
3. Выведи сообщение "***Содержимое архива:***"  
4. Получи список файлов архива с помощью метода getFilesList()  
5. Выведи свойства каждого файла в консоль. Тут нам и пригодится ранее реализованный метод toString() класса **FileProperties**  
6. Выведи сообщение "***Содержимое архива прочитано.***"  
7. Запусти программу и проверь, что команда "***просмотреть содержимое архива***" работает

**Требования:**

* •

В методе execute класса ZipContentCommand нужно получить объект ZipFileManager.

* •

В методе execute класса ZipContentCommand у объекта ZipFileManager нужно получить список файлов в архиве.

* •

В методе execute класса ZipContentCommand нужно вывести свойства каждого файла в архиве на консоль.

**15.**

Пора попробовать что-нибудь распаковать. Для этого добавим публичный метод void extractAll(Path outputFolder) throws Exception в класс **ZipFileManager**.  
**Path** outputFolder - это путь, куда мы будем распаковывать наш архив. У тебя уже большой опыт работы с элементами архива и потоками. Так что, я дам только подсказки по реализации этого метода, а тебе придется хорошенько подумать, как это все сделать:  
1. Проверь, есть ли **zip** файл вообще  
2. Если директория **outputFolder** не существует, то ее нужно создать, как и все папки, внутри которых она лежит.  
3. Внутри архива некоторые файлы могут лежат внутри папок, тогда метод getName() элемента архива **ZipEntry**, вернет не совсем имя, как может показаться из названия, а относительный путь внутри архива. Этот относительный путь должен сохраниться и после распаковки, но уже относительно той директории, куда мы распаковали архив  
4. Реализуй метод execute() класса **ZipExtractCommand**, по аналогии с таким же методом класса **ZipCreateCommand**, сделай такой же блок **try-catch**, только поменяй сообщения выводимые пользователю, чтобы он понял, что сейчас мы будем распаковывать архив, и что нужно ввести полное имя архива и директорию, куда будем распаковывать.  
Не забудь вызвать метод extractAll класса **ZipFileManager**, а не **createZip**, как это было в ZipCreateCommand  
5. Запускай программу и наслаждайся результатом распаковки

**Примечание:**  
Для получения потоков чтения и записи используй Files.newInputStream(Path path), Files.newOutputStream(Path path).

**Требования:**

* •

В классе ZipFileManager нужно создать публичный метод void extractAll(Path outputFolder) throws Exception.

* •

Метод extractAll(Path outputFolder) должен бросать исключение WrongZipFileException, если файл архива не существует.

* •

Метод extractAll(Path outputFolder) должен создавать директорию outputFolder, если она не существует.

* •

Метод extractAll(Path outputFolder) должен распаковывать все файлы из архива в директорию outputFolder.

* •

Метод execute() в классе ZipExtractCommand должен получать объект ZipFileManager и разархивировать файлы в указанную пользователем директорию.

**16.**

Пришло время что-нибудь удалить из архива. Архив очень хитрая штука, нельзя вот так просто взять и удалить какой-то элемент внутри него.  
Почему? Представь, что мы решили сами придумать свой алгоритм сжатия текста. Посмотрев исходный текст, мы видим, что в нем часто встречается фраза "***быть программистом круто***". Мы можем в месте, где второй, третий, **N**-ый раз встречается наша фраза сделать пометку, что тут была фраза, как в строке **S** начиная с символа номер **K** и длиной **N**, а саму фразу удалить. Когда мы заменим много повторяющихся фраз, текст заметно сократится, но станет нечитаемым для тех, кто не знаком с нашим алгоритмом сжатия.  
Мы же этот текст сможем восстановить (*разархивировать*).  
А теперь представь, что нам нужно удалить часть текста, на которую ссылались сжатые фрагменты. В такой ситуации, весь наш архив перестанет иметь смысл. Вот почему нельзя просто так удалить часть архива.  
Это очень примерное описание варианта архивации, в реальности все намного сложнее.  
Поэтому, чтобы что-то удалить из архива, нужно создать новый архив, переписать в него все, кроме удаляемых файлов, а потом заменить старый архив вновь созданным.

1. Добавь публичный метод для удаления файлов из архива void removeFiles(List<Path> pathList) throws Exception в класс **ZipFileManager**.  
В **pathList** будет передаваться список относительных путей на файлы внутри архива.  
**Он должен:**  
1.1. Бросать исключение **WrongZipFileException**, если файл архива не существует.  
1.2. Создать временный файл архива в директории по умолчанию с помощью метода createTempFile() класса **Files**.  
1.3. Пройтись по всем файлам оригинального архива, проверить, есть ли текущий файл в списке на удаление.  
- Если файл есть в списке, вывести сообщение, что мы удалили файл с таким-то именем и перейти к следующему файлу.  
- Если файла нет в списке на удаление, переписать его в новый архив  
1.4. Заменить оригинальный файл архива временным, в который мы записали нужные файлы.  
Это нужно сделать с помощью метода move() класса **Files**  
2. Добавь публичный метод void removeFile(Path path) throws Exception в класс **ZipFileManager**, который будет вызывать метод removeFiles, создавая список из одного элемента.  
Это можно сделать с помощью метода singletonList() класса **Collections**. Посмотри, как он работает.  
3. Реализуй метод execute() класса **ZipRemoveCommand**, создав объект класса **ZipFileManager**, спросив пользователя из какого архива и какой файл будем удалять, и вызвав метод removeFile().  
Все остальное, как и в других командах. Исключение **PathIsNotFoundException** можно не ловить, т.к. метод removeFile() не должен его кидать.  
4. Запусти программу и проверить, что удаление файла из архива работает.

**Требования:**

* •

В классе ZipFileManager нужно создать публичный метод void removeFiles(List<Path> pathList) throws Exception.

* •

Метод removeFiles должен бросать исключение WrongZipFileException, если файл архива не существует.

* •

Метод removeFiles должен создавать временный файл архива с помощью метода Files.createTempFile.

* •

Метод removeFiles должен записывать в новый архив все файлы из старого архива, кроме тех, которые в списке на удаление. Затем, заменить старый архив новым.

* •

В классе ZipFileManager нужно создать публичный метод void removeFile(Path path) throws Exception, который будет делегировать свое выполнение методу removeFiles(List<Path> pathList).

* •

Метод execute() в классе ZipRemoveCommand должен получить объект ZipFileManager и удалить из архива файл, считанный с консоли.

**17.**

Осталась ерунда. Добавить добавление файла в архив. Звучит подозрительно, но именно этим мы и займемся.  
Добавление файлов похоже на удаление, мы создаем временный файл архив, переписываем в него все содержимое старого архива и добавляем новые файлы. Потом заменяем старый файл архива новым.  
1. Добавь публичный метод void addFiles(List<Path> absolutePathList) throws Exception в класс **ZipFileManager**, где absolutePathList - список абсолютных путей добавляемых файлов.  
**Этот метод должен:**  
1.1. Как обычно, бросать исключение **WrongZipFileException**, если файл архива не существует  
1.2. Создать временный файл архива  
1.3. Пройти по всем файлам оригинального архива, переписать каждый файл в новый архив, добавить имя переписанного файла в какой-нибудь локальный список.  
1.4. Пройтись по списку добавляемых файлов.  
1.5. Для каждого файла проверить, есть ли он на диске и является ли он обычным файлом, если что-то не так, кинь исключение PathIsNotFoundException()  
1.6. Проверить, есть ли добавляемый файл уже в архиве (*используй список из п.1.3*). Такое возможно, если пользователь уже когда-то добавлял его.  
- Если файла нет в списке, добавь его в новый архив, выведи сообщение, что такой-то файл добавлен в архив  
- Если файл есть в списке, просто сообщи пользователю, что такой файл уже есть в архиве  
1.7. Заменить оригинальный файл архива временным, в котором уже есть добавленные файлы.  
2. Добавь публичный метод void addFile(Path absolutePath) throws Exception в класс **ZipFileManager**, реализуй его с помощью вызова метода addFiles(), аналогично тому, как мы это делали для удаления файла.  
3. Реализуй метод execute() класса **ZipAddCommand**: все как обычно, но не забудь спросить у пользователя в какой архив и какой файл он хочет добавить, обработай исключение **PathIsNotFoundException**, которое может кинуть метод addFile().  
4. Запусти программу и проверить, что добавление файла теперь работает.

**Требования:**

* •

В классе ZipFileManager нужно создать публичный метод void addFiles(List<Path> absolutePathList) throws Exception.

* •

Метод addFiles должен бросать исключение WrongZipFileException, если файл архива не существует.

* •

Метод addFiles должен создавать временный файл архива с помощью метода Files.createTempFile.

* •

Метод addFiles должен проходить по списку добавляемых файлов и бросить PathIsNotFoundException, если какой-то файл не существует.

* •

Метод addFiles должен записывать в новый архив новые файлы и все файлы из старого архива. Затем, заменить старый архив новым.

* •

В классе ZipFileManager нужно создать публичный метод void addFile(Path absolutePath) throws Exception, который будет делегировать свое выполнение методу addFiles(List<Path> absolutePathList).

* •

Метод execute() в классе ZipAddCommand должен получить объект ZipFileManager и добавить в архив файл, считанный с консоли.

**18.**

Ты ***отличный ученик,*** я немного подправил твой код, можешь использовать этот архиватор для архивации в повседневной жизни. Если будет время и желание, попробуй добавить операцию изменения степени сжатия архива!

**Требования:**

* Архиватор готов.