**Archiver (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:userzipsTest1.zip*.  
Относительный путь – это путь относительно какой-то директории. *zipsTest1.zip* — это пример  
относительного пути файла *Test1.zip* относительно директории (*папки*) *C:user*. Относительный путь,  
относительно директории *C:userzips* 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:  
1. Создай класс ZipFileManager.  
2. Внутри класса ZipFileManager должно быть создано приватное поле Path zipFile.  
3. Внутри класса ZipFileManager должен быть создан конструктор, который будет инициализировать поле zipFile.  
4. Внутри класса ZipFileManager должен быть создан публичный метод void createZip(Path source) throws Exception.  
5. Создай класс Archiver и добавь в него метод main. Реализуй его согласно заданию.

### Archiver (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

**Archiver (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:  
1. Создай enum Operation. Он должен содержать следующие значения: CREATE, ADD, REMOVE, EXTRACT, CONTENT, EXIT.  
2. Создай класс ConsoleHelper.  
3. В классе ConsoleHelper нужно реализовать статический метод void writeMessage(String message), который будет выводить message в консоль.  
4. В классе ConsoleHelper нужно реализовать статический метод String readString(), который будет считывать с консоли строку.  
5. В классе ConsoleHelper нужно реализовать статический метод int readInt(), который будет считывать с консоли число.

**Archiver (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:  
1. В пакете command должен быть создан интерфейс Command.  
2. В Command должен быть объявлен метод void execute() throws Exception.  
3. Создай класс ExitCommand в пакете command. Он должен реализовывать Command.  
4. Метод execute в классе ExitCommand должен выводить в консоль "“До встречи!”" с использованием ConsoleHelper.  
5. В конце main нужно добавить создание объекта ExitCommand и вызов у него метода execute.

**Archiver (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:  
1. В пакете command должен быть создан абстрактный класс ZipCommand реализующий интерфейс Command.  
2. В пакете command должен быть создан класс ZipCreateCommand унаследованный от ZipCommand.  
3. В пакете command должен быть создан класс ZipContentCommand унаследованный от ZipCommand.  
4. В пакете command должен быть создан класс ZipExtractCommand унаследованный от ZipCommand.  
5. В пакете command должен быть создан класс ZipAddCommand унаследованный от ZipCommand.  
6. В пакете command должен быть создан класс ZipRemoveCommand унаследованный от ZipCommand.

**Archiver (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:  
1. В корне задачи должен быть создан класс CommandExecutor.  
2. Запрети явный вызов конструктора CommandExecutor.  
3. В классе CommandExecutor нужно создать приватное статическое константное поле Map allKnownCommandsMap.  
4. Поле allKnownCommandsMap должно быть проинициализировано всеми значениями Operation и соответствующими Command-классами.  
5. В классе CommandExecutor нужно реализовать метод execute(Operation operation), который будет вызывать execute у нужной команды из allKnownCommandsMap.

### Archiver (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:**  
1. В пакете exception должен быть создан класс PathIsNotFoundException унаследованный от Exception.  
2. В пакете exception должен быть создан класс WrongZipFileException унаследованный от Exception.

### Archiver (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:**  
1. В классе Archiver нужно создать публичный статический метод Operation askOperation().  
2. Метод askOperation должен вывести в консоль все возможные операции и их номер.  
3. Метод askOperation должен считать с клавиатуры номер выбранной операции и вернуть ее.  
4. Перепиши метод main согласно заданию.

### Archiver (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:  
1. В корне задачи должен быть создан класс FileManager c конструктором FileManager(Path rootPath) throws IOException.  
2. В классе FileManager должны быть созданы приватные поля Path rootPath и List fileList, которые инициализируются в конструкторе.  
3. В классе FileManager должен быть создан getter для поля fileList.  
4. В классе FileManager нужно реализовать приватный метод void collectFileList(Path path) throws IOException согласно заданию.  
5. Нужно добавить вызов метода collectFileList(rootPath) в конструктор FileManager.