**Streams**

**(ввод/вывод)**

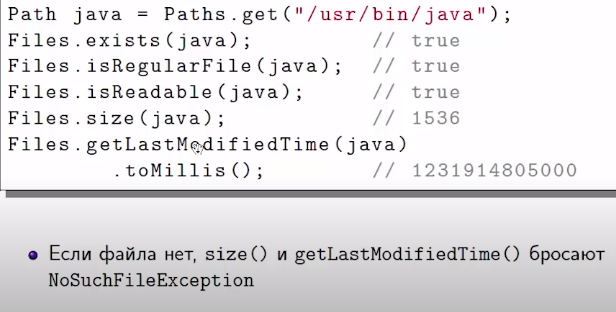
Поток — это по сути, перемещающийся кусок данных. То есть в программиро-вании по потоку течет не вода, а данные в виде байтов или символов.

**IO** – input output.

**NIO** – new input output (non-bloking I/O).

**File** – java.io.File класс предназначенный для работы с директориями и файлами, можно скормить любой путь, при объявлении и передачи пути если файла не существует – создаёт (методы: узнать это файл или папка, доступен для чтения/записи/удаления, узнать родителя, список файлов внутри, создать файл/папку, возможность преобразовать в класс Path или URL и etc). Это самый первый класс для работы с файлами и папками, сейчас есть более удобное решение, не умеет копировать файлы.

**Path -** nio.file.Path доработанный аналог класса File (например в File, был метод getParent(), который возвращал родительский путь для текущего файла в виде строки. Но при этом был метод getParentFile(), который возвращал то же самое, но в виде объекта File) в Path классе нет этого излишества, возвращаемое значение просто Path. Не может скопировать файл. Некоторые статические методы вынесены в специально созданный класс Files.



**Files** – очень полезный статический (утилитный) класс, набор различных методов, похож на один из утилитных классов Arrays или Collections, только работает он с файлами а не с массивами или коллекциями. Получить байты из файла:

byte[] bytes = Files.*readAllBytes*(path);

Есть возможность одним методом скопировать файл, есть методы по чтению 3шт (метод который считает и разобьёт текст по строкам readAllLines() и др), метод для записи в файл. Ещё есть метод walkFileTree() – который предназначен для удобного обхода папок.

**Java определяет два типа потоков**

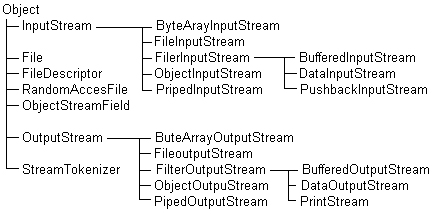
**Байтовые** – предоставляют удобные средства для обработки ввода и вывода байтов, используются для чтения и записи в двоичном коде. Для ввода / вывода используют 8 битные байты.

**Cимвольные** – предоставляют удобные средства для обработки ввода и вывода символов. Используют 16 битный Unicode для ввода/вывода. На самом низком уровне весь ввод/вывод всё равно – байтовый (все классы в java.io используют InputStream и OutputStream и уже под капотом добавляют свой функционал).

**BYTE STREAM (Байтовый поток)**

Читает байт за батом. Подходит для считывания необработанных данных, такие как двоичные коды. Обычно имена байтовых потоков заканчиваются на InputStream / OutputStream.

Иерархия классов батовых потоков ввода-вывода:



**InputStream** and **OutPutStream** – это абстрактные классы, предназначены для задания шаблона всем наследникам.

**ByteArrayInputStream** и **ByteArrayOutputStream** - удобно, когда нужно проверить, что именно записывается в выходной поток. Например, при отладке и тестировании сложных процессов записи и чтения из потоков. Эти классы хороши тем, что позволяют сразу просмотреть результат и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще (конструктор принимает параметр - массив байтов). Этот класс не нужно в обязательном порядке закрывать, как другие.

**FileOutputStream** и **FileInputStream** – специальный класс который предназначен для записи/вывода байтов в/из файл(а). Он максимально абстрактный и может записывать/считывать обьекты/звуки и др, что угодно в/из файл(а). При работе с FileInputStream – метод available() наверника вернёт размер в байтах доступный для считывания (но не на всех платформах) нужно быть аккуратным. И для считывания символов есть более удачные классы, он больше подходит для работы с байтовыми файлами.

**BufferedInputStream** и **BufferedOutPutStream** – удобный способ оптимизации производительности, позволяет заключать в оболочку любой поток класса InputStream or OutputStream.

**DataInputStream** и **DataOutPutStream** – предназначен для чтения байтовых данных (не строк). Есть возможность прочитать значение int,double,string и другое из потока байт.

**PipedInputStream** и **PipedOutputStream** – характеризуются тем что их обьекты всегда используются в паре, полезны, если в программе необходимо организовать обмен данными между модулями (например, между потоками выполнения). Эти классы применяются следующим образом: создается по объекту PipedInputStream и PipedOutputStream, после чего они могут быть соединены между собой. Один объект PipedOutputStream может быть соединен с ровно одним объектом PipedInputStream, и наоборот. Затем в объект PipedOutputStream записываются данные, после чего они могут быть считаны именно в подключенном объекте PipedInputStream. Такое соединение можно обеспечить либо вызовом метода connect() с передачей соответствующего объекта PipedI/OStream.

**ObjectInputStream** и **ObjectOutputStream** – класс который принимает в качестве параметра любой из классов OutputStream или InputStream (в зависимости от потока чтения или записи) и позволяет к примеру с помощью метода writeObject(Object o) преобразовать обьект в массив батов и спомощью обьекта в параметрах контруктора – произвести какие то действия. Если считываем то выполняем довнкастинг (прямое преобразование к нужному обьекту), так же должен существовать класс к которому преобразуем с имплементированным интерфейсом Serializable.

Более подробно о всех Stream классах:

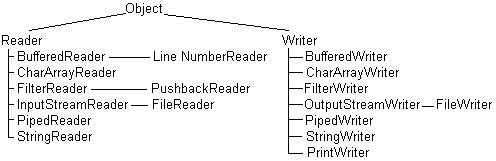
<https://wm-help.net/lib/b/book/3683783285/38>

**CHARACTER STREAM (Символьный поток)**

Используем, когда необходимо обрабатывать текстовые файлы. Обычно имена символьных потоков заканчиваются на Writer / Reader.

Если наследник Writer в конструктор принимает вторым параметром boolean – скорее всего это перезаписывать файл или добавлять в него значения.

Иерархия классов символьных потоков ввода-вывода:



**InputStreamReader –** принимает любой класс байтовой ветки InputStream и работает с ним к с символьным потоком.

**FileWriter** – у него есть конструктор, с параметром boolean, в котором можно указать что информация будет дописываться к файлу, а не перезаписываться. Если файла по пути не существует – создаёт файл.

**BufferedReader** – принимает в конструктор один из потоков Reader и позволяет буфферизировать информацию в своём буфере, так метод read() будет работать быстрее, и добавляет метод readLine().

Метод в writer.flash() можно не вызывать, так как writer.close() сделает это автоматом.

**ОБЩЕЕ**

В блоке finally все потоки обязательны к закрытию, закрывай постоянно в разных блоках try-catch, так как если ошибка закрытия произойдёт, следующий поток за ним в одном блоке try-catch не закроется.

java.io содержит также классы, преобразующие любые данные в набор байт.

Если нужно сохранить какой-то результат вычисления или обьект (сериализуют обьект) – его преобразуют для начала в набор байтов, после чего сохраняют в файл, те же действия можно проделать наоборот.

В пакете io существует интерфейс Closeable – который содержит единственный метод close(). Этот интерфейс имплементируется всеми основными абстрактными классами streams (Writer, Reader, InputStream, OutputStream).

Источники:

<https://wm-help.net/lib/b/book/3683783285/38>

Полезное видео, обо всех классах коротко в IO:

<https://www.youtube.com/watch?v=O0SUpA4ci-4&list=PLGbj5Xe61j2DmseE86Z1d-ZDlK-iEWe11>