**NIO (New I/O) –** новая система ввода-вывода, в которой поддерживается канальный подход к операциям ввода-вывода, ориентированный на применение буферов.

Главная проблема **IO:** один connection – один поток. Чтобы поддерживать несколько операций ввода вывода нужно иметь несколько потоков, переключение между потоками дорогостоящая операция для системы, и сам поток занимает некоторые ресурсы.

В **NIO** работа происходит через **каналы**, которые управляются **Селектором.** Нет необходимости в отдельном потоке на каждый канал.

IO работает по принципу **stream-based** (чтение и запись происходит в стрим). Ввод-вывод в нем блокирующий, пока данные не будут считаны или записаны поток простаивает.

NIO работает по принципу **buffer-based** (чтение и запись происходит в буфер). Мы можем передвигаться по буферу вперед-назад. Неблокирующий ввод-вывод. Поток выполнения может запросить запись в канал, но не дожидаться пока они будут полностью записаны.

Позволяет получать только то, что доступно на данный момент. Данные читаются в буфер, и могут быть записаны в него частично. Мы можем прочитать из буфера то, что доступно на данный момент.

В **Буфере** хранятся данные, а **канал** предоставляет открытое соединение с устройством ввода-вывода, например файлом или сокетом.

Буферы определяются в пакете *java.nio* и являются подклассами класса **Buffer.** Основные поля, содержащиеся в нем:

* **Текущая позиция (position) –** индекс в буфере, с которого следующий раз начнется операция чтения или записи данных. Она перемещается после выполнения большинства операций.
* **Предел (limit) –** определяет значение на 1 большее чем позиция последней доступной ячейки в буфере. В режиме записи равен capacity. А в режиме чтения limit указывает фактическое количество байт в нем. То есть то, сколько можем прочитать
* **Емкость (capacity) –** количество элементов, которые можно хранить в буфере. (Зачастую равен пределу)
* **Метка (mark) –** можем пометить байт меткой, и потом откатываться к ней c помощью reset().

**Типы буферов:**

ByteBuffer

IntBuffer

CharBuffer

LongBuffer  
DoubleBuffer  
MappedByteBuffer – производный от класса ByteBuffer и служит для отображения файла в буфер.

FloatBuffer  
ShortBuffer

Класс ByteBuffer имеет **методы** (другие буферы имеют аналогичные):

* **get()** – получить байт на текущей позиции
* **get(byte vals[])** – копирует буфер в указанный массив.
* **Get(int index)** – Возвращает байт из буфера по указанному индексу
* **put(byte b)** – копирует заданный байт b на текущую позицию в вызывающем буфере.
* **put(byte vals[])** – копирует все элементы из массива byte в вызывающий буфер начиная с текущей позиции. Если элементы не поместятся в буфер, выбрасывает BufferOverflowException
* **put(ByteBuffer bb)** – копирует элементы из буфера bb в текущий буфер. Выбрасывает BufferOverflowException
* **allocate()** – вручную выделить оперативную память под буфер
* **wrap()** – организовать массив в пределах буфера
* **slice()** – создать определенный срез буфера
* **clear()** – очистить буфер

**КАНАЛЫ**

Определены в пакете **java.nio.channels**, представляют открытое соединение с источником или адресатом ввода-вывода. Каналы реализуют интерфейс **Channel.** Channel расширяет AutoCloseable и его можно использовать в try с ресурсами.

Получить канал можно вызовом метода **getChannel()** для объекта, который его поддерживает. Следующие объекты поддерживают этот метод



Типы channel:

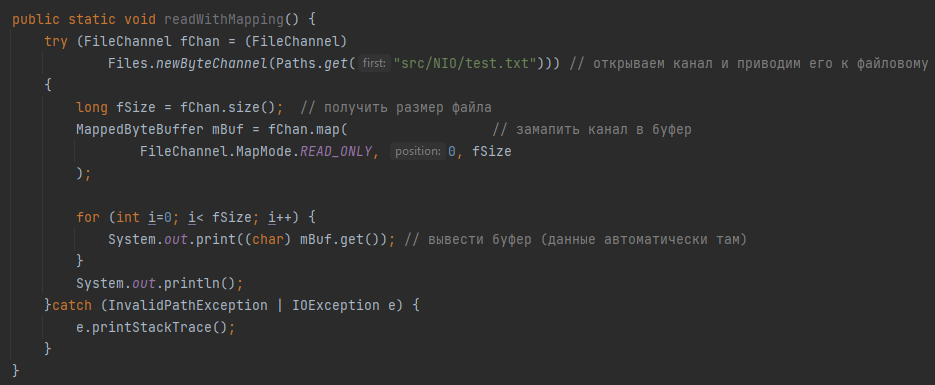
* **FileChannel** – для работы с файлами
* **SocketChannel** – для работы с сокетами
* **ServerSocketChannel** – сокеты работают через TCP
* **DatagramChannel –** работает через UDP

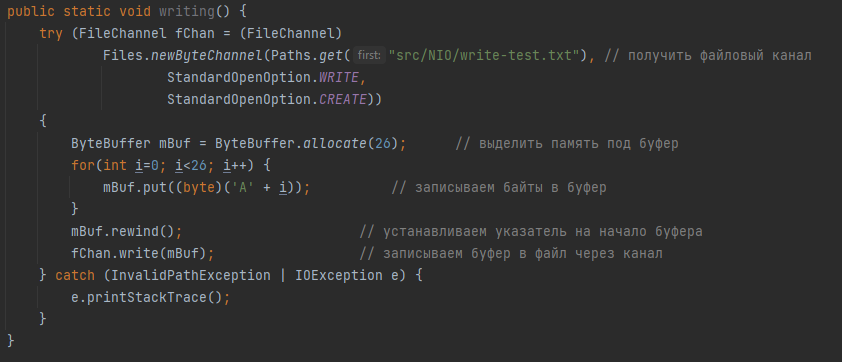
**Методы FileChannel**

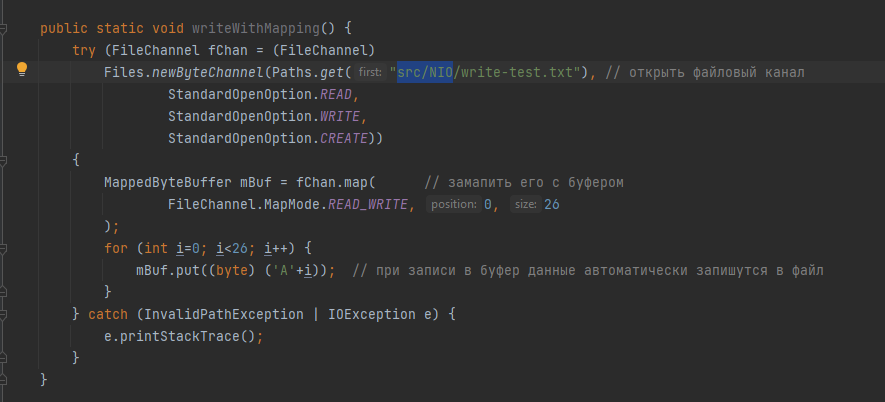
* **read(ByteBuffer bb)** – считывает байты из вызывающего канала в указанный буфер, до тех пор, пока не заполнится буфер либо не исчерпаются вводимые данные. Возвращает число прочитанных байт.
* **read(ByteBuffer bb, long begin)** – аналогичен предыдущему, но считывает с указанной позиции. Если позиция окажется за пределами файла вернется -1.
* **write(ByteBuffer bb)** – записывает содержимое буфера в канал начиная с текущей позиции. Возвращает число записанных байт.
* **Write(ByteBuffer bb, long begin)** – записывает содержимое буфера начиная с указанной позиции.
* Методы для получения текущей позиции, размера канала и блокировки.

**Примеры чтения и записи с помощью каналов и буферов**









**PATH**

Интерфейс **Path** инкапсулирует путь к файлу. Служит связующим звеном для большинства файлов средств в NIO. Он описывает расположение файла в структуре каталогов.

Можно получить из объекта типа File вызвав метод **toPath().**

**Методы:**

* **endsWith(String путь) –** Возвращает логическое значение true если Path оканчивается заданным путем, иначе false.
* **getFileName() –** возвращает имя файла в виде объекта Path.
* **getName(int index) –** возвращает имя элемента пути по указанному индексу (название папки). Крайний слева элемент имеет нулевой индекс и находится ближе всего к корневому каталогу.
* **getNameCount()** – возвращает количество элементов в вызывающем объекте типа Path(сколько вложенных папок)
* **getParent()** – возвращает объект типа Path, который содержит весь путь, кроме имени файла.
* **getRoot()** – возвращает корневой каталог
* **isAbsolute()** – возвращает true если Path абсолютный, иначе false
* **resolve()** – вернуть абсолютный Path
* **startsWith(String путь)** – возвращает true если Path начинается с указанного пути
* **toAbsolutePath()** – возвращает вызывающий объект типа Path в виде абсолютного пути.
* **toString()** – строковое представление вызывающего объекта Path

**FILES**

**Files** – утилитарный класс, предоставляющий статические методы для работы с файлами. Путь к файлу задается объектом Path.

**Методы (статические):**

* **copy(Path источник, Path адресат, CopyOption … способ) –** копирует файл из источника по указанному адресату заданным способом.
* **createDirectory(Path путь, FileAttribute<?> … атрибуты) –** создает каталог по указанному пути и с указанными атрибутами.
* **createFile(Path путь, FileAttribute<?> … атрибуты) -** создает файл по указанному пути и с указанными атрибутами.
* **delete(Path путь) –** удаляет файл по указанному пути.
* **exists(Path путь, LinkOptions … параметры) –** существует ли файл по указанному пути.
* **isDirectory(Path путь, LinkOptions … параметры) –** является ли файл(то что указано в path) каталогом
* **isExecutable(Path путь) –** является ли файл исполняемым
* **isHidden(Path путь) –** является ли файл скрытым
* **isReadable(Path путь) –** доступен ли файл для чтения
* **isWritable(Path путь) –** доступен ли файл для записи
* **move(Path источник, Path адресат, CopyOption … способ) –** перемещает файл из источника по указанному адресату
* **newByteChannel(Path путь, OpenOption … способ) –** открывает файл по указанному пути заданным способом. Возвращает байтовый канал.
* **newDirectoryStream(Path путь) –** открывает каталог по указанному пути. Возвращает DirectoryStream. Может быть использован чтобы перебрать файлы в каталоге через foreach.
* **notExists(Path путь, Link Option … параметры) –** возвращает true если файл по указанному пути не существует, иначе false.
* **readAttributes(Path путь, Class<A> тип\_атрибута, LinkOption … параметры) –** получить атрибуты, связанные с файлом.
* **Size(Path путь) –** получить размер файла
* **newInputStream(Path путь, OpenOption … способ) –** открыть файл и привязать к нему поток ввода.
* **newOutputStream(Path путь, OpenOption … способ) –** открыть файл и привязать к нему поток вывода.
* **walkFileTree(Path корень, FileVisitor<? Extends Path> fv) –** метод для обхода дерева каталогов.

**PATHS**

**Paths –** утилитарный класс для работы с объектами Path.  
**Методы (статические):**

* **get(String имя\_пути, String … части) –** получить объект типа Path из строки. Можно добавлять по частям.
* **get(URI uri) –** получить объект Path из ури.

Создание объекта Path не приводит к открытию или созданию файла. Создается лишь объект, инкапсулирующий путь к файлу.

**OPENOPTION**

**OpenOption –** интерфейс, описывающий способ открытия фала. Реализуется классом **StandardOpenOption,** где определяется перечисление со следующими значениями:

* **APPEND –** присоединить вводимые данные в конец файла
* **CREATE –** создать файл, если он не существует
* **CREATE\_NEW –** создать файл только в том случае, если он еще не существует
* **DELTE\_ON\_CLOSE –** удалить файл, когда он закрывается
* **DSYNC –** немедленно записать вносимые изменения в физический файл (не использовать буфер)
* **READ –** открыть файл для операций ввода
* **SPARSE –** указать файловой системе, что файл разрежен, а следовательно, не может быть полностью заполнен данными.
* **SYNC –** Немедленно записать вносимые изменения в файл или его метаданные в физический файл.
* **TRANCATE\_EXISTING –** укоротить до нуля длину уже существующего файла, открываемого для вывода
* **WRITE –** открыть файл для операций вывода

**SELECTORS**

**Selector –** объект, который может проверять один или несколько экземпляров Channel и определять, какие каналы готовы, например, для чтения записи. Таким образом один поток выполнения может управлять несколькими каналами.

Селекторы могут работать только с неблокирующими каналами (такие как сокетные, с FileChannel не могут).

**Методы:**

* **Selector.open() –** создать селектор
* **selector.select() –** метод блокируется до тех пор, пока по крайней мере один канал не будет готов к работе. Возвращает количество ключей, каналы которых готовы к операции.
* **selector.selectedKeys() –** получить набор ключей, каналы которых готовы к операции. После этого мы перебираем этот набор, получаем канал и выполняем интересующие нас операции.

Чтобы **связать** канал с селектором используется метод:

**SelectionKey key = Channel.register(selector, SelectionKey.OP\_READ)**

Объект **SelectionKey** связывает некоторое событие в канале и селектор. Имеет следующие значения:

* **OP\_ACCEPT –** когда сервер принимает подключение от клиента
* **OP\_CONNECT –** при попытке клиента подключится к серверу
* **OP\_READ –** когда сервер готов к чтению из канала
* **OP\_WRITE –** когда сервер будет готов к записи в канал

Когда дейстиве ввода-вывода происходит на любом из каналов, селектор уведомляет нас об этом.

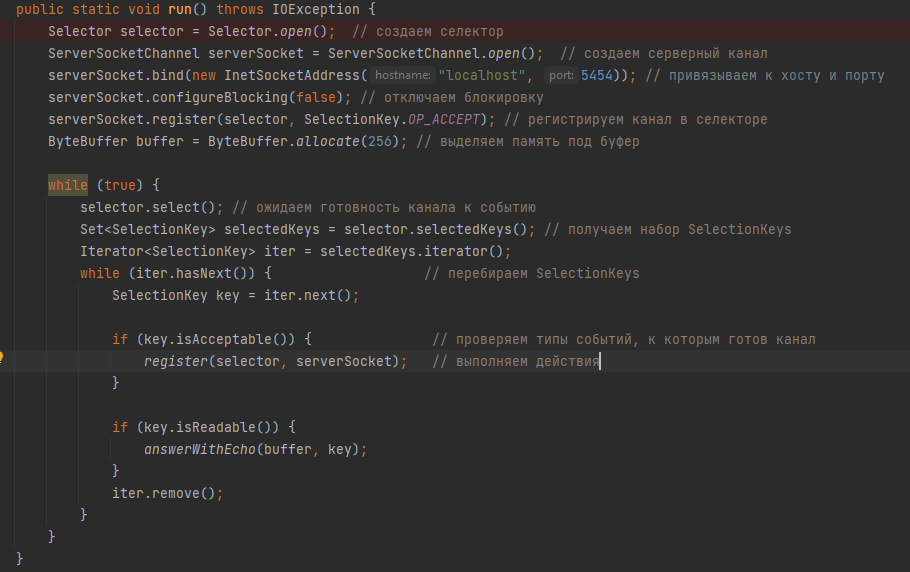
Когда мы регистрируем канал с помощью селектора, мы получаем объект SelectionKey. Этот объект содержит данные, представляющие регистрацию канала.

**Методы SelectionKey:**

* **key.interestOps() -**  посмотреть какие события отслеживаются. Возвращает целое число. Чтобы определить отслеживаемое событие используем

boolean isInterestedInConnect = interestSet & SelectionKey.*OP\_CONNECT*;

* **key.readyOps() –** посмотреть к каким событиям готов канал. Определяем готовность также через амперсанд. Для этой же цели есть более удобные методы, указанные далее.
* **key.isAcceptable()**  - сервер готов принять подключение клиента
* **isConnectable()** – клиент готов подключится к серверу
* **isReadable() –** канал готов для чтения
* **isWritable() –** канал готов для записи
* **key.channnel() –** получить канал
* **key.selector() –** получить селектор
* **key.attach(Object) –** прикрепить объект к ключу выбора (Это может быть какой-нибудь идентификатор или еще что-то)



**FILEVISITOR**

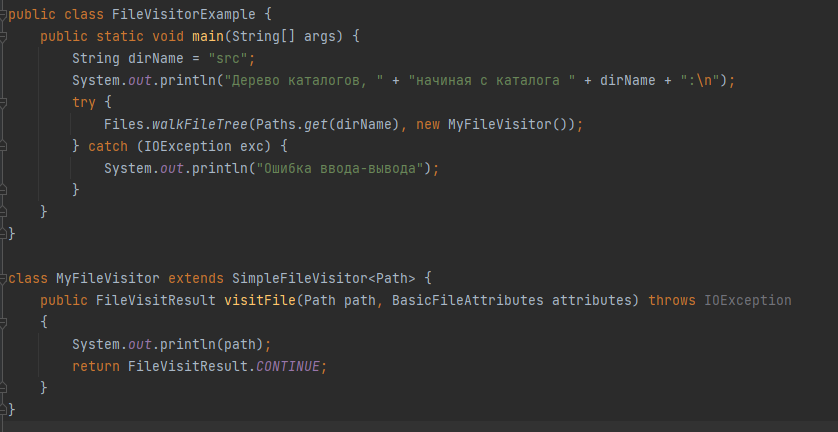
**FileVisitor –** интерфейс определяющий способ обхода дерева каталогов и позволяющий обращаться к сведениям о каталоге. Реализуется классом **SimpleFileVisitor**.

**Методы** определенные в FileVisitor:

* **postVisitDirectory(T каталог, IOException исключение) –** вызывается после посещения каталога.
* **preVisitDirectory(T каталог, BasicFileAttributes атрибуты) –** вызывается перед посещением каталога. Чтобы исследовать каталог, следует возвратить **FileVisitResult.CONTINUE**
* **visitFile(T файл, BasicFileAttributes) –** вызывается при посещении файла. Возвращает полученный резултат.
* **visitFileFailed(T файл, IOException исключение)** – вызывается при неудачной попытке посетить файл.

Каждый метод возвращает значение из перечисления **FileVisitResult,** содержащий следующие значения:

* **CONTINUE –** продолжить обход каталога и его подкаталогов
* **SKIP\_SIBLING –** его нужно возвращать из preVisitDirectory чтобы пропустить каталог и его содержимое.
* **SKIP\_SUBTREE –** пропустить каталог и подкаталоги
* **TERMINATE –** остановить обход каталога



**WATCHSERVICE**

**WatchService –** класс для наблюдения за состоянием файлов в каталоге.

Создать объект WatchService можно следующим образом **FileSystems.*getDefault*().newWatchService()**

Затем нужно зарегистрировать путь к файлу в WatchService

**WatchKey watchKey = path.register(watchService, StandardWatchEventKinds…)**

**StandardWatchEventKinds –** класс, предоставляющий информацию о том, какие события нужно отслеживать. Имеет значения:

* **ENTRY\_CREATE –** когда делается новая запись в каталог (Создание нового файла, переименование существующего)
* **ENTRY\_MODIFY –** все изменения файлов вызывают это событие. На некоторых платформах даже изменение атрибутов файлов вызовет его.
* **ENTRY\_DELETE –** срабатывает при удалении, перемещении или переименовании записи в каталоге просмотра.
* **OVERFLOW –** срабатывает для обозначения потерянных или отброшенных событий.

Получить события с помощью класса watchService можно несколькими способами:

* **WatchKey key = watchService.poll([long timeout])** – возвращает следующий ключ в очереди, любое событие которого произошло, либо null, если событий не произошло. Таймаут указывает сколько времени ожидать событие.
* **WatchKey key = watchService**.**take() –** Этот метод блокирует поток до тех пор, пока событие не произойдет.

Работа обычно происходит с помощью бесконечного цикла, в котором мы получаем объекты WatchKey, из них с помощью **key.pollEvents()** объекты **WatchEvent.**

Из WatchEvent мы можем получить тип события, контекст (какой файл изменился)

