ДИСЦИПЛИНА **Алгоритмы и структуры данных с использованием компилируемых языков**

(полное наименование дисциплины без сокращений)

ИНСТИТУТ **Искусственного интеллекта**

КАФЕДРА **Технологий Искусственного Интеллекта**

(полное наименование кафедры)

ВИД УЧЕБНОГО МАТЕРИАЛА **Материалы для практических/семинарских занятий**

(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ **Куликов Александр Анатольевич**

(фамилия, имя, отчество)

СЕМЕСТР **1, 2023-2024**

(указать семестр обучения, учебный год)

Практическая работа №2

по дисциплине «Алгоритмы и структуры данных с использованием компилируемых языков» дляstудентов 1 курса магистратуры. Форма обучения: Очная.

Тема: Неблокирующий ввод-вывод в JAVA NIO

Теоретическое введение

Java NIO — это библиотека, представленная в Java 1.4. Java NIO с момента своего запуска предоставила альтернативный способ обработки операций ввода-вывода и сетевых транзакций. Он считается альтернативой библиотекам Java Networking и Java IO. Java NIO была разработана с целью сделать транзакции для ввода и вывода асинхронными и неблокирующими.

IO (Input & Output) API — это Java API, которое облегчает разработчикам работу с потоками. Скажем, мы получили какие-то данные (например, фамилия, имя и отчество) и нам нужно записать их в файл — в этот момент и приходит время использовать **java.io**.

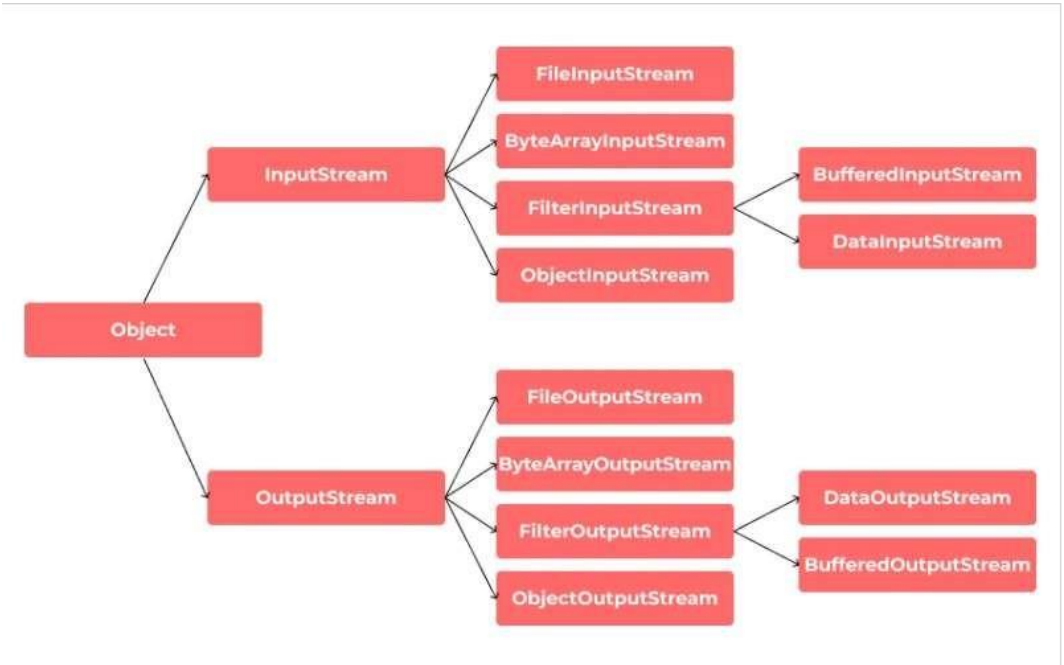


Рисунок 1 – Структура библиотеки java.io

У **Java IO** есть свои недостатки:

1. Блокирующий доступ для ввода/вывода данных. Проблема состоит в том, что когда разработчик пытается прочитать файл или записать что-то в него, используя **Java IO**, он блокирует файл и доступ к нему до момента окончания выполнения своей задачи.
2. Отсутствует поддержка виртуальных файловых систем.
3. Нет поддержки ссылок.
4. Очень большое количество checked исключений.

Работа с файлами всегда несет за собой работу с исключениями: например, попытка создать новый файл, который уже существует, вызовет **IOException**, Листинг 1. В данном случае работа приложения должна быть продолжена и пользователь должен получить уведомление о том, по какой причине файл не может быть создан.

Листинг 1 – IOException

```
try {
	File.createTempFile("prefix", "");
} catch (IOException e) {
	// Handle IOException
}
/**
 * Creates an empty file in the default temporary-file directory
 * any exceptions will be ignored. This is typically used in finally blocks.
 * @param prefix
 * @param suffix
 * @throws IOException - If a file could not be created
 */
public static File createTempFile(String prefix, String suffix)
throws IOException {
...
}
```

Метод createTempFile выбрасывает **IOException**, когда файл не может быть создан. Это исключение должно быть обработано соответственно. Если попытаться вызвать этот метод вне блока try-catch, то компилятор выдаст ошибку и предложит нам два варианта исправления: окружить метод блоком try-catch или сделать так, чтобы метод, внутри которого вызывается File.createTempFile, выбрасывал исключение **IOException** (чтобы передать его на верхний уровень для обработки).

Работа с **Java IO**:

Класс InputStream

Листинг 2 – Класс InputStream

```
try(FileInputStream fin = new FileInputStream("C:/javarush/file.txt")){
    System.out.printf("File size: %d bytes \n", fin.available());
    int i=-1;
    while((i=fin.read())!=-1){
        System.out.print((char)i);
    }
} catch(IOException ex) {
    System.out.println(ex.getMessage());
}
```

Класс FileInputStream предназначен для считывания данных из файла. Он является наследником класса InputStream и поэтому реализует все его методы. Если файл не может быть открыт, то генерируется исключение **FileNotFoundException**.

Класс OutputStream

Листинг 3 – Класс OutputStream

```
String text = "Hello world!"; // строка для записи
try(FileOutputStream fos = new FileOutputStream("C:/javarush/file.txt")){
    // переводим нашу строку в байты
    byte[] buffer = text.getBytes();
    fos.write(buffer, 0, buffer.length);
    System.out.println("The file has been written");
} catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream.

Классы Reader и Writer

Класс FileReader позволяет нам читать символьные данные из потоков, а класс FileWriter используется для записи потоков символов. Реализация записи и чтения из файла приведена ниже.

Листнг 4 – Реализация записи и чтения из файла

```
String fileName = "c:/javarush/Example.txt";
// Создание объекта FileWriter
try (FileWriter writer = new FileWriter(fileName)) {
   // Запись содержимого в файл
   writer.write("Это простой пример,\n в котором мы осуществляем\n с помощью языка Java\n запись в файл\n и чтение из файла\n");
   writer.flush();
} catch (IOException e) {
   e.printStackTrace();
}
// Создание объекта FileReader
try (FileReader fr = new FileReader(fileName)) {
   char[] a = new char[200];// Количество символов, которое будем считывать
   fr.read(a);   // Чтение содержимого в массив
   for (char c : a) {
      System.out.print(c); // Вывод символов один за другими
   }
} catch (IOException e) {
   e.printStackTrace();
}
```

Теперь немного поговорим о том, что нового появилось в **Java NIO2**.

Path

Path представляет из себя путь в файловой системе. Он содержит имя файла и список каталогов, определяющих путь к нему.

```
Path relative = Paths.get("Main.java");
System.out.println("Файл: " + relative);
//получение файловой системы
System.out.println(relative.getFileSystem());
```

Paths — это совсем простой класс с единственным статическим методом get(). Его создали исключительно для того, чтобы из переданной строки или URI получить объект типа Path.

```
Path path = Paths.get("c:\\data\\file.txt");
```

Files

Files — это утилитный класс, с помощью которого можно напрямую получать размер файла, копировать их, и не только.

```
Path path = Paths.get("files/file.txt");
boolean pathExists = Files.exists(path);
```

FileSystem

FileSystem предоставляет интерфейс к файловой системе. Файловая система работает как фабрика для создания различных объектов (Path, PathMatcher, Files). Этот объект помогает получить доступ к файлам и другим объектам в файловой системе.

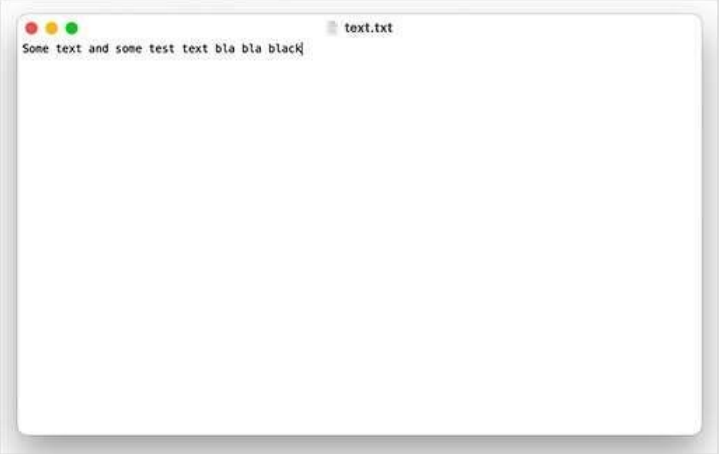
```
try {
    FileSystem filesystem = FileSystems.getDefault();
    for (Path rootdir : filesystem.getRootDirectories()) {
       System.out.println(rootdir.toString());
    }
 } catch (Exception e) {
    e.printStackTrace();
 }
```

Тест производительности

Для этого теста возьмем два файла. Первый — маленький файл с текстом, а второй — большой видеоролик.

Создаем файл и добавляем в него немного слов и символов:

```
% touch text.txt
```



Наш файл по итогу занимает в памяти 42 байта:

```
antonkupreychik@MacBook-Pro testFolder % touch text.txt
antonkupreychik@MacBook-Pro testFolder % ls -l
total 0
-rw-r--r--  1 antonkupreychik  staff  0 Jan 31 13:54 text.txt
antonkupreychik@MacBook-Pro testFolder % ls -l
total 8
-rw-r--r--@ 1 antonkupreychik  staff  42 Jan 31 13:55 text.txt
antonkupreychik@MacBook-Pro testFolder %
```

Теперь напишем код, который будет копировать наш файл из одной папки в другую. Проверим его работу на маленьком и большом файлах, и тем самым сравним скорость работы **IO**, **NIO** и **NIO2**.
Код для копирования, написанный на **Java IO**:

```
    public static void main(String[] args) {
        long currentMills = System.currentTimeMillis();
        long startMills = currentMills;
        File src = new File("/Users/IdeaProjects/testFolder/text.txt");
        File dst = new File("/Users/IdeaProjects/testFolder/text1.txt");
        copyFileByIO(src, dst);
        currentMills = System.currentTimeMillis();
        System.out.println("Время выполнения в миллисекундах: " + (current-
Mills - startMills));
    }

    public static void copyFileByIO(File src, File dst){
        try(InputStream inputStream = new FileInputStream(src);
            OutputStream outputStream = new FileOutputStream(dst)){

            byte[] buffer = new byte[1024];
            int length;
            // Читаем данные в байтовый массив, а затем выводим в OutStream
            while((length = inputStream.read(buffer)) > 0){
                outputStream.write(buffer, 0, length);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

И код для **Java NIO**:

```
    public static void main(String[] args) {
        long currentMills = System.currentTimeMillis();
        long startMills = currentMills;

        File src = new File("/Users/IdeaProjects/testFolder/text.txt");
        File dst = new File("/Users/IdeaProjects/testFolder/text2.txt");
        // копия nio
        copyFileByChannel(src, dst);
        currentMills = System.currentTimeMillis();
        System.out.println("Время выполнения в миллисекундах: " + (current-
Mills - startMills));
    }
```

```
    public static void copyFileByChannel(File src, File dst){
        // 1. Получаем FileChannel исходного файла и целевого файла
        try(FileChannel srcFileChannel  = new FileInputStream(src).getChannel();
            FileChannel dstFileChannel = new FileOutputStream(dst).getChannel()){
            // 2. Размер текущего FileChannel
            long count = srcFileChannel.size();
            while(count > 0){
/**=========================================================
=======
             * 3. Записать байты из FileChannel исходного файла в целевой FileChannel
             *  1. srcFileChannel.position (): начальная позиция в исходном файле не может быть отрицательной
             * 2. count: максимальное количество переданных байтов, не может быть отрицательным
             * 3. dstFileChannel: целевой файл
*=========================================================
=======*/
            long transferred = srcFileChannel.transferTo(srcFileChannel.position(),
                    count, dstFileChannel);
            // 4. После завершения переноса измените положение исходного файла на новое место
            srcFileChannel.position(srcFileChannel.position() + transferred);
            // 5. Рассчитаем, сколько байтов осталось перенести
            count -= transferred;
        }

      } catch (FileNotFoundException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
```

Код для **Java NIO2**:

```
    public static void main(String[] args) {
      long currentMills = System.currentTimeMillis();
      long startMills = currentMills;

      Path sourceDirectory = Paths.get("/Users/IdeaProjects/testFolder/test.txt");
      Path targetDirectory = Paths.get("/Users/IdeaProjects/testFolder/test3.txt");
```

```
        Files.copy(sourceDirectory, targetDirectory);

        currentMills = System.currentTimeMillis();
        System.out.println("Время выполнения в миллисекундах: " + (currentMills -
    startMills));
    }
```

Начнем с маленького файла.

Время выполнения с помощью **Java IO** в среднем было 1 миллисекунду. Запуская тест несколько раз, получаем результат от 0 до 2 миллисекунд.

Время выполнения в миллисекундах: 1

Время выполнения с помощью **Java NIO** гораздо больше. Среднее время — 11 миллисекунд. Результаты были от 9 до 16. Это связанно с тем, что **Java IO** работает не так, как наша операционная система. **IO** перемещает и обрабатывает файлы один за другим, в то время как операционная система отправляет данные в одном большом виде. А **NIO** показал плохие результаты из-за того, что он ориентирован на буфер, а не на поток, как **IO**.

Время выполнения в миллисекундах: 12

И так же тест для **Java NIO2**. **NIO2** имеет улучшенное управление с файлами по сравнению с **Java NIO**. Из-за этого результаты обновленной библиотеки так отличаются:

Время выполнения в миллисекундах: 3

Протестируем большой файл, видео на 521 МБ. Задача будет точно такой же: скопировать в другую папку.

Результаты с **Java IO**:

Время выполнения в миллисекундах: 1866

А вот результат **Java NIO**:

Время выполнения в миллисекундах: 205

Java NIO справился с файлом в 9 раз быстрее при первом тесте. Повторные тесты показывали примерно такие же результаты.

тест на **Java NIO2**:

Время выполнения в миллисекундах: 360

Почему же такой результат? Просто потому что нет особого смысла сравнивать производительность между ними, так как они служат разным целям. **NIO** представляет собой более абстрактный низкоуровневый ввод-вывод данных, а **NIO2** ориентирован на управление файлами.

Итоги

Java NIO существенно повышает эффективность работы с файлами за счет использования внутри блоков. Еще один плюс состоит в том, что библиотека **NIO** разбита на две части: одна для работы с файлами, вторая — для работы в сети.

Новый API, который используется в **Java NIO2** для работы с файлами, предлагает множество полезных функций:

- гораздо более полезную адресацию файловой системы с помощью Path,
- значительно улучшенную работу с ZIP-файлами с использованием пользовательского поставщика файловой системы,
- доступ к специальным атрибутам файла,
- множество удобных методов, например, чтение всего файла с помощью одной команды, копирование файла с помощью одной команда и т. д.

Все это связано с файлом и файловой системой, и все довольно высокого уровня. В современных реалиях **Java NIO** занимает около 80-90% работы с файлами, хотя доля **Java IO** тоже еще существенна.

Основные отличия между Java IO и Java NIO

IO	NIO
Потокоориентированный	Буфер-ориентированный
Блокирующий (синхронный) ввод/вывод	Неблокирующий (асинхронный) ввод/вывод
	Селекторы

Потокоориентированный и буфер-ориентированный ввод/вывод

Основное отличие между двумя подходами к организации ввода/вывода в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным. Разберем подробней.

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если надо произвести подобные манипуляции, придется сначала кэшировать данные в буфере.

Блокирующий и неблокирующий ввод/вывод

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.
Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или

вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.

Каналы – это логические (не физические) порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе, данные из канала помещаются в предоставленный вами буфер. Каналы напоминают трубопроводы, по которым эффективно транспортируются данные между буферами байтов и сущностями по ту сторону каналов. Каналы – это шлюзы, которые позволяют получить доступ к сервисам ввода/вывода операционной системы с минимальными накладными расходами, а буферы – внутренние конечные точки этих шлюзов, используемые для передачи и приема данных.

Тоже самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.

Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

Обработка данных

Обработка данных при использовании Java NIO тоже отличается. При использовании Java IO данные читаются байт за байтом с InputStream или Reader. Представьте, что идет считывание строк текстовой информации:

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

Этот поток строк текста может обрабатываться следующим образом:

```
InputStream input = ... ;
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine   = reader.readLine();
```

```
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```

Состояние процесса обработки зависит от того, насколько далеко продвинулось выполнение программы. Когда первый метод readLine() возвращает результат выполнения– целая строка текста была считана. Метод является блокирующим и действие блокировки продолжается до тех пор, пока вся строка не будет считана. Данная строка содержит имя. Подобно этому, когда метод вызывается во второй раз, получим возраст.

Прогресс в выполнении программы достигается только тогда, когда доступны новые данные для чтения, и для каждого шага вы знаете что это за данные. Когда поток выполнения достигает прогресса в считывании определенной части данных, поток ввода (в большинстве случаев) уже не двигает данные назад. Данный принцип хорошо демонстрирует следующая схема:

Имплементация с использованием Java IO будет выглядеть несколько иначе:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Обратите внимание на вторую строчку кода, в которой происходит считывание байтов из канала в ByteBuffer. Когда возвращается результат выполнения данного метода, нельзя быть уверенным, что все необходимые вам данные находятся внутри буфера. Все, что известно, так это то, что буфер содержит некоторые байты. Это немного усложняет процесс обработки.

После первого вызова метода read(buffer), в буфер было считано только половину строки. Например, “Name: An”. Придется ждать пока, по крайней мере, одна полная строка текста не будет считана в буфер. Единственный вариант узнать достаточно ли данных для корректной обработки содержит буфер, это посмотреть на данные, содержащиеся внутри буфера. В результате придется по нескольку раз проверять данные в буфере, пока они не станут доступными для корректной обработки. Это неэффективно и может негативно сказаться на дизайне программы. Например:

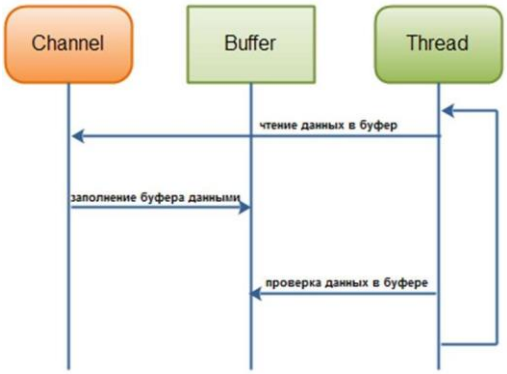
```
ByteBuffer buffer = ByteBuffer.allocate(48);
  int bytesRead = inChannel.read(buffer);
        while(! bufferFull(bytesRead) ) {
           bytesRead = inChannel.read(buffer);
        }
```

Метод bufferFull() должен следить за тем, сколько данных считано в буфер и возвращать true или false, в зависимости от того, заполнен буфер или нет. Другими словами, если буфер готов к обработке, то он считается заполненным.

Также метод bufferFull() должен оставлять буфер в неизмененном состоянии, поскольку в противном случае следующая порция считанных данных может быть записана в неправильное место.

Если буфер заполнен, данные из него могут быть обработаны. Если он не заполнен вы все же будете иметь возможность обработать уже имеющиеся в нем данные, если это имеет смысл в вашем конкретном случае. В большинстве случаев – это бессмысленно.

Схема демонстрирует процесс определения готовности данных в буфере для корректной обработки:



Java NIO Buffer

Буфер, по сути, является средой для хранения данных, пока поток не прочитает данные из него и не запросит новые данные. Основным шагом к чтению данных из источника ввода является чтение данных в буфер.

Буфер — это фиксированная часть памяти, используемая для хранения этих данных перед их чтением в канал. Буфер обеспечивает предварительную загрузку данных определенного размера для ускорения чтения файлов, входных данных и потоков данных. Размер буфера настраивается в блоках от 2 до степени n.

Данные считываются в буфер для последующей обработки. Разработчик может двигаться по буферу вперед и назад, что дает немного больше гибкости при обработке данных. В то же время нужно проверять, содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер не уничтожить еще не обработанные данные, находящиеся там.

```
ByteBuffer buf = ByteBuffer.allocate (2048);
int bytesRead = channel.read(buf);
buf.flip(); // меняем режим на чтение
while (buf.hasRemaining()) {
  byte data = buf.get(); // есть методы для примитивов
}
buf.clear(); // очистили и можно переиспользовать
```

В коде создается буфер из 2048 байтов. Размер буфера является обязательным для указания. 3-я строка выделяет область памяти 2048 байтов для буфера **buf.** Это гарантирует, что необходимая память предварительно выделена для буфера. Процесс чтения и записи в буфер необходимо понять, прежде чем перейдем к их использованию для чтения и записи канала. Затем буфер используется для чтения данных для канала с помощью оператора `channel.read(buf)` . При выполнении этого оператора буфер теперь будет содержать до 2048 байт данных как доступные. Чтобы начать читать данные, используется простое утверждение `buf.get()` .

Буфер читает и толкает первый байт влево. Буфер является распределением памяти типа «Последний пришел — первый вышел». Следовательно, когда надо прочитать файл с помощью буфера, необходимо перевернуть его, прежде чем прочитать файл. Без переворота

данные вышли бы в обратном порядке. Чтобы перевернуть буфер, необходимо выполнить простую строку кода: `buf.flip();`

Как только данные были прочитаны в буфер, пришло время фактически получить данные, которые были прочитаны. Для чтения данных используется функция `buf.get()` . Этот вызов функции читает один байт / символ / пакет при каждом вызове в зависимости от типа буфера. После того, как прочитали доступные данные, также необходимо очистить буфер перед следующим чтением. Очистка необходима, чтобы освободить место для чтения дополнительных данных. Чтобы очистить буфер, есть два возможных способа — очистить буфер или сжать буфер.

Чтобы очистить буфер, выполните команду `buf.clear()` . Чтобы `buf.compact()` буфер, используйте команду `buf.compact()` . Обе эти команды в конечном итоге делают одно и то же. Однако `compact()` очищает только те данные, которые были прочитаны с использованием вызова функции `buf.get()` . Таким образом, он используется, когда нам нужно продолжать оптимизировать объем памяти, используемой буфером.

Пометить и сбросить буфер

Во время процесса чтения часто возникают ситуации, когда необходимо повторно прочитать данные с определенной позиции. В обычном сценарии, когда данные получаются из буфера, они считаются пропавшими. Тем не менее, можно пометить буфер по определенному индексу, чтобы можно было снова прочитать его из определенной позиции. Код демонстрирует, как это делается.

```
    buffer.mark();
    char x = buffer.get();
    char y = buffer.get();
    char z = buffer.get();
    char a = buffer.get();
    //Do something with above data
buffer.reset(); //set position back to mark
```

Основными приложениями отметки и сброса являются многократный анализ данных, многократное повторение информации, отправка команды на определенное количество раз и многое другое.

Различные типы буферов, которые могут использоваться в зависимости от типа ввода:

- **ByteBuffer:** используется для чтения потоков символов или файлов в **байтовом выражении**
- **CharBuffer:** используется для чтения символов в полном наборе ASCII
- **DoubleBuffer:** используется специально для двойных значений данных, таких как показания датчиков
- **FloatBuffer:** используется для чтения постоянных потоков данных для таких целей, как аналитика
- **LongBuffer:** используется для чтения значений типа данных long
- **IntBuffer:** используется для чтения целочисленных значений для результатов или результатов.
- **ShortBuffer:** используется для чтения коротких целочисленных значений

Каждый буфер предназначен для его конкретного использования. Буферы, обычно используемые для файлов, являются **ByteBuffer** и **CharBuffer.**

Основные свойства буфера:

Основные атрибуты	
capacity	Размер буфера, который является длиной массива.
position	Начальная позиция для работы с данными.
limit	Операционный лимит. Для операций чтения предел — это объем данных, который можно поместить в оперативный режим, а для операций записи — предел емкости или доступная для записи квота, указанная ниже.
mark	Индекс значения, до которого будет сброшен параметр position при вызове метода reset().

Во время записи в буфер позиция буфера — это позиция, в которой записывается текущий байт. Во время процесса чтения из буфера позиция буфера — это позиция, из которой читается байт. Положение буфера продолжает динамически меняться, когда мы продолжаем чтение или запись.

Во время записи в буфер предел буфера — это максимальный размер данных, которые могут быть записаны в буфер. По существу, ограничение буфера и емкость буфера являются синонимами во время записи в буфер. Однако во время чтения из буфера предел буфера — это количество доступных байтов для чтения из буфера. Следовательно, ограничение буфера продолжает уменьшаться по мере выталкивания байтов.

Емкость буфера — это максимальные данные, которые можно записать в буфер или прочитать из буфера в любой момент времени. Таким образом, размер 48, выделенный выше, также называется буферной емкостью.

Канальные передачи

Передача канала, как следует из названия, представляет собой процесс передачи данных из одного канала в другой. Передача канала может осуществляться из определенной позиции буфера канала. Однако при значении позиции, равном нулю, можно копировать или реплицировать полный источник ввода в указанное место назначения вывода. Например, установление канала между ключевым словом и текстовым редактором позволит непрерывно переносить ввод с клавиатуры в текстовый редактор. Чтобы облегчить передачу канала, Java NIO оснащен двумя функциями, а именно
— `transferFrom()` и `transferTo()` . Чтобы лучше понять эти функции, будем использовать файл **data.txt,** созданный ранее, в качестве источника ввода. Перенесем данные из этого файла в новый файл **output.txt** . Код на слайде делает то же самое, используя вызов метода TransferFrom `transferFrom()` .

ChannelTransfer.java

```
import java.io.RandomAccessFile;
import java.nio.channels.FileChannel;
public class ChannelTransfer {
    public static void main(String[] args) {
        try {
            RandomAccessFile copyFrom =
RandomAccessFile("src/data.txt", "rw");
            FileChannel fromChannel = copyFrom.getChannel(
            RandomAccessFile copyTo =
RandomAccessFile("src/output.txt", "rw");
            FileChannel toChannel = copyTo.getChannel();
```

```
            long count = fromChannel.size();
            toChannel.transferFrom(fromChannel, 0, count);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Как видно из приведенного кода, канал `fromChannel` используется для чтения данных из data.txt. `toChannel` используется для получения данных из `fromChannel` начиная с позиции 0. Важно отметить, что весь файл копируется с использованием `FileChannel` . Однако в некоторых реализациях `SocketChannel` это может быть не так. В таком случае копируются только те данные, которые доступны для чтения в буфере во время передачи. Такое поведение обусловлено динамической природой реализаций `SocketChannel` .

Реализация `transferTo` довольно похожа. Единственное изменение, которое потребуется, — это то, что вызов метода будет выполняться с использованием исходного объекта, а объект канала назначения будет аргументом в вызове метода.

Теперь поговорим о **Java NIO**:

Java NIO Channel

Каналы являются основной средой для неблокирующего ввода-вывода. Каналы аналогичны потокам, доступным для блокировки ввода-вывода. Эти каналы поддерживают данные по сетям, а также файлы ввода-вывода данных. Каналы читают данные из буферов по мере необходимости. Буферы хранят данные до тех пор, пока они не будут прочитаны.

Каналы имеют несколько реализаций в зависимости от данных, которые нужно прочитать.

В отличие от потоков, которые используются в **Java IO**, NIO Channel является двусторонним, то есть может и считывать, и записывать. Канал **Java NIO** поддерживает асинхронный поток данных как в режиме блокировки, так и в режиме без блокировки.

```
RandomAccessFile aFile = new RandomAccessFile("C:/javarush/file.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

```
ByteBuffer buf = ByteBuffer.allocate(100);
int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {
  System.out.println("Read " + bytesRead);
  buf.flip();
            while(buf.hasRemaining()){
               System.out.print((char) buf.get());
            }
  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

Здесь реализован FileChannel. Для чтения данных из файла используется файловый канал. Объект файлового канала может быть создан только вызовом метода getChannel() для файлового объекта, поскольку нельзя напрямую создать объект файлового канала.

Кроме FileChannel есть и другие реализации каналов:

- FileChannel — используется для чтения и записи данных из файлов и в файлы
- DatagramChannel — используется для обмена данными по сети с использованием пакетов UDP.
- SocketChannel — канал TCP для обмена данными через сокеты TCP
- ServerSocketChannel реализация, аналогичная веб-серверу, который прослушивает запросы через определенный порт TCP. Создает новый экземпляр `SocketChannel` для каждого нового соединения.

Как можно понять из названий каналов, они также охватывают сетевой IO-трафик UDP + TCP в дополнение к IO-файлу. В отличие от потоков, которые могут либо читать, либо записывать в определенный момент, один и тот же канал может беспрепятственно читать и записывать ресурсы. Каналы поддерживают асинхронное чтение и запись, что обеспечивает чтение данных без ущерба для выполнения кода. Буферы, рассмотренные выше, поддерживают эту асинхронную работу каналов.

FileChannel нельзя переключить в неблокирующий режим. Неблокирующий режим **Java NIO** позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. В это же время SelectableChannel и его реализации могут устанавливаться в неблокирующем режиме с помощью метода connect().

Java NIO по своей природе поддерживает рассеяние и сбор данных для чтения и записи данных в несколько буферов. Java NIO достаточно умен, чтобы иметь возможность управлять чтением и записью в нескольких буферах.

Разброс Java NIO используется для разбивки чтения из канала на несколько буферов. Реализация кода довольно проста. Все, что нужно сделать, это добавить массив буферов в качестве аргумента для чтения. Фрагмент кода показан на слайде:

```
ByteBuffer buffer1 = ByteBuffer.allocate(128);
ByteBuffer buffer2   = ByteBuffer.allocate(128);
ByteBuffer[] buffers = {buffer1,buffer2};
channel.read(buffers);
```

В приведенном коде создается два буфера по 128 байт каждый. Обратите внимание, что массив создается из двух буферов. Этот массив затем передается в качестве аргумента в канал для чтения. Канал считывает данные в первый буфер, пока не будет достигнута емкость буфера. Как только емкость буфера достигнута, канал автоматически переключается на следующий буфер. Таким образом, чтение канала рассеивается без какого-либо влияния на поток.

Сбор Java NIO также работает аналогичным образом. Данные, считанные в несколько буферов, также могут быть собраны и записаны в один канал.

Информация записывается в канал, начиная с первого буфера. Канал автоматически переключается на следующий буфер при достижении предела для первого. Во время этой записи переворот не происходит. Если нужно перевернуть буфер перед записью, это нужно сделать перед назначением их в массив.

Direct буфер(прямой буфер)

Java NIO поддерживает тип ByteBuffer, обычно известный как direct(прямой) буфер. Direct буфер по существу могут использоваться как любой другой ByteBuffer (и реализованы как подкласс ByteBuffer), но имеют свойство, заключающееся в том, что их базовая память **выделяется вне кучи Java** . В частности, Direct буферы обладают следующими свойствами:

- после выделения их адрес памяти фиксируется на время жизни буфера;
- поскольку их адрес фиксирован, ядро может безопасно обращаться к ним напрямую, и, следовательно, прямые буферы могут использоваться более эффективно в операциях ввода-вывода;
- в некоторых случаях доступ к ним из Java может быть более эффективным (потенциально меньше накладных расходов при поиске адреса памяти и/или других служебных операций, необходимых перед доступом к объекту Java) ;

- через Java Native Interface вы можете фактически установить адрес произвольно, если это необходимо (например, для доступа к оборудованию по определенному адресу или для самостоятельного выполнения выделения).

На практике в текущих версиях Hotspot память выделяется с помощью malloc(), хотя это может отличаться в других виртуальных машинах или в будущей версии Hotspot.

Чтобы создать прямой буфер из Java, надо вызывать:

ByteBuffer directBuf = **ByteBuffer.allocateDirect(noBytes);**

Затем возвращенный ByteBuffer можно использовать практически как любой другой байтовый буфер. Например, будут работать все различные методы get() и put , а также методы для создания представлений буфера. Одна вещь, которую *нельзя* сделать, по крайней мере, в Hotspot, — это вызвать array() — в основе прямого буфера лежит не массив Java, а просто «сырой» участок памяти. (Хотя строго, согласно Javadoc, реализации фактически могут свободно реализовывать прямой буфер с резервным массивом, если они могут найти способ сделать это...)

Нет явного метода, который можно вызвать из Java для уничтожения или освобождения Direct буфера. Когда выделяется прямой буфер, виртуальная машина эффективно регистрирует метод «очистки» в сборщике мусора, который должен быть вызван в какой-то момент, когда сам объект ByteBuffer больше не доступен, чтобы освободить базовую память, зарезервированную для этого буфера.

Обычно это «автоматическое» освобождение работает достаточно хорошо. Но важно иметь в виду, что нет *непосредственной* связи между последним обращением к прямому буферу и моментом фактического освобождения памяти.

Когда выделяется прямой буфер, как правило, это влияет на другие части процесса, которые могли бы использовать malloc() для выделения памяти, особенно на память, выделенную из любых собственных библиотек, используемых программой.

В некоторых случаях может потребоваться ограничить объем памяти, которую Java-приложение может использовать для выделения прямых буферов. Для этого надо установить для свойства sun.nio.MaxDirectMemorySize требуемый предел в байтах при запуске виртуальной машины. Попытка выделить больше этого лимита безопасно выдаст OutOfMemoryError . Таким образом, можно гарантировать, что Java-приложение

"изящно" откажет, если оно попытается выделить слишком много памяти для прямых буферов, вместо того, чтобы иметь эффект домино для другого машинного кода, который, возможно, не сможет так изящно завершиться ошибкой, если malloc() терпит неудачу.

MappedFileBuffer или MappedByteBuffer

Когда надо загрузить область файла, можно загрузить ее в определенную область памяти, к которой можно получить доступ позже.

Когда заранее известно, что нужно будет прочитать содержимое файла несколько раз, рекомендуется оптимизировать дорогостоящий процесс, например, сохранив это содержимое в памяти. Благодаря этому последующие поиски этой части файла будут идти только в основную память без необходимости загружать данные с диска, что существенно сократит задержку. Одна нужно быть осторожным при использовании MappedByteBuffer , – это когда идет работа с очень большими файлами с диска – нужно убедиться, что файл поместится в память . В противном случае можно заполнить всю память и, как следствие, столкнуться с общим исключением OutOfMemoryException. Этого можно избежать, загрузив только часть файла, например, на основе шаблонов использования.

Чтобы прочитать файл, например: есть файл под названием fileToRead.txt со следующим содержанием:

Тут написано содержание файла

Файл находится в каталоге /resource , поэтому можно загрузить его с помощью функции ***getFileURIFromResources***:

```
Path getFileURIFromResources(String fileName) throws Exception {
   ClassLoader classLoader = getClass().getClassLoader();
   return Paths.get(classLoader.getResource(fileName).getPath());
}
```

Чтобы создать MappedByteBuffer из файла, сначала нужно создать FileChannel из него. После того, как канал создан, можно вызвать метод map() на нем, передавая в режиме Map, позицию, с которой хотим читать, и параметр size , который указывает, сколько байтов необходимо:

```
CharBuffer charBuffer = null;
Path pathToRead = getFileURIFromResources("fileToRead.txt");
try (FileChannel fileChannel (FileChannel) Files.newByteChannel(
pathToRead, EnumSet.of(StandardOpenOption.READ))) {
   MappedByteBuffer mappedByteBuffer = fileChannel
    .map(FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());
   if (mappedByteBuffer != null) {
      charBuffer = Charset.forName("UTF-8").decode(mappedByteBuffer);
   }}
```

Как только сопоставили файл с буфером памяти, можно прочитать данные из него в буфер Char. **Содержимое файла, которое читаем при вызове метода decode(),передающего MappedByteBuffer, чтение происходит из памяти, а не с диска.** Поэтому это чтение будет очень быстрым.
Содержимое, которое читаем из файла, является фактическим содержимым fileToRead.txt файл:

```
assertNotNull(charBuffer);
assertEquals(
  charBuffer.toString(), " Тут написано содержание файла");
```

Каждое последующее чтение из MappedByteBuffer будет очень быстрым, поскольку содержимое файла отображается в памяти, и чтение выполняется без необходимости поиска данных с диска.

Запись

Допустим, надо записать какой-то контент в файл fileToWriteTo.txt с помощью MappedByteBuffer API. Для этого нужно открыть файловый канал и вызвать на нем метод map () , передав его в файловый канал.MapMode.READ_WRITE.
Затем можно сохранить содержимое буфера Char в файл с помощью метода put() из MappedByteBuffer:

```
CharBuffer charBuffer = CharBuffer.wrap("Запись в файл");
Path pathToWrite = getFileURIFromResources("fileToWriteTo.txt");
try (FileChannel fileChannel = (FileChannel) Files
  .newByteChannel(pathToWrite, EnumSet.of(
    StandardOpenOption.READ,
    StandardOpenOption.WRITE,
    StandardOpenOption.TRUNCATE_EXISTING))) {
    MappedByteBuffer mappedByteBuffer = fileChannel
      .map(FileChannel.MapMode.READ_WRITE, 0, charBuffer.length());
    if (mappedByteBuffer != null) {
       mappedByteBuffer.put(
         Charset.forName("utf-8").encode(charBuffer));
    }
}
```

Фактическое содержимое CharBuffer было записано в файл, прочитав его содержимое:

```
List fileContent = Files.readAllLines(pathToWrite);
assertEquals(fileContent.get(0), " Запись в файл ");
```

Итог:

Это очень эффективный способ чтения содержимого файла несколько раз, так как файл отображается в память, и последующие чтения не нужно каждый раз переносить на диск.

Блокировка файла. Java NIO – FileLock

Java NIO поддерживает параллелизм и многопоточность, что позволяет ему работать с несколькими потоками, работающими с несколькими файлами одновременно. Но в некоторых случаях требуется, чтобы файл не получал общий доступ ни от одного потока и не был доступен.

Для такого требования NIO снова предоставляет API, известный как FileLock, который используется для обеспечения блокировки всего файла или его части, чтобы файл или его часть не были доступны для общего доступа.

Чтобы применить такую блокировку, надо использовать FileChannel или AsynchronousFileChannel, который предоставляет для этой цели два метода **lock()** и **tryLock().** Предоставляемая блокировка может быть двух типов:

- **Эксклюзивная блокировка** – эксклюзивная блокировка не позволяет другим программам получать перекрывающуюся блокировку любого типа.
- **Общая блокировка** . Общая блокировка не позволяет другим одновременно работающим программам получать перекрывающуюся эксклюзивную блокировку, но позволяет им получать перекрывающиеся общие блокировки.

Методы, используемые для получения блокировки над файлом:

- **lock()** – этот метод FileChannel или AsynchronousFileChannel получает эксклюзивную блокировку для файла, связанного с данным каналом. Типом возврата этого метода является FileLock, который далее используется для мониторинга полученной блокировки.
- **lock(long position, long size, boolean shared)** – этот метод снова является перегруженным методом блокировки и используется для блокировки определенной части файла.
- **tryLock()** – этот метод возвращает FileLock или null, если блокировка не может быть получена, и он пытается получить явно исключительную блокировку для файла этого канала.
- **tryLock(long position, long size, boolean shared)** – этот метод пытается получить блокировку в заданной области файла этого канала, которая может быть эксклюзивной или общего типа.

Методы класса FileLock

- **acquBy ()** – этот метод возвращает канал, для которого была получена блокировка файла.
- **position ()** – этот метод возвращает позицию в файле первого байта заблокированной области. Блокированная область не должна содержаться внутри или даже перекрывать фактический базовый файл, поэтому значение, возвращаемое этим методом, может превышать текущий размер.
- **size ()** – этот метод возвращает размер заблокированной области в байтах. Блокированная область не должна содержаться внутри или даже перекрывать фактический базовый файл, поэтому значение, возвращаемое этим методом, может превышать текущий размер файла.
- **isShared ()** – Этот метод используется для определения того, является ли блокировка общей или нет.
- **overlaps (длинная позиция, длинный размер)** – этот метод сообщает, перекрывает ли эта блокировка заданный диапазон блокировки.
- **isValid ()** – этот метод сообщает, действительна ли полученная блокировка. Объект блокировки остается действительным до тех пор, пока он не будет освобожден или соответствующий канал файла не будет закрыт, в зависимости от того, что произойдет раньше.
- **release ()** – Освобождает полученную блокировку. Если объект блокировки действителен, то вызов этого метода освобождает блокировку и делает объект недействительным. Если этот объект блокировки недействителен, то вызов этого метода не имеет никакого эффекта.
- **close ()** – этот метод вызывает метод release (). Он был добавлен в класс, чтобы его можно было использовать вместе с блочной конструкцией автоматического управления ресурсами.

Задание на практическую работу №2

Задание 1

Создать файл формата .txt, содержащий несколько строк текста. С помощью пакета java.nio нужно прочитать содержимое файла и вывести данные в стандартный поток вывода.

Задание 2

Реализовать копирование файла размером 100 Мб 4 методами:

1) FileInputStream/FileOutputStream
2) FileChannel
3) Apache Commons IO
4) Files class

Замерить затраты по времени и памяти и провести сравнительный анализ.

Задание 3

Реализовать функцию нахождения 16-битной контрольной суммы файла с использованием бинарных операций и ByteBuffer.

Задание 4

При помощи WatchService реализовать наблюдение за каталогом:

1) При создании нового файла в этом каталоге вывести его название;
2) При изменении файла вывести список изменений(добавленных и удаленных строк);
3) При удалении файла вывести его размер и контрольную сумму(использовать реализацию из задания 3).

Если реализовать пункт 3 не представляется возможным – докажите это.