

# 18. IO, NIO

Информатика, ИТИС, 1 курс

М.М.Абрамский

2017

```
PK.....!.ДвФ'm ..... [Content_Types].xml џ. (. . . . .  
.  
. . . . .  
  
..Д""НГЬ0.Ъп·ъ.Q@.1ЧЫнq'мѝ?Ѓю-ФН.ё  
Й$ЕМ.Лбтуы:.КЩ('!Бь.R.ЖуКАЯТgqs_WC.Ь·KJcАМг.TЦдR·иэогм:Г~.*.UJ K·`г>eуг<>-  
...щheУхехь^KHUP.>4."■R4|.Q_ль'h'э.%6y_-Q.">№vкxNx....XW.ьрпоч$Z-qф@>»кеЭж-  
чйhdO.g†Зfс...#.ZW2.Oп.ШЕ|РлlWGВ#»чш·Фц../ц.Uыdq.±.тЗ.I.Dxx·ч)- эGfd.C-0o<Ё  
эю.-.С.-яШ.-щwo'влѝђ.dM¶@э''Ибухи2@..T.e.f±·'ь,-y_/rPA/MV°ц■1hhBpьCA,mD9..ка  
%:Б+т,+т,чи.чи.oP.и...яь_.<.я<_.ья.ья_.k.яk_.тъ.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.  
.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.ят.  
m+ёшs»їKүѝт,,?%tiv»YЮyb&.NE...=)Lклk}*S..δk}LS>ж.|лмс.Yц|1...йҫ.V4ЖиLa:e4|@  
§T.Ÿ°Lo3JJk4Sl.зЯIx..-C3n█KЪq.ц#д±Yк>J..үүрӨФ.uжKc>'>t3зе....ая..PK.....  
...?.hwtŸ.....В....._rels/.rels џ. (. . . . .
```



# Файл

- Всегда набор битов!
  - Байтов, килобайтов, мегабайтов
  - У бита два возможных значения, поэтому файл называется **бинарным**.

# Чтение файла

- Каждое приложение знает свой формат файлов
  - Что означает байт на определенной позиции

# Пример – TIFF (отрывок)

- TIFF-файл начинается с 8-байтового заголовка, содержащего следующую информацию:
- Байты 0-1: Первое слово файла определяет порядок байтов, используемый в файле. Допустимыми его значениями являются: II (hex 4949) MM (hex 4D4D)
- Байты 2-3: Второе слово TIFF-файла - это номер версии. Это число, равное 42 (2A hex), но оно не равно номеру редакции текущей спецификации TIFF (в данном случае номер редакции текущей спецификации - это 5.0). Фактически номер версии TIFF (42) никогда не меняется и, возможно, никогда не изменится. Но если это случится, то будет означать, что TIFF изменился настолько радикально, что программа чтения TIFF должна немедленно прекратить работу. Число 42 было выбрано из-за его глубокого философского смысла. Оно может и должно использоваться для дополнительной проверки того, что это действительно TIFF-файл. TIFF-файлы не имеют явного номера редакции (т.е. 5.0 для текущей редакции).
- Байты 4-7: Это слово типа long, содержащее смещение в байтах первой директории файла (Image File Directory). Директория может располагаться в любом месте файла вслед за заголовком, но ее начало должно быть выровнено на границу слова. В частности, директория может следовать за данными изображения, которое она описывает. Программы чтения должны просто перемещаться по этому указателю, вне зависимости от того, куда он указывает. (Термин байтовое смещение (byte offset) всегда используется в этом документе, чтобы ссылаться на положение относительно начала файла. Первый байт файла имеет смещение, равное 0). Директории файла (Image File Directory) Директории файла (Image File Directory - IFD) состоят из 2-байтового счетчика числа элементов (т.е. числа тегов в данной директории), вслед за которым расположена последовательность 12-байтовых тегов и далее 4-байтовое смещение для следующей директории (или 0, если таковая отсутствует). Не забывайте записывать 4 нулевых байта в конце последней директории! Каждый 12-байтный элемент IFD имеет следующий формат: Байты 0-1 содержат Тег (Tag) поля. Байты 2-3 содержат Тип (Type) поля. Байты 4-7 содержат Длину (Length) поля (здесь, возможно, более удачным термином является Count - Счетчик). Байты 8-11 содержат Смещение для значения (Value Offset), т.е. байтовое смещение того места в файле, где расположено само значение. Предполагается, что это смещение должно быть выровнено на границу слова, т.е. Value Offset должно быть четным числом. Это смещение может указывать на любое место в файле. Элементы в IFD должны быть отсортированы в порядке возрастания поля Tag. Заметим, что этот порядок отличен от того, в котором поля описаны в данном документе. Упорядоченный по номерам список тегов приведен в Приложении Е. Значения, на которые указывают элементы директории, могут следовать в файле в любом порядке. Для экономии времени и пространства поле Value Offset интерпретируется как само значение, а не как указатель на значение, если значение умещается в 4 байтах. Если значение меньше 4 байтов, то оно выравнивается по левому краю 4-байтового поля, т.е. запоминается в байтах с младшими номерами. Для того, чтобы определить умещается или нет значение в 4 байтах, следует проверить значения полей Type и Length. - 13 - Поле Length описывает данные в терминах типов данных, а не общим числом байтов в поле. Например, одиночное 16-битное слово (SHORT) имеет Length равное 1, а не 2. Ниже приведены типы данных и их длины: 1 = BYTE 8-битное беззнаковое целое. 2 = ASCII 8-битные байты, которые содержат ASCII-коды; последний байт должен быть нулевым. 3 = SHORT 16-битное (2-байтовое) беззнаковое целое. 4 = LONG 32-битное (4-байтовое) беззнаковое целое. 5 = RATIONAL Два числа типа LONG: первое представляет числитель дроби, второе - ее знаменатель.

# Текстовый редактор

- Интерпретирует каждые 1 (или 2) байта как код символа – и отображает символ на экране!
  - Но далеко не все байты бинарных файлов означают символы.
    - » А такие файлы, который состоят из символов, называются текстовыми.

# Текстовые / не текстовые

- Текстовые
  - html, java, cs, cpp, pas, ini, txt, css, js, csv, rtf, fb2
    - Могут содержать нетекстовые вставки
- Не текстовые
  - Doc, docx, xls, bmp, jpg, djvu, pdf
    - Могут содержать текстовые вставки



# Почему не все файлы текстовые

- См. пример с 65 и 66

**А ТОЛЬКО ЛИ ИЗ ФАЙЛОВ МОЖНО  
СЧИТЫВАТЬ ИНФОРМАЦИЮ? А  
ПЕРЕДАВАТЬ/ЗАПИСЫВАТЬ?**

# Откуда еще?

- Консоль (куда без нее)
- Сеть (во всех смыслах)
- Устройства
- И даже оперативная память
  - структуры данных, хранящиеся в ней

**НО ТАК ЛИ РАЗЛИЧЕН ВВОД  
ДАННЫХ С ЭТИХ ИСТОЧНИКОВ?**

# Общие особенности:

- Количество байтов заранее неизвестно
- Ввод идет побайтово.
- Вывод идет побайтово.
- Современный подход к вводу/выводу – есть **объекты**, которые, работая с разными источниками данных (файл, сеть и др.), имеют общий интерфейс
  - Источник данных может поменяться, но это никак не повлияет на код
- Такие объекты называются...

# ПОТОКИ? ОПЯТЬ ПОТОКИ?

- Streams, потоки – не связаны ни с потоками-тредами (Thread), ни с потоками-стримами (Stream API).

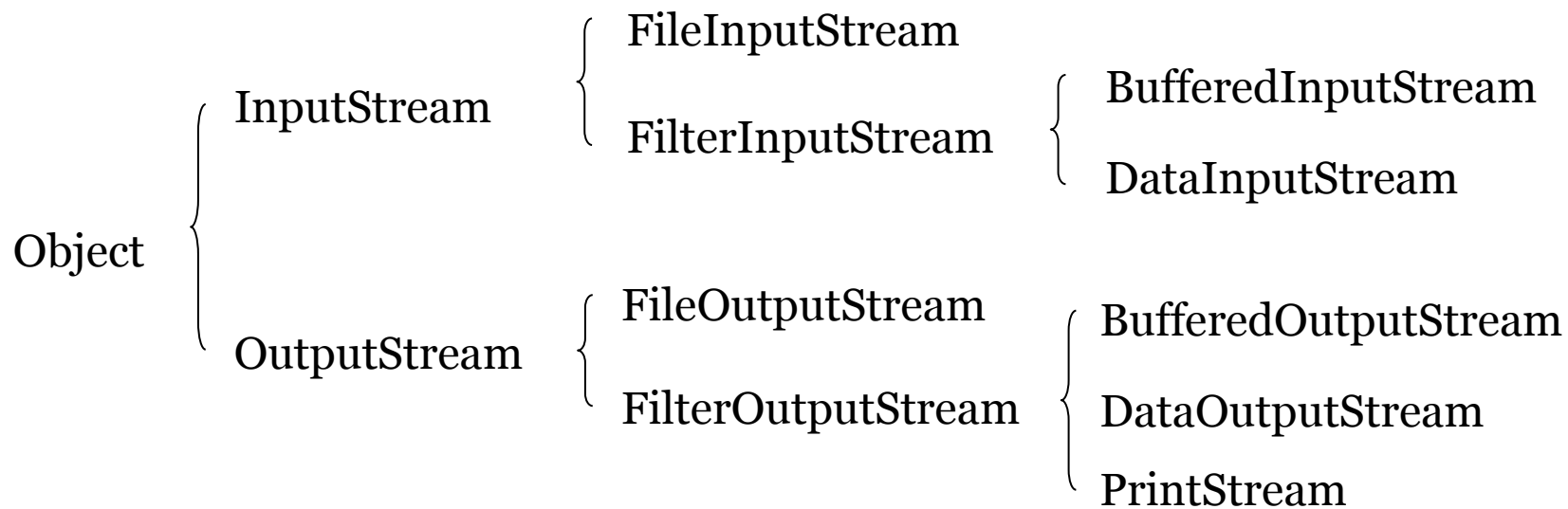
# **java.io – два важнейших класса!**

- InputStream – входной поток
  - read()
- OutputStream – выходной поток
  - write(x)

# Их наследники

- BufferedInputStream
- ByteArrayInputStream
- DataInputStream
- FileInputStream
- FilterInputStream
- LineNumberInputStream
- ObjectInputStream
- PipedInputStream
- PrintStream
- PushbackInputStream
- SequenceInputStream
- StringBufferInputStream
- BufferedOutputStream
- ByteArrayOutputStream
- DataOutputStream
- FileOutputStream
- FilterOutputStream
- ObjectOutputStream
- PipedOutputStream
- PipedOutputStream





# Как работать с потоком

```
import java.io.*;
```

- *Открыть поток*
- *Пока есть информация, читать с него*
- *Заккрыть поток*

# Тонкости read

- Метод `read()` считывает один байт и возвращает его в виде `int`
  - Зачем, кстати?
- Если достигнут конец ввода, то возвращается `-1` — и это означает, что ввод нужно прекратить.

```
import java.io.*;
public class Echo {
    public static void main(String[] args) {
        echo(System.in);
    }
    public static void echo(InputStream in) {
        try {
            int i = in.read();
            while (i != -1) {
                char c = (char) i;
                System.out.print(c);
                int i = in.read();
            }
        } catch (IOException e) {
            System.err.println(e);
        }
        System.out.println();
    }
}
```

# Чтение нескольких байтов сразу

- Чтение/запись дорогая операция. Чем меньше раз ее вызывать — тем лучше.
- Перегрузка метода `read` — чтение нескольких байтов сразу.
  - `public int read(byte b[])`
  - `public int read(byte b[], int offset, int length)`

# Чтение нескольких байтов сразу

```
byte[ ] b = new byte[10];  
int j = inputStream.read(b);
```

- `j` — количество реально считанных байтов.

# Откуда можно прочитать данные

Так и называется соответствующий `InputStream`.


- `ByteArrayInputStream`
- `FileInputStream`
- `StringBufferInputStream`
- `PipedInputStream` (в связке с `PipedOutputStream`)
  - Что записали в `pipedOutput`, то будет доступно в `pipedInput`

# Использование FileInputStream

```
import java.io.*;
```

Имя файла

```
class FileInputStreamDemo {  
    public static void main(String args[]) {  
        try {  
            FileInputStream fis = new FileInputStream(args[0]);  
            int i;  
            while ((i = fis.read()) != -1) {  
                System.out.println(i);  
            }  
            fis.close();  
        } catch (Exception e) {  
            System.out.println("Exception: " + e);  
        }  
    }  
}
```






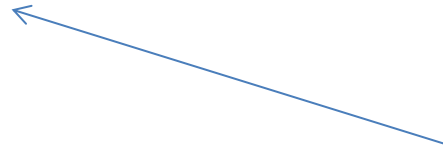
**А ДАВАЙТЕ-КАК ВЗГЛЯНЕМ НА  
НАСЛЕДНИКОВ INPUT STREAM**

# Так, стоп.

**ByteArrayInputStream**  
**StringBufferInputStream**  
**FileInputStream**  
...  
**BufferedInputStream**  
**DataInputStream**  
**FilterInputStream**



Тут понятно,  
откуда читать.



А тут? Из буфера? Из  
фильтра? Из данных?  
Шта?



**БУФЕРИЗОВАННЫЕ ПОТОКИ**

# Буфер

- Промежуточное хранилище.

```
BufferedInputStream bis =  
new BufferedInputStream(  
new FileInputStream("hello.txt");
```

...

```
bis.mark(); x = bis.read();... bis.reset();
```

```
BufferedOutputStream bos = ...  
bos.write(x);... bos.flush();
```

# Buffered Streams

- The `java.io.BufferedInputStream` and `java.io.BufferedOutputStream` classes buffer reads and writes by first storing the in a buffer (an internal array of bytes). Then the program reads bytes from the stream without calling the underlying native method until the buffer is empty. The data is read from or written into the buffer in blocks; subsequent accesses go straight to the buffer.

# BufferedПотоки

- Что такое буфер? (! если забыли)
  - `BufferedInputStream`,
  - `BufferedOutputStream`,
- `flush()` ?
- `mark()` ?
- `reset()` ?

# Надстройки

- ?



# Надстройки

- Возможность подставлять одни потоки в другие, получая новую функциональность

```
new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("input.dat")  
    )  
)
```

Buffered – поддержка буферизации,  
DataInput – возможность читать примитивы.

# Reading from Buffered Input Streams

```
import java.io.*;
class FileBufferedStreamDemo {
    public static void main(String args[]) {
        try { //Create a file input stream
            FileInputStream fis = new FileInputStream(args[0]);
            //Create a buffered input stream
            BufferedInputStream bis = new BufferedInputStream(fis);
            //read 12 byte from the file
            int i;
            while ((i = bis.read()) != -1)
                { System.out.println(i); }
            //Close file output stream
            fis.close();
        }
    }
}
```

# DataInput пример

```
try {  
    DataInputStream dis = new DataInputStream(  
        new FileInputStream(args[0])  
    );  
    System.out.println(dis.readBoolean());  
    System.out.println(dis.readByte());  
    System.out.println(dis.readChar());  
    System.out.println(dis.readDouble());  
    System.out.println(dis.readFloat());  
    System.out.println(dis.readInt());  
    System.out.println(dis.readLong());  
    System.out.println(dis.readShort());  
    dis.close();  
} catch (Exception e) {  
    System.out.println("Exception: " + e);  
}
```

# Random Access Files

- Наследник как `DataInputStream` и `DataOutputStream`.
  - "r" for read-only access
  - "rw" for read and write access. Java does not support write only access. For example,
- `RandomAccessFile raf = new RandomAccessFile("29.html", "r");`

# Методы

- `getFilePointer()` – текущая позиция
- `length()` – длина
- `seek()` – перемещение на позицию
  - `seek` на большую позицию, чем длина – перемещение в конец
  - Запись в конец файла расширяет файл.

**READERS/WRITERS**

- `read()` – char
- `write(x)`, `x` – char
- `PrintWriter.println()`
- `BufferedReader.readLine()`

# Мост

- InputStreamReader
- OutputStreamWriter



- `InputStream getIS() {}` // скомпилирован в jar

```
new BufferedReader(  
    new InputStreamReader(  
        getIS()));
```

**СЕРИАЛИЗАЦИЯ**

# Сериализация и десериализация

- **Сериализация** – запись объекта в байтовый поток
- **Десериализация** – чтение объекта из байтового потока

При сериализации и десериализации используется все данные объекта

# Сериализация объектов

- Запись объектов
  - Интерфейс `ObjectOutput` extends `DataOutput`
  - Класс `ObjectOutputStream`
  - Метод `writeObject(object)`
  - Исключение `NotSerializableException`

# Десериализация объектов

- Чтение объектов
  - Интерфейс `ObjectInput` extends `DataInput`
  - Класс `ObjectInputStream`
  - Метод `readObject()`
  - Исключение
    - `ClassNotFoundException`
    - `InvalidClassException`

# Что можно сериализовать

- Автоматически сериализуемые классы
  - Маркерный интерфейс Serializable
- Классы, сериализуемые в ручную
  - Интерфейс Externalizable

# Автоматическая сериализация

- Процесс записи
  - Записывается предок
  - Записываются значения всех полей, не имеющих модификатора **transient**
- Процесс чтения
  - Выделяется память под объект
  - Считывается предок
  - Считываются значения всех полей , не имеющих модификатора **transient**

# Версии сериализованных классов

- Применяется для обеспечения совместимости когда версии сериализованного объекта меняются
- Поле
  - `private final static long serialVersionUID`
- Инструмент
  - `serialver <имя класса>`



**NIO**

# Потоковая безопасность (thread safety)

- ? (! если забыли)
- Java IO – синхронизировано всё!
  - Осуществляется блокирующий (синхронизированный) ввод/вывод

# NIO

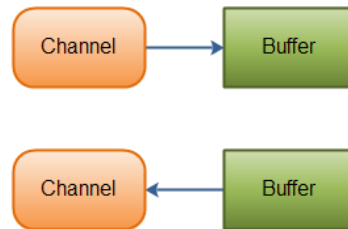
- Non-blocking IO
- Данные
  - Читаются из каналов в буферы
  - Записываются из буферов в каналы

# NIO. Каналы

- Это типа такие «*потоки*» («*streams*»)
    - *ОПЯТЬ ПОТОКИ? СЕРЬЕЗНО?*
  - FileChannel
  - SocketChannel
  - ServerSocketChannel
  - DatagramChannel
- 
- Работают только с Буферами

# NIO. Буферы

- Получают данные из каналов,
- Отправляют данные в каналы.

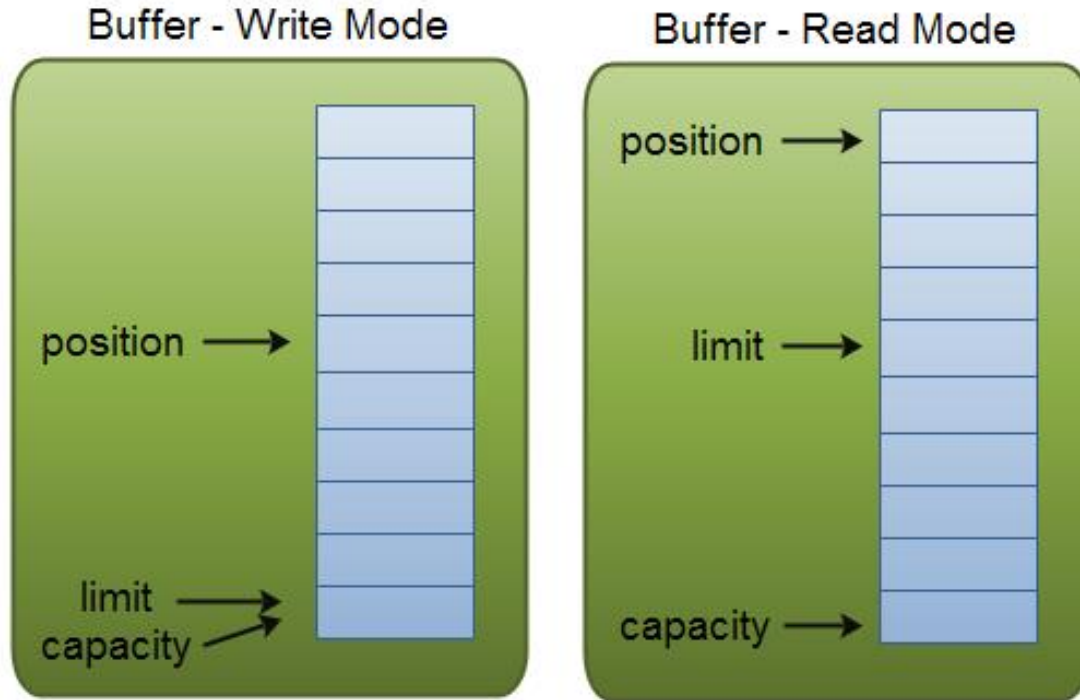


Это может делать каждый буфер!

- Нет разделения на входные/выходные буферы
- Ячейки буфера – примитивы
  - ByteBuffer, IntBuffer, LongBuffer...

- Channel -> Buffer
  - change mode of Buffer
  - Buffer -> program
- 
- program -> Buffer
  - change mode of Buffer (write)
  - Buffer -> Channel

# Режимы буфера



# Чтение из файла с помощью NIO

```
RandomAccessFile aFile = new RandomAccessFile("data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
// создаем буфер размера 48 байтов (!)
ByteBuffer buf = ByteBuffer.allocate(48);
// читаем из канала в буфер, возвращается количество считанных байтов
int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {
    // режим чтения полученных данных из буфера
    buf.flip();
    while (buf.hasRemaining()) {
        System.out.print((char) buf.get());
    }
    // режим записи новых данных в буфер
    buf.clear();
    bytesRead = inChannel.read(buf);
}
aFile.close();
```



**ОТВЕТЫ НА ВОПРОСЫ**



«Плюсы, минусы и практика использования функционального стиля при разработке»

«нужно несколько ярких практических примеров [лямбда-выражений], с пояснениями»

«Практических примеров stream api, уже на java, тоже просят»

«Еще пожелание рассмотреть функциональные интерфейсы, и как работать со встроенными типа Predicate, и др. Также работа с Future)»