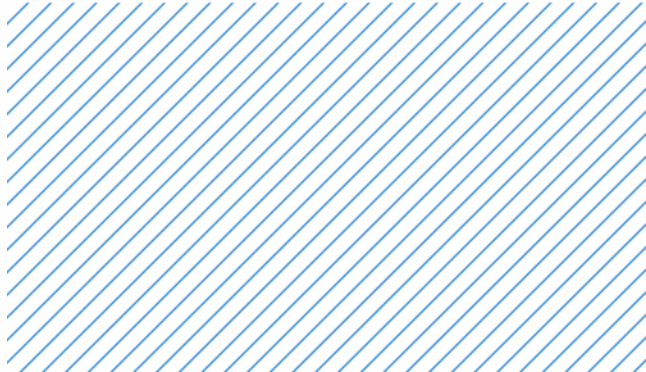
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 17

## Потоки данных

# Программирование на языке Java

Роман Гуров
ВШЭ БИ 2021



# Потоки байт

Мы уже научились работать с файловой системой: создавать, удалять, переименовывать файлы, а также перечислять содержимое директорий

Но мы всё ещё не знаем, как работать с содержимым самих файлов

Для чтения и записи содержимого файлов в `Java` используется понятие *потоков ввода и вывода*

Важно не путать эти потоки с известными нам Streams API, между ними нет ничего общего, кроме названия

Потоки есть как для чистых байтов, так и для текстовых символов

Для байтового ввода-вывода есть два *абстрактных* класса: `java.io.InputStream` и `java.io.OutputStream`

Заметим, что сами потоки не имеют никакой привязки к понятию файла, и могут применяться и в других ситуациях, например, для отправки/получения данных через интернет

# InputStream

`InputStream` – поток байтов, из которого можно читать как по по одному байту, так и целыми блоками

```
public abstract class InputStream implements Closeable {
    public abstract int read() throws IOException;

    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }

    public int read(byte b[], int off, int len)
        throws IOException {
        // ...
    }

    public long skip(long n) throws IOException {
        // ...
    }

    public void close() throws IOException {}
    // ...
}
```

`read()` читает из потока ровно один байт и возвращает его

Если поток закончился, возвращается `-1` (потому и `int`)

Чтобы получить сам `byte`, нужно использовать преобразование:

```
byte b = (byte) stream.read();
```

Есть две версии `read` для чтения сразу нескольких байт:

Первая принимает массив и пытается считать сколько в него влезет

Вторая позволяет задать индекс в массиве, с которого его надо начать заполнять, и лимит на количество читаемых байт

Оба метода возвращают количество прочитанных (=записанных в массив) байт, и `-1`, если поток закончился и читать было нечего

Потоку разрешается читать меньшее число байтов, чем просили, даже если он не закончился

Поэтому обязательно всегда проверять возвращённое значение

# InputStream

`InputStream` – поток байтов, из которого можно читать как по по одному байту, так и целыми блоками

```
public abstract class InputStream implements Closeable {
    public abstract int read() throws IOException;

    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }

    public int read(byte b[], int off, int len)
        throws IOException {
        // ...
    }

    public long skip(long n) throws IOException {
        // ...
    }

    public void close() throws IOException {}
    // ...
}
```

`skip()` пропускает указанное количество байт в стриме: можно считать, что просто читает их в никуда

Возвращает число успешно пропущенных, или `-1`

`close()` закрывает поток и освобождает системные ресурсы, поэтому потоки удобно использовать в `try-with-resources`

Все методы в случае ошибки бросают проверяемое исключение – `IOException`

# OutputStream

`OutputStream` – поток байтов, в который можно писать как по по одному байту, так и целыми блоками

```
public abstract class OutputStream
    implements Closeable, Flushable {

    public abstract void write(int b) throws IOException;

    public void write(byte b[]) throws IOException {
        write(b, 0, b.length);
    }

    public void write(byte b[], int off, int len)
        throws IOException {
        // ...
    }

    public void flush() throws IOException {
        // ...
    }

    public void close() throws IOException {
        // ...
    }
}
```

Методы `write()` аналогичны `read()` у `InputStream`'а

Но они, естественно, записывают байты в поток

Поток вывода не обязательно сразу записывает байты в точку назначения, он может накопить некоторый буфер, чтобы потом записать всё большой пачкой

Если такое поведение вас не устраивает,
то можно, после операций записи, явно попросить поток слить весь буфер методом `flush()`

`flush` самостоятельно делается при вызове `close()`,
так что не нужно руками `flush`'ить поток перед закрытием

# Пример: копирование из потока в поток

Пусть нам дан входной поток и нужно просто перенаправить все байты в некоторый (тоже данный) выходной

```
int totalBytesWritten = 0;
byte[] buf = new byte[1024];
int blockSize;
while ((blockSize = inputStream.read(buf)) > 0) {
    outputStream.write(buf, 0, blockSize);
    totalBytesWritten += blockSize;
}
```

Мы не можем просто соединить их как две трубы,
поэтому приходится использовать промежуточный буфер для хранения пачки прочитанных данных

После прочтения блока данных, этот блок отправляется в поток вывода, и так пока поток ввода не закончится

# Потоки из файлов

Рассмотрим способ создать байтовый поток для файла

Такую возможность дают классы `FileInputStream` и `FileOutputStream` – наследники `InputStream` и `OutputStream`:

```
InputStream inputStream =
    new FileInputStream(new File("in.txt"));

OutputStream outputStream =
    new FileOutputStream(new File("out.txt"));
```

Не забываем, что `File` считается устаревшим, и нынче принято использовать `Path`

Для получения этих файловых поток из `Path` есть статические методы класса `Files`:

```
InputStream inputStream =
    Files.newInputStream(Paths.get("in.txt"));

OutputStream outputStream =
    Files.newOutputStream(Paths.get("out.txt"));
```

# Возможность читать файлы из `classpath`

Есть простой способ обратиться к файлам, лежащим в `classpath`:

```
try (InputStream inputStream = Main.class.getResourceAsStream("Main.class")) {
    int read = inputStream.read();
    while (read >= 0) {
        System.out.printf("%02x", read);
        read = inputStream.read();
    }
}
```

При помощи такого кода, класс `Main` может прочитать свой собственный байт-код

Помним, что `classpath` может содержать как обычные директории, так и `jar`-архивы

В обоих случаях всё успешно сработает, причём для нас разницы видно не будет

Таким образом, можно запаковывать в `jar`-архив не только `.class`-файлы, но и любые ресурсы, нужные для работы нашей программы, например, картинку с логотипом вашей компании, которая отображается при запуске

# Потоки на массивах

Потоки не обязательно создавать только на файлах, можно сделать поток над массивом:

```
byte[] data = {1, 2, 3, 4, 5};
InputStream inputStream = new ByteArrayInputStream(data);

ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
// ...
byte[] result = outputStream.toByteArray();
```

Этим удобно пользоваться, если есть существующий код, работающий на потоках,
но захотелось использовать его с уже прочитанными данными, или даже просто протестировать

# Оборачивание стримов

Поток байтов сам по себе очень базовый и малофункциональный, чтобы вывести в него `int`, придётся тратить время на написание ручной конвертации инта в байты

Чтобы расширить возможность потока, в языке есть возможность обернуть один стрим в другой

Внутренний стрим будет производить низкоуровневую работу – реальную запись в файл, например

А внешний стрим будет декорировать внутренний – добавлять новую полезную функциональность

Например, рассмотрим такие стримы-обёртки, позволяющие в байтовом стриме оперировать разными типами

# Оборачивание стримов

Например, рассмотрим такие стримы-обёртки, позволяющие в байтовом стриме оперировать разными типами

```
public class DataOutputStream
    extends FilterOutputStream implements DataOutput {
    public DataOutputStream(OutputStream out) {
        // ...
    }

    public final void writeInt(int v) throws IOException {
        out.write((v >>> 24) & 0xFF);
        out.write((v >>> 16) & 0xFF);
        out.write((v >>> 8) & 0xFF);
        out.write((v >>> 0) & 0xFF);
        incCount(4);
    }
    // ...
}
```

```
public class DataInputStream
    extends FilterInputStream implements DataInput {
    public DataInputStream(InputStream in) {
        // ...
    }

    public final int readInt() throws IOException {
        int ch1 = in.read();
        int ch2 = in.read();
        int ch3 = in.read();
        int ch4 = in.read();
        if ((ch1 | ch2 | ch3 | ch4) < 0)
            throw new EOFException();
        return ((ch1 << 24) + (ch2 << 16)
            + (ch3 << 8) + (ch4 << 0));
    }
    // ...
}
```

С их помощью можно обернуть существующий байтовый стрим и читать / писать не просто байты,
а уже более интересные типы данных (в том числе строки)

Но это всё ещё работа с сырыми байтами, текстовые документы так не делаются

# Сжимающие и разжимающие потоки

Ещё одно интересное применение стримов-обёрток – добавление сжатия данных к стриму

Классы `DeflaterOutputStream` и `InflaterInputStream` позволяют на лету сжимать и разжимать данные из потока при помощи алгоритма сжатия `Deflate`:

```
byte[] originalData = {1, 2, 3, 4, 5};
ByteArrayOutputStream os = new ByteArrayOutputStream();
try (OutputStream dos = new DeflaterOutputStream(os)) {
    dos.write(originalData);
}

byte[] deflatedData = os.toByteArray();
try (InflaterInputStream iis = new InflaterInputStream(new ByteArrayInputStream(deflatedData))) {
    int read = iis.read();
    while (read >= 0) {
        System.out.printf("%02x", read);
        read = iis.read();
    }
}
```

Обратим внимание, что закрытие стрима-обёртки закрывает и внутренний стрим, поэтому закрывать внутренний какими-то мудрёными путями не требуется

# Символьные потоки

Потоки байтов годятся только для работы с бинарными данными,
текст с ними записывать, конечно, можно, но это будет неудобно из-за специфики хранения текстов

Для работы с потоками символов есть большое семейство классов

Самые базовые из них – абстрактные классы `Reader` и `Writer`

# Символьные потоки

Reader и Writer выглядят уже нам знакомыми:

```
public abstract class Reader implements Readable, Closeable {
    public int read() throws IOException {
        // ...
    }

    public int read(char cbuf[]) throws IOException {
        return read(cbuf, 0, cbuf.length);
    }

    public abstract int read(char cbuf[], int off, int len)
        throws IOException;

    public long skip(long n) throws IOException {
        // ...
    }

    public abstract void close() throws IOException;
    // ...
}
```

```
public abstract class Writer
    implements Appendable, Closeable, Flushable {
    public void write(int c) throws IOException {
        // ...
    }

    public void write(char cbuf[]) throws IOException {
        write(cbuf, 0, cbuf.length);
    }

    public abstract void write(char cbuf[], int off, int len)
        throws IOException;

    public abstract void flush() throws IOException;

    public abstract void close() throws IOException;
    // ...
}
```

Все как у байтовых потоков, но вместо байтов – char

Соотвественно в случае инта, сам символ хранится в младших двух его байтах,
и получается преобразованием этого инта к char

# Создание символьных потоков

Произвольный поток байтов можно превратить в поток символов с помощью классов-обёрток:

```
Reader reader =
    new InputStreamReader(inputStream, "UTF-8");

Charset charset = StandardCharsets.UTF_8;
Writer writer =
    new OutputStreamWriter(outputStream, charset);
```

При этом, нужно указывать кодировку, согласно которой байты будут превращаться в символы

Кодировки бывают разными, и могут иметь как переменную, так и фиксированную длину символа

Например в `UTF-8` один символ может занимать от 1 до 4 байтов, и в ней представим любой символ Юникода

А в какой-нибудь `Windows-1251` каждый символ занимает ровно 1 байт, но и набор символов там невелик

Передать кодировку в конструктор стрима можно как её строчным названием, так и объектом `Charset`

# Charset

Charset можно получить несколькими способами

Существует класс `StandardCharsets`, содержащий набор констант
с гарантированно доступными на любой `JVM` кодировками:

```
package java.nio.charset;

public final class StandardCharsets {
    public static final Charset US_ASCII = sun.nio.cs.US_ASCII.INSTANCE;
    public static final Charset ISO_8859_1 = sun.nio.cs.ISO_8859_1.INSTANCE;
    public static final Charset UTF_8 = sun.nio.cs.UTF_8.INSTANCE;
    public static final Charset UTF_16BE = new sun.nio.cs.UTF_16BE();
    public static final Charset UTF_16LE = new sun.nio.cs.UTF_16LE();
    public static final Charset UTF_16 = new sun.nio.cs.UTF_16();
}
```

Метод `Charset.forName(String name)` позволяет получить кодировку по её имени,
и потенциально может предоставлять больший их ассортимент,
но тогда никто не обещает, что ваша программа найдёт эту кодировку на всех платформах

Если не передавать в конструктор символьного потока кодировку, то будет использована кодировка по-умолчанию, которую можно узнать вызовом метода `Charset.defaultCharset()`

Кодировка по-умолчанию зависит от операционной системы и текущих её настроек, то есть, очень изменчива

# FileReader и FileWriter

Вообще, в языке есть удобные классы, позволяющие создать символьный поток из файла напрямую:

```
Reader reader = new FileReader("in.txt");
Writer writer = new FileWriter("out.txt");
```

Но они, вплоть до Java 11, не умеют принимать кодировку, из-за чего работают только с дефолтной

Поэтому, часто прописывают всю последовательность вложенных потоков вручную:

```
Reader reader2 = new InputStreamReader(
    new FileInputStream("in.txt"), StandardCharsets.UTF_8);
Writer writer2 = new OutputStreamWriter(
    new FileOutputStream("out.txt "), StandardCharsets.UTF_8);
```

А ещё, так можно создать File<In|Out>putStream из Path

# Символьные потоки с массивами и строками

Можно создать Reader, читающий символы из char-массива или строки:

```
Reader reader = new CharArrayReader(
    new char[]{'a', 'b', 'c'});

Reader reader2 = new StringReader("Hello, World!");
```

И аналогичные Writer'ы:

```
CharArrayWriter writer = new CharArrayWriter();
writer.write("Test");
char[] resultArray = writer.toCharArray();

StringWriter writer2 = new StringWriter();
writer2.write("Test");
String resultString = writer2.toString();
```

# BufferedReader

Для Reader'ов есть очень полезная обертка, добавляющая буферизацию к чтению

Даже если читать символы поштучно, BufferedReader будет наперёд запрашивать их большими блоками, и отвечать на будущие чтения без лишних обращений к вложенному Reader'у

Это даёт огромный прирост производительности, особенно на больших объёмах данных

```
public class BufferedReader extends Reader {
    public BufferedReader(Reader in) {
        // ...
    }

    public String readLine() throws IOException {
        // ...
    }
    // ...
}
```

Также, BufferedReader предоставляет полезный метод readLine(), позволяющий получить следующую строчку в тексте, до ближайшего символа переноса строки (сам символ не включается), в конце потока вернётся null

# Пример: построчное чтение файла

Допустим, надо прочитать и обработать файл построчно

В старых версиях Java строили такую многоэтажку из вложенных потоков:

```
try (BufferedReader reader =
         new BufferedReader(
             new InputStreamReader(
                 new FileInputStream("in.txt"),
                 StandardCharsets.UTF_8))) {
    String line;
    while ((line = reader.readLine()) != null) {
        // process line
    }
}
```

Но с java.nio.file появился удобный способ создать это всё одним действием:

```
try (BufferedReader reader = Files.newBufferedReader(
    Paths.get("in.txt"), StandardCharsets.UTF_8)) {
    String line;
    while ((line = reader.readLine()) != null) {
        // process line
    }
}
```

# BufferedWriter

Симметрично классу `BufferedReader`, есть и `BufferedWriter`

```
try (BufferedWriter writer = Files.newBufferedWriter(
    Paths.get("out.txt"), StandardCharsets.UTF_8)) {
    writer.write("Hello");
    writer.newLine();
}
```

Точно так же, он будет буферизировать внутри себя операции записи, и передавать их вложенному потоку отложенно, но сразу большим блоком символом (или через `flush`)

Также, у него есть метод `newLine()`, пишущий в поток символ переноса строки для текущей платформы

А если нужно просто записать небольшое количество данных,
то можно просто передать массив строк и `Path` к файлу в `Files.write()`:

```
List<String> lines = Arrays.asList("Hello", "world");
Files.write(Paths.get("out.txt"), lines,
    StandardCharsets.UTF_8);
```

# Форматированный вывод

`Writer`'ы, конечно, хороши, но не дают простого способа вывести в них, например, число (в его строчном виде)

Для этого есть специальный класс-обёртка `PrintWriter`

```
public class PrintWriter extends Writer {
    public PrintWriter(Writer out) {
        // ...
    }

    public void print(int i) {
        // ...
    }

    public void println(Object obj) {
        // ...
    }

    public PrintWriter printf(String format, Object... args) {
        // ...
    }

    public boolean checkError() {
        // ...
    }
    // ...
}
```

Мы уже с ним знакомы по `System.out.println()`

В нём есть методы print для каждого примитивного типа, а также для `Object` (вызывающий `toString()`)

`printf` позволяет напечатать объекты согласно переданной форматирущей строке (как `printf` в языке C)

В случае ошибки, все эти методы не бросают исключений, наличие ошибки смотрится через метод `checkError()`

# Форматированный вывод

Есть ещё класс `PrintStream`:

```
public class PrintStream extends FilterOutputStream
    implements Appendable, Closeable {
    public PrintStream(OutputStream out) {
        // ...
    }

    public void print(int i) {
        // ...
    }

    public void println(Object obj) {
        // ...
    }

    public PrintWriter printf(String format, Object... args) {
        // ...
    }

    public boolean checkError() {
        // ...
    }
    // ...
}
```

Почти то же самое, что `PrintWriter`

Но конструируется и `OutputStream`
и сам является `OutputStream`

То есть, работает с байтами

Получается гибрид `PrintWriter` и `OutputStream`

# Форматированный ввод

Чтобы читать из символьного потока целые объекты, существует знакомый нам класс `Scanner`

```
Reader reader = new StringReader(
    "abc|true|1,1e3|-42");

Scanner scanner = new Scanner(reader)
    .useDelimiter("\\|")
    .useLocale(Locale.forLanguageTag("ru"));

String token = scanner.next();
boolean bool = scanner.nextBoolean();
double dbl = scanner.nextDouble();
int integer = scanner.nextInt();
```

`Scanner` можно создать из `Reader`'а

Он позволяет прочитать из стрима любой примитивный тип или строку

`Scanner` имеет набор настроек

Например, можно поменять шаблон-разделитель, по которому текст разбивается на «токены» для сканнера

По умолчанию таковыми считаются пробельные символы, в примере мы заменили их на символ вертикальной черты: "|"

Занимателен также и параметр `Locale`, который меняет правила оформления чисел

Например, русская локаль требует записи десятичных чисел через запятую, а не точку

# Стандартные потоки

И вот чудо! Наконец-то мы можем понять потоки в классе `System`:

```
package java.lang;

public final class System {
    public static final InputStream in = null;
    public static final PrintStream out = null;
    public static final PrintStream err = null;
    // ...
}
```

Если System.in используется для текста, то его обычно
оборачивают в `InputStreamReader+BufferedReader`, или сразу в `Scanner`

`System.out` и `System.err` – экземпляры `PrintStream`, то есть позволяют и печатать текст через `print()`, и писать бинарные данные через `write()`

`System.err` – поток вывода, использующийся для вспомогательной информации, типа логов и ошибок, которую не хочется перепутать с реальным выводом (результатом работы) программы

Обычно они оба выводятся одинаково в консоль, но в ОС есть возможность перенаправления потока, например, в файл или на вход другой программе. Таким образом их можно разделить

# Сериализация

Может возникнуть желание сохранить объект произвольного класса в файл,
с возможностью восстановить его оттуда в будущем и снова получить этот же объект

Такой процесс консервации в байты называется сериализацией,
а обратный процесс получения объекта – десериализацией

Такая возможность в языке есть. Для начала, нужно пометить класс маркерным интерфейсом `Serializable`,
чтобы указать `JVM`, что для него требуется поддержка сериализации

Если какое-то поле класса не является необходимым для сохранения (например, получаемое и кэшируемое при первом обращении), то такое поле можно пометить ключевым словом `transient`, для экономии размера сохраняемых данных

# Сериализация

```
class Client implements Serializable {
    private long id;
    private String name;
    private LocalDate birthDate;
    private transient int ageInYears;

    public long getID() { return id; }

    public void setID(long id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public LocalDate getBirthDate() { return birthDate; }

    public void setBirthDate(LocalDate birthDate) { this.birthDate = birthDate; }

    public int getAgeInYears() {
        if (ageInYears == 0) {
            ageInYears = birthDate.until(LocalDate.now()).getYears();
        }
        return ageInYears;
    }
}
```

# Сериализация

Для (де-)сериализации объектов из стримов / в стримы
есть потоки-обёртки `ObjectInputStream` и `ObjectOutputStream`:

```
public static void main(String[] args) throws Exception {
    Client originalClient = new Client();
    originalClient.setID(1);
    originalClient.setName("Chuck Norris");
    originalClient.setBirthDate(LocalDate.of(1940, 3, 10));

    Path path = Paths.get("object.bin");
    try (ObjectOutputStream oos = new ObjectOutputStream(Files.newOutputStream(path))) {
        oos.writeObject(originalClient);
    }

    Client deserializedClient;
    try (ObjectInputStream ois = new ObjectInputStream(Files.newInputStream(path))) {
        deserializedClient = (Client) ois.readObject();
    }

    System.out.printf("%-15s %-30s\n", "ID", deserializedClient.getID());
    System.out.printf("%-15s %-30s\n", "Name", deserializedClient.getName());
    System.out.printf("%-15s %-30s\n", "Date of Birth", deserializedClient.getBirthDate());
    System.out.printf("%-15s %-30s\n", "Age", deserializedClient.getAgeInYears());
}
```

При сериализации объекта сериализуются и все его поля, значит все поля тоже должны быть сериализуемыми