лаба 6 - клиент и сервер

Сетевое взаимодействие - клиент-серверная архитектура, основные протоколы, их сходства и отличия.

Сетевой обмен

ИТМО

- Клиент
  - Работает на любом хосте (сервер не знает, где именно)
  - Свободный порт выбирается при отправлении запроса
  - Посылает запрос серверу, ждет ответ
- Запрос
  - Содержит данные и информацию о клиенте

- Сервер
  - Работает на известном хосте (известный IP-адрес)
  - Прослушивает известный порт (зависит от сервиса)
  - Ждет запрос от клиента, посылает ответ
- Ответ
  - Содержит данные

Запрос-ответ

Основной принцип взаимодействия между клиентом и сервером — это модель запрос-ответ. Клиент отправляет запрос на сервер, сервер обрабатывает этот запрос и отправляет ответ обратно клиенту. Этот процесс может включать передачу данных, выполнение операций или предоставление услуг.

Стоит обратить внимание, что при прекращении работы сервера клиент продолжит функционировать. Он просто потеряет связь.

Модель запрос-ответ может быть **синхронной или асинхронной**. В синхронной модели клиент ожидает ответа от сервера перед тем, как

продолжить выполнение своих задач. В асинхронной модели клиент может продолжать выполнять другие задачи, пока сервер обрабатывает запрос. Асинхронная модель часто используется для улучшения производительности приложений.

💡 **Протоколы общения**

Для взаимодействия клиента и сервера используются различные протоколы, такие как HTTP, HTTPS, FTP и другие. Эти протоколы определяют правила и формат обмена данными между клиентом и сервером. Например, HTTP (HyperText Transfer Protocol) широко используется для передачи веб-страниц.

Протоколы могут быть текстовыми или бинарными, в зависимости от требований к производительности и безопасности. Например, HTTPS (HTTP Secure) добавляет слой шифрования к HTTP, обеспечивая защиту данных при передаче (в HTTP данные передаются в открытом виде, так как HTTP сам по себе не предоставляет никаких средств шифрования, поэтому появился HTTPS).

Другие протоколы, такие как WebSocket, позволяют устанавливать постоянное соединение между клиентом и сервером, что полезно для приложений реального времени (чаты и онлайн-игры)

Можно почитать в этой статье подробнее про HTTP-запросы:

https://selectel.ru/blog/http-request/

Разница между TCP и UDP

Параметр	TCP	UDP
Установка соединения	Гарантируется устойчивое соединение перед передачей данных.	Передача информации осуществляется сразу.
Гарантии доставки	Обеспечивает гарантированною доставку данных с подтверждением.	Не гарантирует доставку. Здесь также отсутствует подтверждение получения информации.
Контроль ошибок	Задействует специальные механизмы для обнаружения и исправления ошибок.	Не предусматривает наличие специальных механизмов.
Порядок доставки	Строго правильный	Хаотичный
Подтверждение	Обладает механизмами подтверждения доставки и повторной передачи.	Не поддерживает подходящих механизмов.
Пропускная способность	Ниже	Выше
Примеры протоколов	HTTP, SMTP, FTP	VoIP, DNS

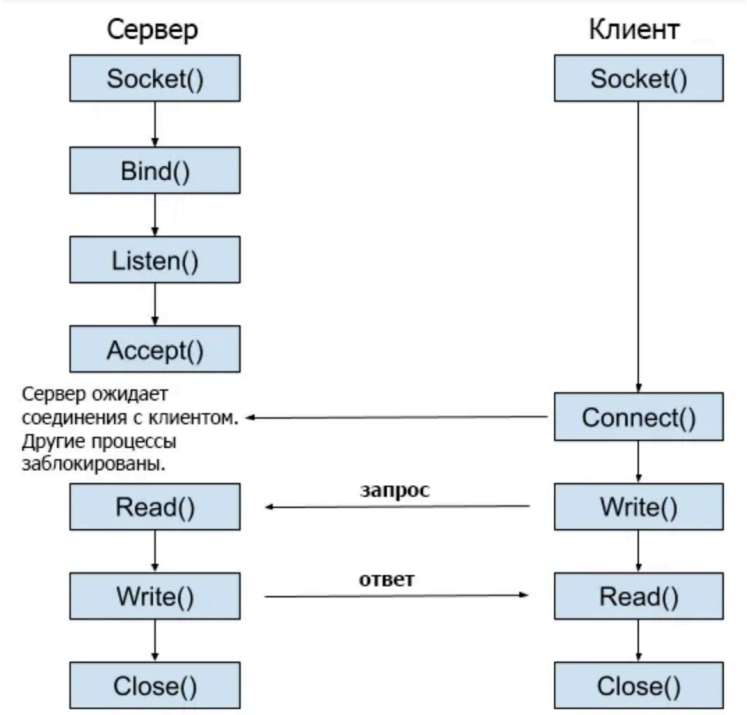
Протокол TCP. Классы Socket и ServerSocket.

TCP – технология передачи данных, выступающая одним из главных транспортных протоколов. Соответствующая технология появилась еще в 1974 году. С тех пор она претерпела множество трансформаций и стала одним из фундаментальных элементов современного Интернета.

TCP передает данные с пользовательского устройства на веб-сервера. Он является надежным и гарантирует доставку информации до получателя. Такой результат достигается за счет установки стабильного соединения.

TCP не только гарантирует получение, но и отправку сегментов в одном и том же порядке. Каждый пакет получает свой собственный номер, указывающий на позицию данных в потоке.

В Java TCP обычно используется через Интернет-протокол, который называется **TCP/IP**



Блок-схема коммуникации TCP-сокета

TCP-сокет устанавливает связь между клиентом и сервером в несколько этапов

https://proglang.su/java/networking

Сокеты лежат в основе современной работы в сети, т.к. сокет позволяет одному
компьютеру одновременно обслуживать множество разных клиентов и поддерживать
различные виды информации. Цель достигается за счет использования порта, который является нумерованным сокетом на отдельной машине. Говорят, что серверный процесс "прослушивает" порт до тех пор, пока к нему не подключится клиент

Когда одно приложение знает сокет другого, создается сокетное протоколо-ориентированное соединение по протоколу TCP/IP.

Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер прослушивает сообщение и ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер, в свою очередь, создает сокет, который будет использоваться для связи с клиентом, и посылает его клиенту с первым сообщением. После этого устанавливается соединение

- **Socket** — класс для создания объектов клиентского сокета
- **ServerSocket** — класс, создающий объекты, используемые серверными программами для организации соединения с программами клиентов

Чтобы клиент мог соединиться с сервером, он должен знать адрес и порт сервера, а затем создать объект клиентского сокета:

При этом указывается IP-адрес сервера и номер порта. Если указано символьное имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу.

```
//Конструкторы класса Socket
Socket(String имя_хоста, int порт) throws UnknownHostException, IOException
Socket(InetAddress IP-адрес, int порт) throws UnknownHostException
```

- «имя_хоста» — подразумевает под собой определённый узел сети, ip-адрес
- Если класс сокета не смог преобразовать его в реальный, существующий, адрес, то сгенерируется исключение UnknownHostException
- Если в качестве номера порта будет указан 0, то система сама выделит свободный порт. Также при потере соединения может произойти исключение IOException
- Следует отметить тип адреса во втором конструкторе — **InetAddress.** Он приходит на помощь, например, когда нужно указать в качестве адреса доменное имя. Класс

InetAddress используется для инкапсуляции числового IP-адреса и доменного имени для данного адреса.

Для взаимодействия с классом **InetAddress** применяется имя IP-хоста, что более удобно и понятно, чем его IP-адрес

Чтобы сервер мог принимать соединения от клиентов, он должен создать объект типа **ServerSocket*:***

```
ServerSocket server = new ServerSocket(int port);
```

Получив запрос на соединение, сервер должен согласиться на него и создать объект клиентского сокета:

```
Socket client = server.accept();
```

- Сервер создает экземпляр объекта ServerSocket, определяющий, по какому номеру порта должна происходить связь.
- Сервер вызывает метод accept() класса ServerSocket. Этот метод ожидает, пока клиент не подключится к серверу по указанному порту.
- По завершению ожидания сервера клиент создает экземпляр объекта сокета, указывая имя сервера и номер порта подключения.
- Конструктор класса Socket осуществляет попытку подключить клиента к указанному серверу и номеру порта. Если связь установлена, у клиента теперь есть объект Socket, способный связываться с сервером.
- На стороне сервера метод accept() возвращает ссылку к новому сокету на сервере, который подключен к клиентскому сокету.

Для возможности непосредственного обмена данными обе программы, и клиент, и сервер, должны создать входные и выходные потоки, используя классы пакета ***java.io*** и методы объектов класса **Socket**

```
InputStream in = client.getInputStream();
OutputStream out = client.getOutputStream();
```

Далее обмен данными между программами выполняется через объекты потоков ввода-вывода с помощью соответствующих
методов **read(...)** и **write(...)**

Так как читать и писать голые байты не так эффективно - потоки можно обернуть в буферизированные. Например:

```
BufferedReader in = newBufferedReader(newInputStreamReader(socket.getIn
putStream()));
BufferedWriter out = newBufferedWriter(newOutputStreamWriter(socket.getO
utputStream()));
```

Также можно отправлять объекты через потоки сокета

```
ObjectOutputStream out = newObjectOutputStream(socket.getOutputStream
());
ObjectInputStream in = newObjectInputStream(socket.getInputStream());
```

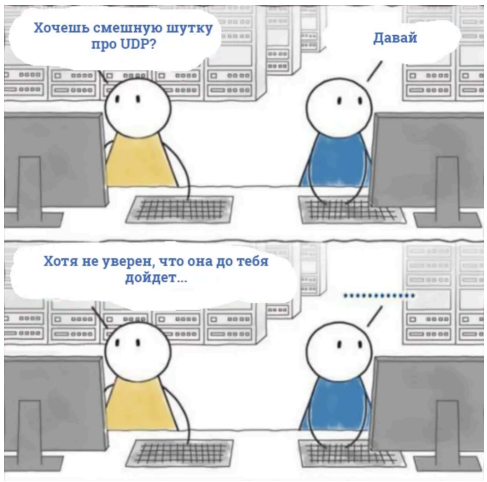
Протокол UDP.
Классы DatagramSocket и DatagramPacket.

UDP – более быстрая, но менее надежная технология обмена данными. Ее рекомендуется использовать там, где требуется непрерывный поток (потоковая передача в реальном времени).

Он появился чуть позже TCP – в 1980 году. С его помощью удастся организовать обмен пакетами по IP-сети без необходимости предварительной установки стабильных каналов или путей передачи данных.

UDP работает при помощи датаграмм – информационных блоков, передаваемых напрямую, без создания выделенного виртуального канала. Этот протокол не подразумевает отправку подтверждений. Часть датаграмм может быть утеряна в процессе информационного обмена. UDP часто используется в сервисах потокового видео, а также онлайн-играх.

Чтобы защитить этот вид передачи данных требуется использовать дополнительные меры. Примеры – прокси или туннельное соединение между пользователем и серверами организации.



Сокет протокола UDP создается объектом класса ***DatagramSocket***

Каждый объект класса ***DatagramSocket*** имеет следующие основные методы:

- **getInetAddress()** возвращает адресс, к которому осуществляется подключение;
- **getPort()** возвращает порт, к которому осуществляется подключение;
- **getLocalAddress()** возвращает локальный адресс компьютера, с которого осуществляется подключение;
- **getLocalPort()** возвращает локальный порт, через который осуществляется подключение;
- **send(DatagramPacket pack) — *передача пакета;***
- **receive(DatagramPacket pack) *прием пакета*.**

DatagramSocket имеет три конструктора:

```
DatagramSocket();
DatagramSocket(int port);
DatagramSocket(int port, InetAddress addr);
```

- Создаётся объект DatagramSocket(). У клиента он создается несвязанным (unbound) – в его конструкторе не указывается порт (он устанавливается через конструктор класса DatagramPacket)
- Подготавливается массив байтов (byte[]), он передаётся в конструктор, создаётся объект класса DatagramPacket(...)

Чтобы создать дейтаграмму для отправки на удалённую машину, используется следующий конструктор:

```
public DatagramPacket(byte[] ibuf, int length, InetAddress iaddr, int iport);
```

ibuf – массив байт, содержащий закодированное сообщение, *length* – количество байт, которое должно быть помещено в пакет(это определяет размер дейтаграммы)

iaddr – это экземпляр класса *InetAddress*, который хранит *IP*-адрес получателя

iport указывает номер порта, на который посылается дейтаграмма.

Чтобы получить дейтаграмму, необходимо использовать другой конструктор для объекта *DatagramPacket*, в котором будут находиться принятые данные. Прототип конструктора имеет вид.

```
public DatagramPacket(byte[] ibuf, int length);
```

ibuf – массив байт, куда должны быть скопированы данные из дейтаграммы, *length* – количество байт, которое должно быть скопировано.

- Отправка и получение пакета осуществляются через методы класса DatagramSocket:

метод **receive(DatagramPacket pack)**, который блокируется до момента получения

дейтаграммы, и метод **send(DatagramPacket pack)**

```
//Пример UDPClient
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.*;
public class UDPClient {
    DatagramPacket packet;
    DatagramSocket socket;
    InetAddress host;
    int port;

    public void connect() {
        try {
            socket = new DatagramSocket();
            host = InetAddress.getByName("localhost");
            port = 8882;
        } catch (UnknownHostException | SocketException e) {
            throw new RuntimeException(e);
        }
    }

    public boolean sendData(byte[] data) {
        try {
            packet = new DatagramPacket(data, data.length, host, port);
            socket.send(packet);
            return true;
        } catch (IOException e) {
            return false;

        }
    }

    public byte[] receiveData(int length) {
```

```
        try {
            byte[] data = new byte[length];
            packet = new DatagramPacket(data, length);
            socket.receive(packet);
            return data;
        } catch (IOException e) {
            return null;
        }
    }

    private byte[] serialize(Object obj) {
        try (ByteArrayOutputStream outputStream = new ByteArrayOutputStream())
            ObjectOutputStream objectOutputStream = new ObjectOutputStream(outp
            objectOutputStream.writeObject(obj);
            return outputStream.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    //также будет метод для десериализации ответа от сервера
}
```

Отличия блокирующего и неблокирующего ввода-вывода, их преимущества и недостатки. Работа с сетевыми каналами.

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**.

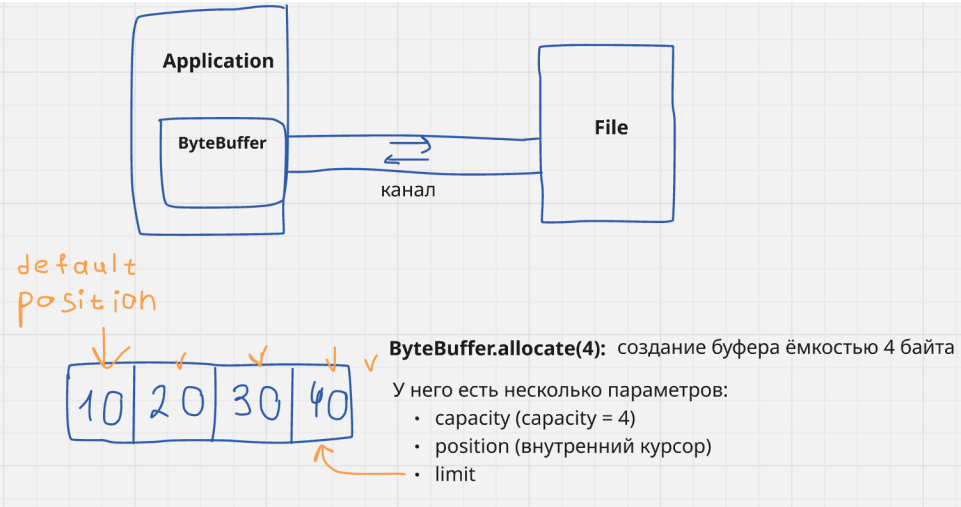
Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если вы хотите произвести подобные манипуляции,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

Подход, на котором основан **Java NIO** немного отличается.

Данные считываются в буфер для последующей обработки, буфер можно **воспринимать как высокоуровневую обёртку над массивом байт**. Вы можете двигаться по буферу вперед и назад. Это дает немного больше гибкости при обработке данных. Существует большое число методов по модификации его содержимого, навигации и т.д.

В то же время, необходимо проверять содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер вы не уничтожили ещё не обработанные данные, находящиеся в буфере.

В стандартной библиотеке буфер представлен абстрактным классом **Buffer** и множеством его наследников. Основное отличие наследников - тип данных, который они будут хранить: byte, int, long и другие примитивные типы данных. Поэтому рассмотрим **ByteBuffer** - все его ключевые особенности будут справедливы и для остальных классов.



- **get()** для чтение из буфера, с помощью которого можно считать из буфера 1 байт из места, на которое указывает маркер позиции
- **put()** для записи, и ситуация с ним аналогична методу get: запись происходит в место, на которое указывает position, position сдвигается, и нельзя выйти за пределы лимита, выйдет исключение
- **clear()** сбрасывает все метки в начальное состояние: postion - на первый элемент, limit - на последний
- **flip()** устанавливает маркер limit на текущее значение position, после чего перемещает position на начало массива
- **rewind()** - метод перемотки, используется, когда требуется перечитывание, так как он устанавливает позицию в 0 и не изменяет значение лимита

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.

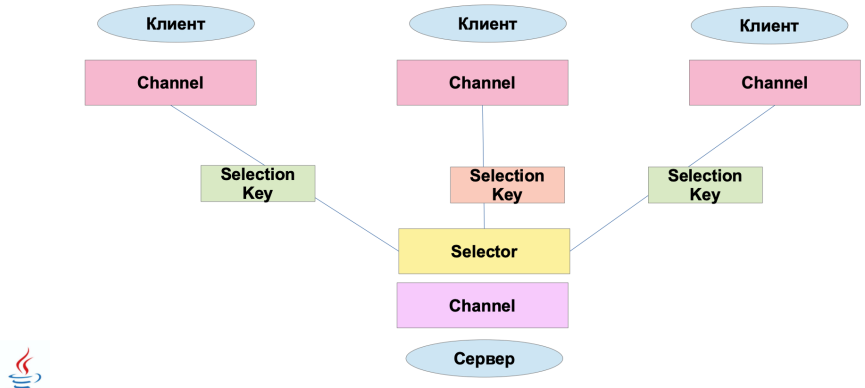
> 💡 Через **каналы** осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода данные, которые нужно отправить, помещаются в буфер, а он передается в канал. При вводе данные из канала помещаются в предоставленный вами буфер.

То же 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 пустой траты времени на ожидание в заблокированном состоянии. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах **(через Selector)**

Selector

Селекторы и каналы



Недостаток создания отдельных потоков: переключение контекста между потоками дорого обходится операционной системе, кроме того, каждый поток занимает память

Поэтому эффективнее использовать Selector

> 💡 Selector - объект, который позволяет одному потоку вместо нескольких одновременно контролировать несколько каналов ввода/вывода

Для использования селектора канал должен находиться в неблокирующем режиме. Это значит, что вы не можете использовать FileChannel с селектором, поскольку FileChannel нельзя переключить в неблокирующий режим

Мы можем зарегистрировать несколько каналов с помощью объекта-селектора. Когда активность ввода-вывода происходит на любом из каналов, селектор уведомляет нас. Вот так можно читать из большого количества источников данных в одном потоке

Любой канал, который мы регистрируем с помощью селектора, должен быть подклассом **SelectableChannel.** Это особый тип каналов, которые можно перевести в неблокирующий режим

Мы можем **прослушивать 4 разных события**, каждое из которых представлено константой в классе **SelectionKey** :

- **Соединиться —** когда клиент пытается подключиться к серверу. Представлен SelectionKey.OP_CONNECT
- **Принять —** когда сервер принимает соединение от клиента. Представлен SelectionKey.OP_ACCEPT
- **Чтение —** когда сервер готов читать из канала. Представлен SelectionKey.OP_READ
- **Запись —** когда сервер готов писать в канал. Представлен SelectionKey.OP_WRITE

Примеры:

Пример

ИТМО

```
Selector selector = Selector.open();
ServerSocketChannel server = ServerSocketChannel.open();
server.configureBlocking(false);
server.register(selector, SelectionKey.OP_ACCEPT);
while(true) {
    selector.select();
    Set<SelectionKey> keys = selector.selectedKeys();
    for (var iter = keys.iterator(); iter.hasNext(); ) {
        SelectionKey key = iter.next(); iter.remove();
        if (key.isValid()) {
            if (key.isAcceptable()) { doAccept(); }
            if (key.isReadable()) { doRead(); }
            if (key.isWritable()) { doWrite(); }
        }
    }
}
selector.close();
```

Java

60

accept, read, write

ИТМО

```
doAccept() {
  var ssc = (ServerSocketChannel) key.channel();
  var sc = ssc.accept();
  key.attach(clientData);
  sc.configureBlocking(false);
  sc.register(key.selector(), OP_READ);
}
```

```
doRead() {
  var sc = (SocketChannel) key.channel();
  var data = (ClientData) key.attachment();
  sc.read(data.buffer);
  sc.register(key.selector(), OP_WRITE);
}
```

```
doWrite() {
  var sc = (SocketChannel) key.channel();
  var data = (ClientData) key.attachment();
  sc.write(data.buffer);
}
```

61

```
package server.network;
import common.network.Request;
import common.network.Serializer;
import common.utility.AppLogger;
import common.utility.ExecutionResponse;
import server.managers.CollectionManager;
import server.managers.CommandManager;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Scanner;

/**
 * Класс, который принимает подключение от клиента,
 * читает, обрабатывает полученные запросы и отправляет ответы клиенту
 */
public class NetworkServer {
    private final InetSocketAddress address;
    private Selector selector;
```

```
    private final CommandManager commandManager;
    private final AppLogger logger = new AppLogger(NetworkServer.class);
    private final RequestParser requestParser;
    private final CollectionManager collectionManager;

    /**
     * Конструктор класса NetworkServer
     * @param address адрес (порт), на котором будет запущен сервер
     */
    public NetworkServer(InetSocketAddress address) {
        this.address = address;
        this.commandManager = CommandManager.getCommandManagerInstance
        this.requestParser = RequestParser.getRequestParserInstance();
        this.collectionManager = CollectionManager.getCollectionManagerInstance(
    }

    /**
     * Метод, который инициализирует сервер, создаёт канал,
     * регистрируя его в селекторе
     * @return true, если сервер инициализирован, иначе false
     */
    public boolean init() {
        try {
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.oper
            serverSocketChannel.bind(address);
            //канал в неблокирующем режиме
            serverSocketChannel.configureBlocking(false);
            //при помощи статического метода создаём селектор
            selector = Selector.open();
            //регистрируем канал с помощью селектора
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
            logger.info("Сервер инициализирован, ждём подключения клиента ..."
            return true;
        } catch (IOException e) {
            logger.error("Ошибка при инициализации сервера");
            return false;
```

```
    }
}

/**
 * Метод, который запускает сервер и ожидает подключения клиента
 */
public void start() {
    try {
        while (true) {
            //метод блокируется до тех пор, пока хотя бы один канал
            //не будет готов к операции
            selector.select(); //ждёт событий на зарегистрированных каналах

            //итератор по множеству из ключей(Set), которые представляют
            //каналы с готовыми событиями
            Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
            while (keys.hasNext()) {
                SelectionKey key = keys.next();
                keys.remove(); //удаляем ключ из набора обработанных событий
                //проверка на действительный ключ(можно убрать)
                if (key.isAcceptable()) {
                    acceptClient(key); //обрабатываем новое подключение
                } else if (key.isReadable()) {
                    handleClientRequest(key); //читаем данные от клиента,
                    //выполняем команду и отправляем ответ от сервера
                }
            }
        }
    } catch (IOException e) {
        logger.error("Ошибка в обработке подключений");
    }
}

/**
 * Метод, который принимает подключение клиента
 * @param key объект, полученный из SelectionKey
```

```
     * и содержащий данные для регистрации канала
     */
    private void acceptClient(SelectionKey key) throws IOException {
        var serverSocketChannel = (ServerSocketChannel) key.channel();
        var client = serverSocketChannel.accept();
        client.configureBlocking(false);
        client.register(selector, SelectionKey.OP_READ);
        logger.info("Клиент подключился");
    }

    /**
     * Метод, который обрабатывает запрос клиента и
     * отправляет ответ клиенту через канал
     * @param key объект, полученный из SelectionKey и
     * содержащий данные для регистрации канала
     */
    private void handleClientRequest(SelectionKey key) throws IOException {
        var client = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(2000);
        int bytesRead = client.read(buffer);
        if (bytesRead == -1) {
            logger.info("Клиент отключился");
            client.close();
            key.cancel(); //отмена регистрации канала
        }
        buffer.flip();
        byte[] data = new byte[buffer.remaining()];
        buffer.get(data);
        //десериализация полученного запроса от клиента
        Request receivedRequest = Serializer.getInstance().deserialize(data, Reques
        if (receivedRequest == null) {
            logger.error("Сервер получил некорректные данные!");
        }
        String requestFromClient = receivedRequest.getCommandName().toString()
        if (receivedRequest.getCommandObjectArg() != null) {
            requestFromClient += " " + receivedRequest.getCommandObjectArg().toS
```

```
            }
            var command = commandManager.getCommands().get(receivedRequest.ge
            if (command == null) {
                sendResponse(client, new ExecutionResponse("Команда не найдена: " +
            }
            String[] commandAndArgs = requestParser.parseCommand(receivedReque
            ExecutionResponse response = command.execute(commandAndArgs);
            sendResponse(client, response);
        }

        /**
         * Метод, который отправляет ответ клиенту
         * @param client канал, в который отправляется ответ
         * @param response ответ от сервера
         */
        private void sendResponse(SocketChannel client, ExecutionResponse respons
            byte[] responseData = Serializer.getInstance().serialize(response);
            ByteBuffer lengthBuffer = ByteBuffer.allocate(4);
            lengthBuffer.putInt(responseData.length);
            lengthBuffer.flip();
            //блокирует поток до тех пор, пока данные не будут записаны в канал
            client.write(lengthBuffer);
            ByteBuffer buffer = ByteBuffer.wrap(responseData);
            client.write(buffer);
            logger.info("Ответ отправлен клиенту :)");
        }
    }
```

Классы SocketChannel и DatagramChannel.

Эти 2 класса принадлежат пакету **Java-NIO**

SocketChannel представляет собой канал для работы с потоковыми соединениями, использующими протокол TCP. Используется на **клиентской стороне** или на **сервере** для **обработки** уже установленных соединений. Про использование на сервере: когда сервер принимает соединение от

клиента, ServerSocketChannel на серверной стороне создает соединение с клиентом и возвращает объект SocketChannel, который используется для обмена данными.
На сервере
SocketChannel используется для обработки уже установленных соединений. Это может быть чтение данных от клиента и отправка данных клиенту.

Также есть **ServerSocketChannel.** Это это канал для работы с серверными сокетами, использующими протокол TCP. Он предоставляет возможность для создания серверов, которые могут принимать входящие соединения от клиентов (он слушает указанный порт и ожидает, когда клиент подключится).

Пример обмена по протоколу TCP (NIO)

ИТМО

```
// клиент

byte arr[] = {0,1,2,3,4,5,6,7,8,9};
int len = arr.length;
InetAddress host; int port;
SocketAddress addr; SocketChannel sock;

port = 6789;
addr = new InetSocketAddress(host,port);
sock = SocketChannel.open();
sock.connect(addr);

buf = ByteBuffer.wrap(arr);
sock.write(buf);

buf.clear();
sock.read(buf);

for (byte j : arr) {
    System.out.println(j);
}
```

```
// сервер

byte arr[] = new byte[10];
int len = arr.length;
InetAddress host; int port = 6789;
SocketAddress addr; SocketChannel sock;
ServerSocketChannel serv;

serv = ServerSocketChannel.open();
serv.bind(port);
sock = serv.accept();

buf = ByteBuffer.wrap(arr);
sock.read(buf);

for (int j = 0; j < len; j++) {
    arr[j] *= 2;
}

buf.flip();
sock.write(buf);
```

DatagramChannel предназначен для работы с датаграммами, используя протокол UDP. Это означает, что доставка данных не гарантирована, но обеспечивается высокая производительность и низкая задержка.

Пример обмена по протоколу UDP (NIO)

ИТМО

```java
// клиент

byte arr[] = {0,1,2,3,4,5,6,7,8,9};
int len = b.length;
DatagramChannel dc; ByteBuffer buf;
InetAddress host; int port;
SocketAddress addr;

addr = new InetSocketAddress(host,port);
dc = DatagramChannel.open();

buf = ByteBuffer.wrap(arr);
dc.send(buf, addr);

buf.clear();
addr = dc.receive(buf);

for (byte j : arr) {
    System.out.println(j);
}
```

```java
// сервер

byte arr[] = new byte[10];
int len = arr.length;
DatagramChannel dc; ByteBuffer buf;
InetAddress host; int port = 6789;
SocketAddress addr;

addr = new InetSocketAddress(port);
dc = DatagramChannel.open();
dc.bind(addr);

buf = ByteBuffer.wrap(arr);
addr = dc.receive(buf);

for (int j = 0; j < len; j++) {
    arr[j] *= 2;
}

buf.flip();
dc.send(buf, addr);
```

Оба класса поддерживают схожие методы для базовой работы с каналами, такими как открытие, закрытие и подключение.

Общие методы:

1. **open()** открывает новый канал.

- Для **SocketChannel** открывает канал для работы с TCP-соединениями.
- Для **DatagramChannel** открывает канал для работы с UDP-сообщениями.

2. **close()** закрывает канал, освобождая все связанные с ним ресурсы.

3. **bind(SocketAddress local)** привязывает канал к локальному адресу. Это необходимо для прослушивания входящих соединений или получения сообщений(UDP) **(сервер)**

4. **read(ByteBuffer dst)** читает данные из канала в переданный буфер.

5. **write(ByteBuffer src)** записывает данные из переданного буфера в канал.

- Для **SocketChannel** отправляет данные через установленное соединение (TCP).
- Для **DatagramChannel** отправляет данные через канала без установления соединения (UDP).

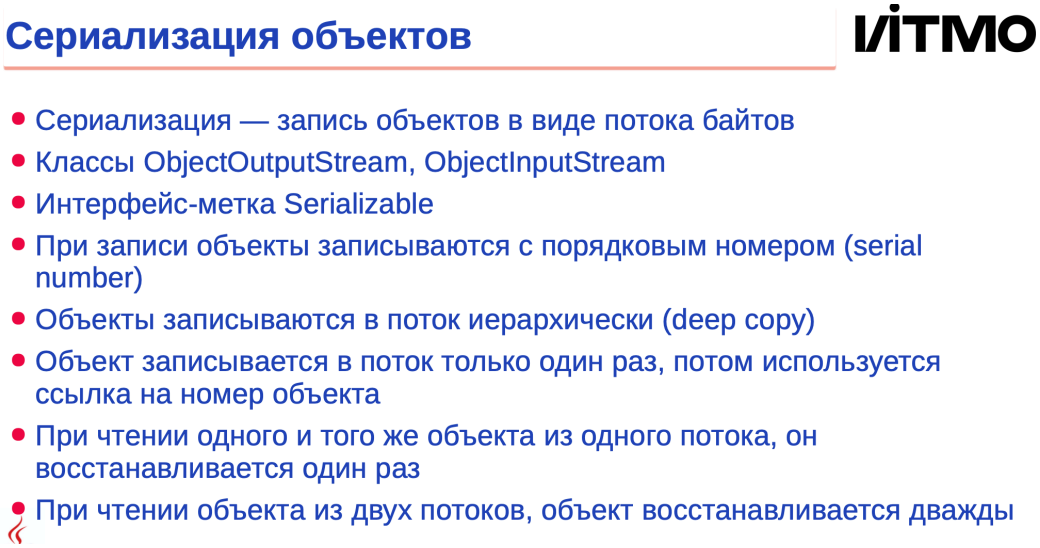
6. **connect(SocketAddress remote)** устанавливает соединение с удаленным адресом **(клиент)**

- Для **SocketChannel** подключает канал к удаленному серверу (TCP).
- Для **DatagramChannel** устанавливает соединение с удаленным адресом (UDP), но это не означает обязательную установку соединения, как в TCP.

Методы, которые есть только у DatagramChannel:

7. **disconnect()** отключает канал от удаленного адреса
8. **receive(ByteBuffer dst)** получает датаграмму и записывает данные в буфер
9. **send(ByteBuffer src, SocketAddress target)** отправляет датаграмму на удаленный адрес

Передача данных по сети. Сериализация объектов.



Сериализация — процесс, при котором данные объекта представляются в виде последовательности байтов. Она используется для того, чтобы обеспечить передачу сложных объектов (например, пользовательских классов) через каналы связи, такие как сокеты.

Обратный процесс, т.е. преобразование последовательности байтов в объект, называется **десериализацией**.

Стандартные сериализация и десериализация

1. **Сериализация объекта:**

Объект, который нужно передать, преобразуется в последовательность байтов с помощью **ObjectOutputStream.** Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись: `ObjectOutputStream(OutputStream out)`

Основные методы:

- **void close()**: закрывает поток
- **void flush()**: очищает буфер и сбрасывает его содержимое в выходной поток
- **void write(byte[] buf)**: записывает в поток массив байтов
- **void write(int val)**: записывает в поток один младший байт из val
- **void writeBoolean(boolean val)**: записывает в поток значение boolean
- **void writeByte(int val)**: записывает в поток один младший байт из val
- **void writeChar(int val)**: записывает в поток значение типа char, представленное целочисленным значением
- **void writeDouble(double val)**: записывает в поток значение типа double
- **void writeFloat(float val)**: записывает в поток значение типа float
- **void writeInt(int val)**: записывает целочисленное значение int
- **void writeLong(long val)**: записывает значение типа long
- **void writeShort(int val)**: записывает значение типа short
- **void writeUTF(String str)**: записывает в поток строку в кодировке UTF-8
- **void writeObject(Object obj)**: записывает в поток отдельный объект

Пример:

```
import java.io.*;
public class Program {
    public static void main(String[] args) {
        //сериализация
        try(ObjectOutputStream oos = new ObjectOutputStream
        (new FileOutputStream("person.dat")))
        {
            Person p = new Person("Sam", 33, 178, true);
            oos.writeObject(p);
        }
```

```
        catch(Exception ex){

            System.out.println(ex.getMessage());
        }
    }
}
class Person implements Serializable{
    private String name;
    private int age;
    private double height;
    private boolean married;
    Person(String n, int a, double h, boolean m){
        name=n;
        age=a;
        height=h;
        married=m;
    }
    String getName() {return name;}
    int getAge(){ return age;}
    double getHeight(){return height;}
    boolean getMarried(){return married;}
}
```

2. **Передача данных:**

   Поток байтов может быть передан через сеть (например, через сокет) или записан в файл

3. **Десериализация объекта:**

   Поток байтов восстанавливается в объект с помощью **ObjectInputStream.**

   В конструкторе он принимает ссылку на поток ввода:

   `ObjectInputStream(InputStream in)`

   Основные методы:

- **void close()**: закрывает поток
- **int skipBytes(int len)**: пропускает при чтении несколько байт, количество которых равно len
- **int available()**: возвращает количество байт, доступных для чтения
- **int read()**: считывает из потока один байт и возвращает его целочисленное представление
- **boolean readBoolean()**: считывает из потока одно значение boolean
- **byte readByte()**: считывает из потока один байт
- **char readChar()**: считывает из потока один символ char
- **double readDouble()**: считывает значение типа double
- **float readFloat()**: считывает из потока значение типа float
- **int readInt()**: считывает целочисленное значение int
- **long readLong()**: считывает значение типа long
- **short readShort()**: считывает значение типа short
- **String readUTF()**: считывает строку в кодировке UTF-8
- **Object readObject()**: считывает из потока объект

Пример:

```
import java.io.*;
public class Program {
   public static void main(String[] args) {
      try(ObjectInputStream ois = new ObjectInputStream
      (new FileInputStream("person.dat")))
      {
         Person p=(Person)ois.readObject();
         System.out.printf("Name: %s \t Age: %d \n",
         p.getName(), p.getAge());
      }
      catch(Exception ex){
         System.out.println(ex.getMessage());
      }
```

```
    }
}
```

Интерфейс Serializable. Объектный граф, сериализация и десериализация полей и методов.

Serializable

1. **Маркерный интерфейс (не содержит методов)**

   Класс реализует интерфейс **Serializable** для того, чтобы его объекты могли быть сериализованы

2. **Что сериализуется:**

   - **Значения всех нестатических полей** (кроме `transient`)
   - **Метаданные класса**, включая:
     - Название класса
     - `serialVersionUID` (идентификатор версии сериализации)
     - Описание полей (их типы и названия)
     - Информация о суперклассе (если он сериализуемый)

3. **Что не сериализуется:**

   - Статические поля не сериализуются, потому что они не привязаны к экземплярам объектов
   - Значения полей с **модификатором transient** не сериализуются

Проблемы стандартной сериализации

ИТМО

- Сериализованный объект зависит от внутренней структуры класса
- Объекты могут создаваться в обход конструкторов
- Потенциальные проблемы совместимости версий
- Возможные проблемы с:
  - продолжительностью сериализации
  - необходимой памятью на сериализацию
  - необходимой глубиной стека

Замена стандартной сериализации

ИТМО

- Задать несериализуемым полям модификатор transient
- Реализовать методы в сериализуемом классе
  - private void writeObject(ObjectOutputStream os)
  - private void readObject(ObjectInputStream is)
- внутри этих методов вызываются методы
  - os.defaultWriteObject()
  - is.defaultReadObject()
- Суперкласс не трогаем

Замена стандартной сериализации

ИТМО

- Реализуем интерфейс Externalizable
- реализуем методы (полный контроль сериализации)
  - writeExternal(ObjectOutput o)
  - readExternal(ObjectInput i)
- Необходимо самостоятельно обрабатывать суперкласс
- При сериализации вместо стандартного механизма будет вызван метод writeExternal, при десериализации - readExternal

Если нужно изменить стандартное поведение сериализации, класс может определить методы

- **private void writeObject(ObjectOutputStream out) throws IOException -** вызывается при сериализации объекта, позволяет записать в поток данные
- **private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException -** вызывается при десериализации объекта, позволяет восстановить поля объекта после чтения данных из потока

Также внутри этих методов можно вызвать методы **defaultWriteObject()**, чтобы использовать стандартную сериализацию для остальных полей и **defaultReadObject()**, чтобы стандартно восстановить остальные поля объекта, после чего можно добавить свою логику восстановления других данных.

Externalizable

Интерфейс, который даёт полный контроль над процессом сериализации и десериализации объекта.

Вместо автоматического сохранения всех полей, объект сам определяет, что именно и как сохранять и восстанавливать.

Когда класс реализует **Externalizable**, он сам реализует два метода:

- **writeExternal(ObjectOutput out)** — записывает нужные данные в поток;
- **readExternal(ObjectInput in)** — считывает данные из потока.

Это даёт возможность:

- Сохранять только необходимые поля
- Применять шифрование или другие методы защиты данных

serialVersionUID

Это поле, используемое в сериализуемых классах, уникальный идентификатор для каждого класса, который используется в процессе десериализации для проверки того, что загруженный класс и оригинальный класс в точности совпадают.

https://itsobes.com/ru/java/zachem-ispolzuetsia-serial-version-uid-chto-esli-ne-opredelit-ego/

Объектный граф:

- Когда сериализуется объект, все его **ссылки на другие объекты** также будут сериализованы. Это называется **объектным графом**. Все связанные объекты будут сериализованы
- Если объект ссылается на другой, то **ссылка** будет сохранена, а не сам объект, таким образом предотвращается дублирование

```
import java.io.*;
public class Address implements Serializable {
    String city;
    public Address(String city) {
        this.city = city;
    }
}
public class Person implements Serializable {
    String name;
    int age;
    Address address; //ссылка на объект другого класса (тоже сериализуется)
    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

Java Stream API. Создание конвейеров. Промежуточные и терминальные операции.

Пакет java.util.stream

- Конвейерная обработка данных
- Поток — последовательность элементов

- Поток может быть последовательным или параллельным
- Конвейер — последовательность операций

Благодаря стримам больше не нужно писать стереотипный код каждый раз, когда приходится что-то делать с данными: сортировать, фильтровать, преобразовывать.

Еще несколько преимуществ стримов:

- Поддержка слабой связанности. Чем меньше классы знают друг про друга, тем лучше.
- Распараллеливать проведений операций с коллекциями стало проще. Там, где раньше пришлось бы проходить циклом, стримы значительно сокращают количество кода.
- Методы не изменяют исходные коллекции, уменьшая количество побочных эффектов.

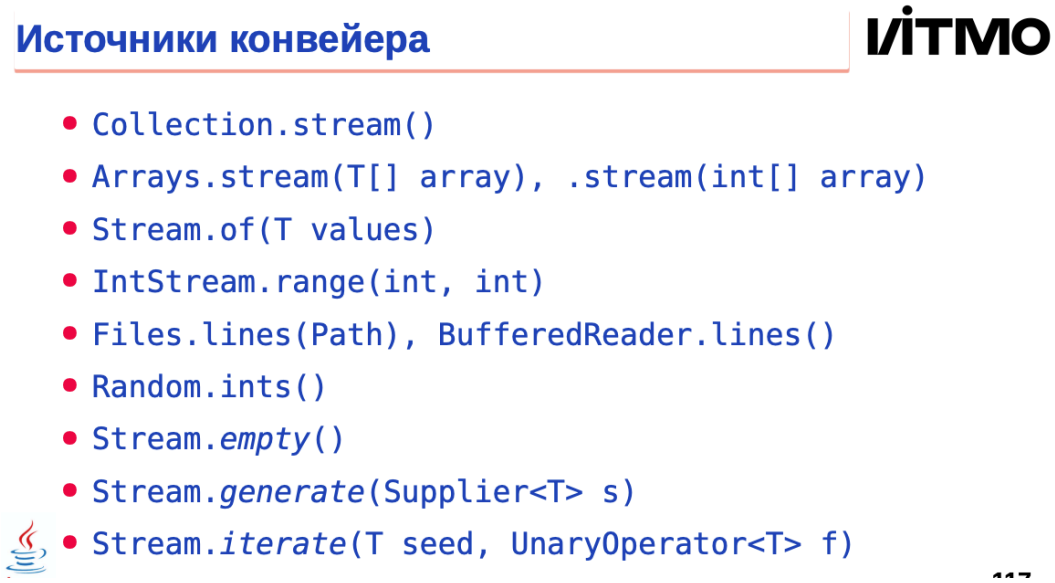
  Stream API

Состав конвейера

Конвейер Stream API состоит из трех обязательных частей:

1. **Источник** (источник данных)
2. **Промежуточные операции** (0 или более)
3. **Терминальная (завершающая) операция** (ровно одна)

Источники конвейера



Промежуточные операции

Характеристики:

- Всегда возвращают новый поток
- Выполняются "лениво" (только при вызове терминальной операции)
- Не изменяют исходные данные

Типы промежуточных операций:

Stateless (не хранящие состояние):

- Не зависят от других элементов потока
- Примеры: `filter()`, `map()`, `flatMap()`, `peek()`

Stateful (хранящие состояние):

- Требуют знания о других элементах
- Примеры: `distinct()`, `sorted()`, `limit()`, `skip()`

Терминальные операции

Характеристики:

- Завершают конвейер
- Возвращают результат или имеют побочное действие

- После выполнения поток нельзя использовать повторно

Типы терминальных операций:

Агрегация: `reduce()`, `collect()`, `count()`

Поиск: `findFirst()`, `findAny()`

Проверка: `anyMatch()`, `allMatch()`, `noneMatch()`

Итерация: `forEach()`, `forEachOrdered()`

Получение результатов: `toArray()`, `min()`, `max()`

Отличная статья на тему Stream API: https://struchkov.dev/blog/ru/java-stream-api/

Примеры:

```
public void removeByIdFromCollection(Long id) {
    organizationCollection.stream() //источник (создаём поток из коллекции)
    //промежуточная операция (фильтрация по id)
    .filter(organization → Objects.equals(organization.getId(), id))

    //превращает поток в Optional<Organization>
    .findFirst() //терминальная операция

    //выполняет переданное действие, если значение в Optional присутствует
    .ifPresent(organizationCollection::remove); //терминальная операция
}


public class Example {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java", "OPD", "Database");
        //преобразуем List в Stream<String>
        List<Integer> lengths = words.stream() //источник

                //преобразует Stream<String> в Stream<Integer>
                //используем лямбду функцию
                //можно написать ссылку на метод
```

```
            //map(String::length)
            .map(s -> s.length()) //промежуточная операциия


            //преобразует Stream<Integer> обратно в List<Integer>
            //Collectors.toList() — стандартный коллектор для сбора элементов
            .collect(Collectors.toList()); //терминальная операция
        System.out.println(lengths);
    }
}
```