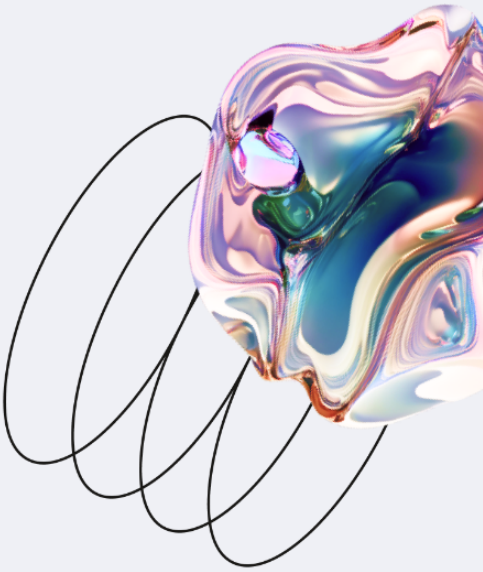
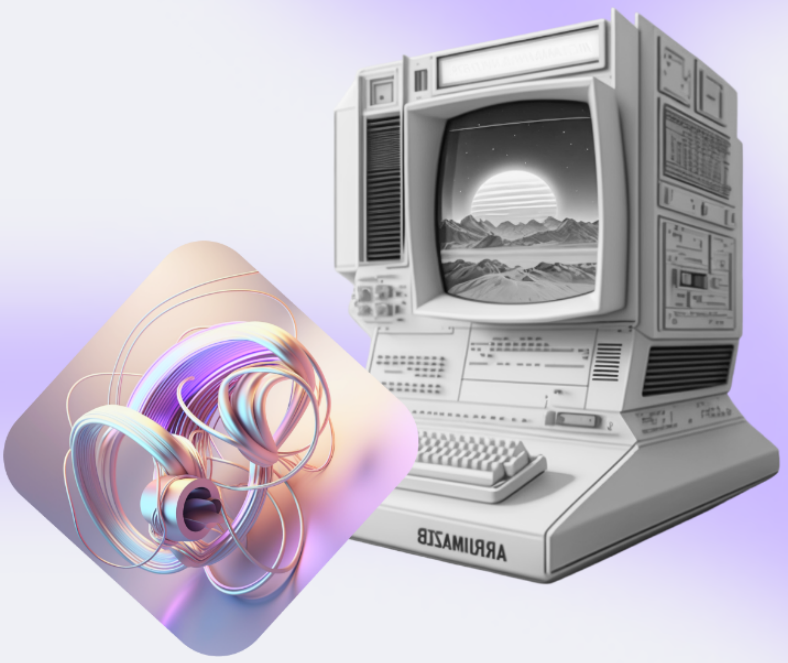
GeekBrains



# Клиент/Сервер своими руками

## Java Junior



# Оглавление

# Введение

Добро пожаловать на пятое занятие! Сегодня мы перейдем к увлекательной теме работы с сетью, и рассмотрим библиотеки языка Java, предназначенные для этой цели. Но, как всегда, начнем с небольшой рефлексии!

На предыдущем занятии мы погрузились в мир баз данных. Разобрались с универсальным подключением через JDBC, обсудили его плюсы и минусы. Познакомились с концепцией ORM и библиотекой Hibernate, ставшей стандартом в этой области. Это знание станет непременным инструментом в вашей работе с базами данных. Теперь, освоив азы работы с серверами баз данных, мы перейдем к вопросу о работе в сети.

На сегодняшней лекции мы рассмотрим и попрактикуем весь цикл работы с сетью. Начнем с создания клиентского приложения и сокета, затем перейдем к серверному приложению и серверному сокету, рассмотрим их особенности и отличия. Мы создадим канал связи и научимся передавать данные по этому каналу. В процессе усложним сервер, превратив его в эхо-сервер, и попробуем его в действии. Однако основной акцент сегодняшней лекции будет сделан на создании полноценного текстового чат-сервера и клиентов, способных подключаться к нему и обмениваться сообщениями. По окончании лекции вы освоите принципы построения сетей, механизмы передачи и маршрутизации данных на прикладном уровне, а также научитесь создавать свои собственные клиент-серверные приложения!

Давайте погружаться в мир сетевого взаимодействия и разрабатывать функциональные приложения вместе!

# Работа в сети

Сегодня сеть, это фактически обязательная составляющая работы с компьютером. Даже когда мы просто пишем текст или, тем более, рисуем в графическом редакторе. Мы ждём авто проверку, доступ к графическим ресурсам, примерам, онлайн помощь и многое другое. А значит, практически любая программа должна уметь работать с сетью. Инфраструктура сети уже устоялась и изменений в ней не предвидится, а в Java имеется пакет java.net специально для работы с сетью! Первый инструмент, это socket что дословно переводится как розетка. Которой он собственно и является. С помощью сокета мы можем подключиться к другому компьютеру, зная его адрес и порт. Вся современная сеть использует IP(Internet Protocol), это маршрутизируемый протокол позволяющий объединить отдельные компьютеры во всемирную сеть за счёт индивидуального идентификатора. Адрес, требуемый сокету, это и есть ip-адрес компьютера, к которому вы хотите подключиться. Существуют также сервера доменных имён(dns), позволяющие обращаться к компьютеру не по ip-адресам а по именам, yandex.ru или gb.ru и у нас есть возможность использовать и такую возможность. То есть один адрес – один socket. А как же тогда к одному серверу подключается большое количество пользователей? За это отвечают порты! Это механизм позволяющий по одному адресу обращаться, как бы в разные квартиры.  Ну а если серьёзней, этот механизм позволяет серверам слушать разные порты в ожидании запросов на связь конкретных сервисов. Например, порт для связи с сервером MySQL или порт для HTTP и так далее и тому подобное. Всего портов на один ip-адрес может быть 65536, номера с 0 по 1023 зарезервированы системой, а остальные используются для реализации гибкой адресации. Именно по этому, для подключения к компьютеру в сети требуется адрес, и порт которые слушает сервер. Что же, давайте уже попробуем, как это всё работает. Для начала нужно создать сокет. И тут у нас несколько возможностей, а вернее несколько конструкторов сокета. Рассмотрим каждый из них.

Первый:
Socket()
Это не подключенный сокет для приложений реалтзующих алгоритм создания подключения самостоятельно. На данный момент сильно выходит за рамки нашего курса, но знать о такой возможности нужно!
Второй:
Socket(String host, int port)

Это классический конструктор, в нём два параметра. Первый это ip-адрес в виде строки а второй это номер порта. Хоть порт и ограничен 16ю битами параметр интовый. На Java сама проверит корректность порта.

Третий:
Socket(InetAddress address, int port)
Этот конструктор тоже можно назвать классическим, отличие заключается только в том, что первый параметр это не строка а экземпляр класса InetAddress. Этот класс позволяет работать с узлами сети по их доменному имени.

Четвертый:
Socket(String host, int port, InetAddress localAddr, int localPort)
Отличие этого конструктора от второго заключается в том, что в третьем и четвёртом параметрах мы передаём свой адрес и порт. Сервер, имея свой ip-адрес, слушает один порт в ожидании запроса на связь. Как только запрос приходит, клиент подключается, и порт фактически занят.

Получается,  для того чтобы сервер мог обеспечить возможность подключения нескольких клиентов по одному порту понадобится специальный механизм. И такой механизм есть, получая запрос на подключение сервер это подключение производит не по тому порту который слушает, а по одному из свободных портов, или порту указанному в этом конструкторе четвёртым параметром.

Пятый:
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
А этот конструктор отличается от предидущего только тем, что вместо строки адреса используется экземпляр класса InetAddress.

Ну вот, конструкторы рассмотрели, можно создавать сокет. Создадим класс Client и в main кинем такой код (Пример 1.1)
Пример 1.1

```
try {
   InetAddress address = InetAddress.getLocalHost();
   Socket client = new Socket(address, 1300);
} catch (UnknownHostException e) {
   throw new RuntimeException(e);
} catch (IOException e) {
```

```
    throw new RuntimeException(e);
}
```

Для примера я воспользовался конструктором требующем InetAddress. Сам по себе он не требует конструктора, так как он возвращает не новый ip-адрес а сервера адрес существующего. Для этого служит статический метод getLocalHost() возвращающий локальный адрес. Кроме этого в классе есть метод getByName(String host) возвращающий адрес по имени. Именно по этому требуется обработать исключение UnknownHostException ибо указанный в host может просто отсутствовать. Само создание сокета может вызвать исключение IOException так как может возникнуть ошибка ввода вывода. Хорошо, в коде разобрались, давайте запустим.

```
Exception in thread "main" java.lang.RuntimeException: java.net.ConnectException: Connection refused: connect
        at socket.Client.main(Client.java:19)
```

Выскочило исключение, и говорит оно о том, что соединение отклонено. И это верно, ведь нет никакого сервера на нашем локальном компьютере слушающий порт 1300. А мы просимся именно туда. А значит нам нужно сделать сервер. За дело! Создадим класс Server.java и в его мэйн кидаем такой код. (Пример 1.2)

Пример 1.2

```
try {
    ServerSocket serverSocket = new ServerSocket(1300);
    Socket socket = serverSocket.accept();
} catch (UnknownHostException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Серверный сокет описан классом ServerSocket и тоже имеет несколько конструкторов, но я использовал самый важный и интересный из них. В нём указан порт, который слушает сервер находящийся по ip-адресу локальной машины. В следующеё строке мы вызываем метод accept() и фактически начинаем слушать

порт. Работа программы приостонавливается пока не произойдёт соединение. Так как во время создания связи может разорваться соединение или произойти другая ошибка ввода вывода, требуется и здесь обработать исключения UnknownHostException и IOException . Давайте вернёмся в класс client и добавим сразу после создания сокета ещё две строки. (Пример 1.3)

Пример 1.3

```
System.out.println(client.getInetAddress());
System.out.println(client.getLocalPort());
```

И запустим проект. Но сначала запустим класс Server, чтобы наш сервер слушал порт, а потом client. И посмотрим, что получится.

```
DESKTOP-ECGAV6S/192.168.1.37
55664
```

Отлично! Связь есть! Метод getInetAddress() возвращает имя компьютера и адрес в сети, а метод getLocalPart() соответственно возвращает порт для нашего конкретного подключения. Сервер, в данный момент может продолжать слушать порт 1300, и, в необходимости, принимать новые подключения. Теперь мы можем обмениваться данными с сервером и для этого мы можем получить поток ввода и поток вывода. Но работать с ними не так удобно, ибо они оперируют только с байтами и массивами байт, поэтому сразу создадим более удобные и гибкие объекты. Добавим в класс client.java ещё несколько строк. (Пример 1.4)

Пример 1.4

```
InputStream inStream = client.getInputStream();
OutputStream outStream = client.getOutputStream();
DataInputStream dataInputStream = new DataInputStream(inStream);
PrintStream printStream = new PrintStream(outStream);
```

Первые две строки, это собственно потоки. Метод getInputStream() возвращает поток ввода клиентского сокета а getOutputStream() соответственно поток вывода. Класс DataInputStream позволяет читать из потока любые примитивные типы данных, массивы и строки! Это сильно удобнее чем оперировать байтами, поэтому мы его и используем. Класс PrintStream добавляет функционал потоку, фактически позволяя нам просто печатать туда строки как будто это обычный print. Этими возможностями мы обязательно воспользуемся, и для этого допишем в класс client.java ещё пару строк. (Пример 1.5)

Пример 1.5

```
printStream.println("Привет!");
System.out.println(dataInputStream.readLine());
client.close();
```

Первая трока, как раз и показывает возможности класса PrintStream, передача строки по сети свелось к вызову классического принта с переносом строки ( println() ). Вторая строка использует возможности класса DataInputStream чтобы получить по сети строку и сразу вывести её в консоль. Ну и в третьей строке мы закрываем сокет. Делать это необходимо, хотя бы, чтобы освободить ресурсы сервера, если они вам больше не нужны. Откроем наш класс Server.java и допишем его функционал. (Пример 1.6)

Пример 1.6

```
OutputStream outStream = socket.getOutputStream();
PrintStream printStream = new PrintStream(outStream);
printStream.println("Hello!");

socket.close();
serverSocket.close();
```

Из неизвестного у нас только закрытие серверного сокета. Нужно это если вы получили от клиента всё, чего ожидали и в дальнейшем слушать порт не собираетесь. Теперь оба наши класса готовы для повторной проверки. Запустим, как и в прошлый раз, сначала Server а потом Client.

```
DESKTOP-ECGAV6S/192.168.1.37
60095
Hello!
```

Всё отлично! Мы снова видим свой адрес и порт, причём порт другой, но это нормально. А ещё видим Hello! Строка, отправленная сервером после подключения к нему обратно на сторону клиента и только тогда уже выведенная в консоль.

## Эхо!

А давайте нашими разработками воспользуемся и создадим полноценный эхо сервер! Отличаться он будет тем, что не станет закрываться сразу, а продолжит отвечать на запросы.

Но вот реализовать это будет несколько сложнее чем, кажется. Если рассмотреть код нашего сервера (Пример 1.6), то мы увидим конкретную последовательность, сначала ждём связи, потом настраиваем канал вывода, отправляем сообщение и связь закрываем. Можно добавить канал вводи и ожидание сообщения, но у нас возникает серьёзная проблема. Ожидание входящего сообщения, это цикл. И Это вариант, но дальнейшее расширение функционала будет затруднительно потому, что пока мы не выйдем из тела цикла мы не выполним никаких других действий. И тут нам на помощь приходит много поточность, а точнее её реализация в Java классе Thread. Если не вникать в подробности, то много поточность в яве реализовать очень просто и именно этим мы и воспользуемся. Создадим новый класс ServerReadThread.java тут будет лежать описание работы сервера с входящим

потоком. Так как этот класс долен стать потоком, он должен расширять класс Thread а значит к сигнатуре класса нам нужно добавить extends Thread.
Сама работа в потоке будет завязана на сокете, поэтому в теле класса объявим поле Socket, мы не будем его инициализировать, так как ссылку получим в конструкторе. А значит, у нас будет и параметризованный конструктор, ну и переопределённый метод run() вызов которого и запускает наш поток в работу. Давайте посмотрим, что получилось и рассмотрим, что получается. Пример (2.1)
Пример 2.1

```
class ServerReadThread extends Thread{
   Socket socket;
   public ServerReadThread(Socket socket) {
      this.socket = socket;
      this.run();
   }
   @Override
   public void run() {
      try {
         BufferedReader reader = new BufferedReader(
               new InputStreamReader(socket.getInputStream()));
         String inLine;
         OutputStream outStream = socket.getOutputStream();
         PrintStream printStream = new PrintStream(outStream);
         while ((inLine = reader.readLine()) != null){
            printStream.println("("+socket.getPort()+") "+ inLine);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Сначала про метод run(), в него я положил всё, что касается потоков ввода и вывода. Создаём новою строку, для хранения входящего сообщения. Цикл крутится, пока работает поток, в нашем случае всегда. И в тело падает, только если пришедшее сообщение не null, а значит что-то пришло со стороны клиента. В самом теле передаём методом println класса PrintStream сообщение обратно клиенту. Формат сообщения «( № Порта клиента ) inLine». Класс сокет предоставляет два метода, getLockalPort() и getPort(). Первый возвращает номер порта, который слушает сервер, а второй номер порта, по которому сервер подключился к клиенту. Таким образом можно легко идентифицировать клиентов! Что же, класс сделали теперь

им нужно воспользоваться в классе Server. А вот этот класс нам придётся переписать полностью, даже скорее не переписать а сильно сократить (Пример 2.2)

Пример 2.2

```
public class Server {
   public static void main(String[] args) {
      try{
         ServerSocket serverSocket = new ServerSocket(1300);
         Socket socket = serverSocket.accept();
         ServerReadThread thread = new ServerReadThread(socket);
         while (!thread.isAlive()){}
         socket.close();
         serverSocket.close();
      } catch (UnknownHostException e) {
         throw new RuntimeException(e);
      } catch (IOException e) {
         throw new RuntimeException(e);
      }
   }
}
```

Вот всё, что осталось от класса Server! Создаём серверный сокет, ждём подключение, создаём само запускающийся поток, и просто крутимся в вайле. isAlive вернёт ложь только если пеоток будет закрыт. Ну и закрытие сокетов, как и было раньше. Но и это не всё, что нужно сделать для работы эхо сервера. Ещё нужно, чтобы наш клиент использовал новые возможности! Возвращаемся в класс Client.java и перед закрытием клиентского сокета пишем такой код

Пример 2.3

```
String inLine;
Scanner scanner = new Scanner(System.in);
while (!(inLine = scanner.nextLine()).equals("exit")){
   printStream.println(inLine);
   System.out.println(dataInputStream.readLine());
}
scanner.close();
```

Мы создали строку ввода чтобы хранить в ней строку на передачу и сканер, как самый простой класс предназначенный для ввода текста с консоли. Цикл вайл будет крутиться пока мы не введём ключевое слово «exit». В теле цикла мы просто посылаем серверу строку, а потом ждём ответа. А если ввели «exit» то и сканер и сокет и весь клиент будут закрыты.

DESKTOP-ECGAV6S/192.168.1.37

63640
Hello Server!
(63640) Hello Server!
I am your client!)
(63640) I am your client!)
exit
Process finished with exit code 0
Как видите и сервер, и клиент работают по плану. Мы пишем сообщения со стороны клиента и получаем ответ со стороны сервера!

## Чат сервер

Ну что же, немного размялись, пора переходить к гораздо более серьезному проекту, чат серверу. Давайте, сразу определим, что должны уметь сервер, и клиент. Сервер должен уметь слушать заданный порт в ожидании запроса на подключение, как и раньше, но, и после получения запроса, и после формирования канала связи с клиентом, он должен продолжать слушать порт и быть готовым к новым подключениям. Сервер должен уметь идентифицировать участников чата по имени. Сервер, также, должен уметь оповещать уже подключенных участников чата о том, что к чату подключился новый пользователь и указывать его имя. При отключении участника, сервер должен уметь оповестить оставшихся участников и об этом. Количество участников не должно быть ограничено! Сервер должен уметь транслировать переданное одним участником сообщение всем остальным кроме его написавшего. Получается достаточно много, но это минимальные необходимые требования для чат-сервера. Разберёмся с клиентом, тут всё немного проще. Клиент должен уметь подключаться к серверу и, иметь возможность параллельно принимать и отправлять сообщения. То есть, пока вы пишите свое сообщение, вам может прийти ещё несколько сообщений от других пользователей чата. Зацикливание работы или на вводе или на передаче сообщения сделает чат совсем не адекватным. Так же и сервер и клиент должны уметь корректно закрывать все используемые ресурсы. С требованиями, вроде, разобрались, можно приступать и к работе. Клиент сильно проще сервера, да и зная, как работает сервер, клиент станет писать проще. Поэтому начнём работу с сервера. Создадим новый проект, пусть он называется ChatServer, и первым классом будет Server.java.

```
public class Server {}
```
Что нам понадобится в классе обязательно? Понадобится нам сервер сокет, добавим его сразу.
```
private ServerSocket serverSocket;
```
Ещё понадобится параметризированный конструктор, принимающий сокет.
```
public Server(ServerSocket serverSocket) {this.serverSocket = serverSocket;}
```
Следующим шагом, нужно сделать метод запускающий наш сервер.
```
public void runServer() {}
```
В теле метода мы будем работать с сокетом, а значит нам понадобиться отлавливать возможные исключения, так добавим сразу обёртку try/catch
```
try {} catch (IOException e){}
```

Тело try мы рассмотрим но начнём catch. Попасть сюда мы можем, только если возникнет проблема со связью, а это значит обработка исключения может свестись просто к закрытию сокета. Чтобы не загромождать метод кодом, я напишу в теле catch вызов другого метода, которого пока не существует, но мы допишем его позже.
```
closeSocket();
```
Теперь переходим к телу try. На самом деле весь функционал сервера должен находиться здесь. Но это сильно усложнит класс, поэтому я вынесу большую часть функционала во внешний класс и упрощу тем самым и код, и его понимание и возможности расширения.

Пример 3.1
```
while (!serverSocket.isClosed()){
            Socket socket = serverSocket.accept();
            System.out.println("Подключен новый клиент!");
            ClientManager client = new ClientManager(socket);
            Thread thread = new Thread(client);
            thread.start();
        }
```
Цикл получился похожим на код сервера (Пример 2.2), но тут сервер крутится, пока не закрыт серверный сокет. Первая же строка блокирует выполнение приложения в ожидании запроса на подключение. Сервер слушает порт! Как только запрос получен, формируется канал связи. В объект socet сохраняется вся информация о только что созданном подключении. В консоль выводиться строка: «Подключен новый клиент!», а в следующей строке Мы создаём объект клиента на основе того самого класса, о котором говорил чуть раньше. Класс будет называться ClientManager, возьмёт на себя весь функционал связанный с меж клиентным общением и,

будет выполняться как отдельный поток. Много поточность, это необходимая составляющая, позволяющая серверу слушать всех клиентов и рассылать всем клиентам сообщения. Две следующие строки как раз создают поток и запускают его. Напишем теперь метод closeSocket

Пример 3.2

```
public void closeSocket(){
    try{
      if (serverSocket != null) serverSocket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Причина, по которой я вынес закрытие сокета в отдельный метод, заключается в том, что этот процесс потребует ещё одну обёртку try/catch и условие. Закрыть сокет мы можем только если он не нулевой, иначе получим нулл поинтер, а сам метод close требует обработки исключения IOException. В данном случае при возникновении исключения, просто выводим его стек в консоль.

Осталось дописать точку входа сервера.

Пример 3.3

```
public static void main(String[] args) throws IOException {
    ServerSocket serverSocket = new ServerSocket(1300);
    Server server = new Server(serverSocket);
    server.runServer();
  }
```

Обычный main. Первой строкой объявляем и инициализируем сервер сокет. Порт 1300. Затем создаём объект, экземпляр написанного нами только что класса и передаём ему в параметр конструктора уже готовый сокет. Ну и запускаем сервер! Обработка исключений здесь тоже требуется, но, так как это точка входа, и выше нашего кода нет, мы просто пробросим исключение добавив в сигнатуру метода throws IOEception. Теперь класс ClientManager. Создадим его рядом.

```
public class ClientManager implements Runnable{}
```

Наш новый класс реализует интерфейс Runnable. Это обязательно так как класс, выполняющийся в потоке, должен реализовать метод run объявленный в этом интерфейсе. Так добавим реализацию метода run.

```
  @Override
  public void run() {}
```

Метод пока пустой, мы наполним его чуть позже. Итак, какие поля должен иметь класс описывающий клиента со стороны сервера?

```
    private Socket socket;
    private BufferedReader bufferedReader;
    private BufferedWriter bufferedWriter;
    private String name;
    public static ArrayList<ClientManager> clients = new ArrayList<>();
```

Во первых нам нужен наш канал связи, для этого объявим локальное приватное поле класса Socket. BufferedReader и BufferedWriter. Они нам понадобятся для того, чтобы передавать и принимать данные. Объявим их тоже приватными. Имя клиента, тоже нужное поле и тоже приватное, так как ни к чему засорять им интерфейс нашего класса. А вот следующее поле интереснее. Это публичный список клиентов, и содержит он экземпляра класса ClientManager. Но интересно это поле модификатором static. Привязывающий это поле не к экземпляру класса, а к самому классу. Это сильно упростит работу со списком, фактически, откуда бы мы не работали с этим списком, мы будем обращаться к одному общему списку. Далее нам нужен конструктор.

```
public ClientManager(Socket socket) {}
```

В теле конструктора мы будем работать с сокетами и потоками, а значит нужно будет обрабатывать исключения. Добавим обёртку сразу.

```
        try {} catch (IOException e) {}
```

Сначала рассмотрим тело catch. Если произошло исключение, значит какая-то проблема. Но не со всеми клиентами или сервером, а только с этим клиентом. Поэтому в теле catch пока поставим вызов метода закрывающего все ресурсы, а чуть позже этот метод напишем.

```
        closeEverything(socket, bufferedReader, bufferedWriter);
```

Теперь тело try.

Пример 3.4

```
        this.socket = socket;
                                        bufferedWriter = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));
                                        bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        name = bufferedReader.readLine();
        clients.add(this);
        broadcastMessage("Server: "+name+" подключился к чату.");
```

В первой строке мы инициализируем наш локальный сокет из пришедшего в параметрах. Далее две строки инициализируют buffered Writer и Reader. Тут есть интересная особенность. В яве классы описывающие потоки бывают двух видов. Названия первых заканчиваются на Stream и они работают с байтами, а названия вторых заканчиваются на Writer или Reader и они работают с символами. Так как мы передаём текст, нам удобнее использовать символьные потоки. Но существуют ещё и обёртки. Например, для OutputStreamWriter есть обёртка BufferWriter. Он позволяет буферизировать поток и упростить работу с ним. Поэтому мы и используем обёртки! Следующая строка, это как раз пример удобства обёртки. Мы ждём пока из потока не прейдет строка текста. То есть строка, заканчивающаяся или символом переноса строки или возвратом каретки. Вобщето мы ждём пока клиент введёт своё имя и сохраним его в переменную name. Дальше мы добавляем нового клиента в список. А затем вызываем метод рассылки. Пока этого метода нет, но мы его сейчас напишем!

Пример 3.5

```
  private void broadcastMessage(String massageToSend) {
    for (ClientManager client: clients) {
      try {
        if (!client.name.equals(name)) {
          client.bufferedWriter.write(massageToSend);
          client.bufferedWriter.newLine();
          client.bufferedWriter.flush();
        }
      } catch (IOException e){
        closeEverything(socket, bufferedReader, bufferedWriter);
      }
    }
  }
```

Метод приватный, потому, что он нам нужен только внутри класс. Принимает строку для передачи и в теле у него простой, яковский forEach. Мы перебираем всех клиентов из списка и, если имя клиента не такое же, как у нас, используем bufferWriter клиента для передачи ему строки. Метод write() записывает строку, метод newline() передаёт символ переноса строки а flush() чистит буфер дабы он не переполнялся и не рос. В исключении опять же метод closeEverything(), он закрывает все ресурсы и пора его написать.

Пример 3.6

```
    private void closeEverything(Socket socket, BufferedReader bufferedReader,
BufferedWriter bufferedWriter) {
    removeClient();
```

```
        try {
            if (bufferedReader != null) {
                bufferedReader.close();
            }
            if (bufferedWriter != null) {
                bufferedWriter.close();
            }
            if (socket != null) {
                socket.close();
            }
        } catch (IOException e){
            e.printStackTrace();
        }
    }
```

Задача этого метода, просто закрыть все ресурсы и удалить клиента из списка. Первым делом удаляем текущего клиента, removeClient(), опять метод, который мв ещё не написали. Потом мы просто проверяем что ресурсы инициализированы, то есть не null и, если это так, закрываем их. Во время выполнения методы может быть выброшено исключение IOException, в данном случае мы просто выведем его стек в консоль. Теперь метод удаления клиента.

Пример 3.7

```
public void removeClient(){
        clients.remove(this);
        broadcastMessage("SERVER: "+name+" покинул чат.");
    }
```

Тут всё просто, удаляем клиента из списка и сообщаем всем остальным, что клиент покинул чат. Пора вернуться к методу run(), ведь именно с него начинается работа потока клиента. Первое, что понадобится в этом методе, это локальная строка для передачи в чат.

```
String massageFromClient;
```

Следом напишем цикл, который будет крутиться, пока клиент подключен.

Пример 3.8

```
        while (socket.isConnected()){
            try {
                massageFromClient = bufferedReader.readLine();
                broadcastMessage(massageFromClient);
            } catch (IOException e){
                closeEverything(socket, bufferedReader, bufferedWriter);
```

```
            break;
        }
    }
```

Тут всё довольно просто. Если всё нормально ждём сообщение от клиента и, дождавшись, рассылаем всем остальным клиентам. Если возникло исключение закрываем все ресурсы клиента и останавливаем поток. Мы рассмотрели два класса, Server и ClientManager. Они оба работают на серверной стороне и обеспечивают интерактивное общение клиентов чата. Фактически это и есть основная часть работы сервера, которая была поставлена нами в начале. Но, чтобы с этим сервером работать нужен ещё и клиент. Займёмся написание клиентской части. Класс назовём Client.java и положим его рядом с серверными классами. Это неважно для функционирования, они должны работать на разных компьютерах, но в нашем проекте это сделано для удобства.

```
public class Client {}
```

Класс тоже довольно большой, поэтому пойдём не большими шажками. Для работы со стороны клиента нам понадобится сокет, BufferedReader, BufferedWriter, ну и строка со своим именем.

```
private Socket socket;
private BufferedReader bufferedReader;
private BufferedWriter bufferedWriter;
private String name;
```

Добавили необходимые поля класса, пора сделать ему конструктор.

Пример 3.9

```
public Client(Socket socket, String userName) {
    this.socket = socket;
    name = userName;
    try {
                                    bufferedWriter = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));
                                bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    } catch (IOException e) {
        closeEverything(socket, bufferedReader, bufferedWriter);
    }
}
```

Параметрами конструктор принимает инициализированный сокет, и своё имя. Сохраняем их в локальные переменные. Дальше инициализируем bufferWriter и bufferReader заранее обернув их в try/catch. Если выскочило исключение,

закрываем все ресурсы. Хорошо класс инициализирован, что он должен уметь? Он должен уметь посылать сообщения.
Пример 3.19

```
public void sendMessage(){
    try {
        bufferedWriter.write(name);
        bufferedWriter.newLine();
        bufferedWriter.flush();

        Scanner scanner = new Scanner(System.in);
        while (socket.isConnected()){
            String message = scanner.nextLine();
            bufferedWriter.write(name+": "+ message);
            bufferedWriter.newLine();
            bufferedWriter.flush();
        }
    } catch (IOException e){
        closeEverything(socket, bufferedReader, bufferedWriter);
    }
}
```

Кода в методе довольно много, но он не так сложен. Сначала посылаем серверу своё имя, потом ини циализируем сканер для чтения текста из консоли и, пока есть связь, читаем строку и передаём прочитанное серверу. Передаём с символом переноса строки и чистим буфер передачи.
При работе с сервером может выскочить исключение, в этом случае мы закроем все ресурс. В методе передачи сообщения есть одна хитрость, но я расскажу о ней чуть позже. Ещё клиент должен уметь слушать сообщения.
Пример 3.11

```
public void listenForMessage(){
    new Thread(new Runnable() {
        @Override
        public void run() {
            String messageFromGroup;
            while (socket.isConnected()){
                try {
                    messageFromGroup = bufferedReader.readLine();
                    System.out.println(messageFromGroup);
```

```
            } catch (IOException e){
                closeEverything(socket, bufferedReader, bufferedWriter);
            }
        }
    }
}).start();
}
```

Тут текста казалось бы ещё больше, но метод ещё проще. Так как он должен работать в параллельном потоке, чтобы исполняя свой функционал не отвлекать приложение от передачи сообщений, он должен запустить поток. Это мы и делаем! Создаём анонимный Thread, переопределяем в нём метод run и запускаем( start() ). В потоке мы просто крутимся пока есть коннект и ждём сообщения со стороны сервера, если сообщение пришло, печатаем его в консоль. Если со связью что то произошло , отлавливаем исключение и закрываем все ресурсы клиента. Пришла пора последнего метода клиента, метод closeEverything. По сути он работает так же как и в ClientManager, если ресурс инициализирован, закрываем его. Если возникает исключение выводим его стек в консоль.

Пример 3.12

```
private void closeEverything(Socket socket, BufferedReader bufferedReader, BufferedWriter bufferedWriter) {
    try {
        if (bufferedReader != null) {
            bufferedReader.close();
        }
        if (bufferedWriter != null) {
            bufferedWriter.close();
        }
        if (socket != null) {
            socket.close();
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

Осталась точка входа клиента и мы готовы проверить работу чат сервера. Мэйн будет состоять всего из 7ми строчек кода! Мы объявляем и инициализируем сканер, просим ввести имя пользоватеоя, подключаемся к серверу по порту 1300, объявляем и инициализируем клиента. В параметры конструктора передаём сокет

подключения и имя пользователя. Затем вызываем методы для ожидания и отправки сообщений.

```
public static void main(String[] args) throws IOException {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ввыдите своё имя: ");
    String name = scanner.nextLine();
    Socket socket = new Socket("localhost", 1300);
    Client client = new Client(socket, name);
    client.listenForMessage();
    client.sendMessage();
}
```

Я говорил, что метод sendMessage() имеет особенность, она заключается в том, что этот метод сначала передаёт серверу имя клиента, для его инициализации на стороне сервера, а потом крутится в цикле, пока клиент активен. Это объясняет три повторяющиеся строки кода. Ну что же,

Код вроде бы готов и мы можем попробовать, что из этой нашей работы получится. Запускать я буду отдельно класс Server и отдельно, три раза класс Client. Idea позволяет параллельно запускать один и тот же класс. Этим мы и воспользуемся.

Так выглядит состояние чата при запуске.

Я ввёл имя первого пользователя.

Теперь имя второго.

И имя третьего клиента.

Антон послал в чат сообщение. Сервер ничего не показывает, так как сейчас занят ClienManager, он маршрутизирует пакеты так, чтобы переданная строка пришла только тем кому надо.

Дима ответил, и всё идее именно так, как мы запланировали!

А вот ответил Джимми.

Ответил на русском, и всё прошло нормально, потому, что мы используем не байтовый канал а обёртку вокруг символьного канала!

Дима попрощался и покинул чат!  Сервер корректно сообщил всем оставшимся пользователям информационную строку.

Ну вот, на мой взгляд, сервер работает и работает именно так, как мы запланировали. Было бы интересно, если бы вы смогли самостоятельно добавить, например приоритет сообщений сервер, и команды, позволяющие показывать или нет низко приоритетные сообщения сервера. Было бы интересно, если бы вы смогли добавить сообщение сервера по типу: «Пользователь Dima печатает.». Чат сервер получился довольно большой и на нём я предлагаю закончить сегодняшнюю лекцию.

Сегодня мы рассмотрели сетевое взаимодействие. Попробовали подключаться к компьютерам по сети и передавать пакеты. Делали эхо сервер и чат сервер. Познакомились и потрогали многие классы и методы позволяющие реализовать сетевое взаимодействие и создать клиент/серверное приложение своими руками. Надеюсь было интересно. Жду ваших вопросов. Пока!

Пример 3.1

```
public class Server {
    private ServerSocket serverSocket;
    public Server(ServerSocket serverSocket) {
        this.serverSocket = serverSocket;
    }
    public void runServer() {
        try {
            while (!serverSocket.isClosed()){
                Socket socket = serverSocket.accept();
                System.out.println("Подключен новый клиент!");
                ClientManager client = new ClientManager(socket);
                Thread thread = new Thread(client);
                thread.start();
            }
        } catch (IOException e){
            closeSocket();
        }
    }
    public void closeSocket(){
        try{
            if (serverSocket != null) serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(1300);
        Server server = new Server(serverSocket);
        server.runServer();
    }
}
```

Пример 3.2

```
public class ClientManager implements Runnable{
    private Socket socket;
    private BufferedReader bufferedReader;
    private BufferedWriter bufferedWriter;
    private String name;
    public static ArrayList<ClientManager> clients = new ArrayList<>();

    public ClientManager(Socket socket) {
        try {
            this.socket = socket;
                                                bufferedWriter = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));
                                            bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            name = bufferedReader.readLine();
            clients.add(this);
            broadcastMessage("Server: "+name+" подключился к чату.");
        } catch (IOException e) {
            closeEverything(socket, bufferedReader, bufferedWriter);
        }
    }

    @Override
    public void run() {
        String massageFromClient;

        while (socket.isConnected()){
            try {
                massageFromClient = bufferedReader.readLine();
                broadcastMessage(massageFromClient);
            } catch (IOException e){
                closeEverything(socket, bufferedReader, bufferedWriter);
                break;
            }
        }
    }

    private void broadcastMessage(String massageToSend) {
        for (ClientManager client: clients) {
```

```
            try {
                if (!client.name.equals(name)) {
                    client.bufferedWriter.write(massageToSend);
                    client.bufferedWriter.newLine();
                    client.bufferedWriter.flush();
                }
            } catch (IOException e){
                closeEverything(socket, bufferedReader, bufferedWriter);
            }
        }
    }

    private void closeEverything(Socket socket, BufferedReader bufferedReader,
BufferedWriter bufferedWriter) {
        removeClient();
        try {
            if (bufferedReader != null) {
                bufferedReader.close();
            }
            if (bufferedWriter != null) {
                bufferedWriter.close();
            }
            if (socket != null) {
                socket.close();
            }
        } catch (IOException e){
            e.printStackTrace();
        }
    }

    public void removeClient(){
        clients.remove(this);
        broadcastMessage("SERVER: "+name+" покинул чат.");
    }
}
```

Пример 3.3

```
public class Client {
    private Socket socket;
    private BufferedReader bufferedReader;
```

```
    private BufferedWriter bufferedWriter;
    private String name;

    public Client(Socket socket, String userName) {
        this.socket = socket;
        name = userName;
        try {
                                                bufferedWriter = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));
                                            bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        } catch (IOException e) {
            closeEverything(socket, bufferedReader, bufferedWriter);
        }
    }

    public void sendMessage(){
        try {
            bufferedWriter.write(name);
            bufferedWriter.newLine();
            bufferedWriter.flush();

            Scanner scanner = new Scanner(System.in);
            while (socket.isConnected()){
                String message = scanner.nextLine();
                bufferedWriter.write(name+": "+ message);
                bufferedWriter.newLine();
                bufferedWriter.flush();
            }
        } catch (IOException e){
            closeEverything(socket, bufferedReader, bufferedWriter);
        }
    }

    public void listenForMessage(){
        new Thread(new Runnable() {
            @Override
            public void run() {
                String messageFromGroup;
```

```
            while (socket.isConnected()){
                try {
                    messageFromGroup = bufferedReader.readLine();
                    System.out.println(messageFromGroup);
                } catch (IOException e){
                    closeEverything(socket, bufferedReader, bufferedWriter);
                }
            }
        }
    }).start();
}

    private void closeEverything(Socket socket, BufferedReader bufferedReader,
BufferedWriter bufferedWriter) {
    try {
        if (bufferedReader != null) {
            bufferedReader.close();
        }
        if (bufferedWriter != null) {
            bufferedWriter.close();
        }
        if (socket != null) {
            socket.close();
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ввыдите своё имя: ");
    String name = scanner.nextLine();
    Socket socket = new Socket("localhost", 1300);
    Client client = new Client(socket, name);
    client.listenForMessage();
    client.sendMessage();
}
}
```

## Заключение

Это было пятое и завершающее занятие в рамках курса Java Junior. Я уверен, что вы стали более опытными программистами. В ходе курса вы освоили лямбда-выражения и возможности Stream API, пробовали свои силы в создании новых функциональных интерфейсов и тренировались с функционалом Stream API на семинарах. Вы познакомились с рефлексией и поняли ее значение в программировании, а на практических занятиях смогли создать свой объект и провести его анализ по байтам с использованием Reflection API. Вы научились не только работать с файлами, но и выполнять сериализацию и десериализацию объектов. На семинарах вы убедились, насколько эти процессы более просты и быстры по сравнению с обычной работой с файлами. Мы представили вам базы данных и методы работы с ними напрямую, как с объектами. На практике вы освоили JDBC и Hibernate. Впереди вас ждут новые курсы, и мы уверены, что полученный уровень подготовки поможет вам стать квалифицированными специалистами в области информационных технологий.