ЛАБОРАТОРНАЯ РАБОТА №3.
ПО ПРЕДМЕТУ
ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»
ТЕМА: «РАЗРАБОТКА ПРИЛОЖЕНИЙ В АРХИТЕКТУРЕ КЛИЕНТ-СЕРВЕР ПО ПРОТОКОЛУ TCP»

1-40 05 01 «Информационные системы и технологии (в бизнес-менеджменте и промышленной безопасности)»

Сетевой протокол *TCP/IP* поддерживается в *Java* благодаря имеющимся интерфейсам потокового ввода-вывода, а также средств, необходимых для построения объектов ввода-вывода через сеть. В *Java* поддерживаются оба семейства протоколов – *TCP* и *UDP*.

Протокол *TCP* применяется для надежного потокового ввода-вывода через сеть. А протокол *UDP* поддерживает более простую, а, следовательно, и быструю модель передачи дейтаграмм от одной точки сети к другой. Ниже в таблице перечислены классы, входящие в пакет *java.net.*

Таблица 1 – Основные классы, входящие в пакет *java.net*

Интерфейсы	***Классы***
ContentHandlerFactory	*Authenticator*
CookiePolicy	*CacheRequest*
CookieStore	*CacheResponse*
DatagramSocketImplFactory	*ContentHandler*
FileNameMap	*CookieHandler*
ProtocolFamily	*CookieManager*
SocketImplFactory	*DatagramPacket*
SocketOption <T>	*DatagramSocket*
SocketOptions	*DatagramSocketImpl*
URLStreamHandlerFactory	*HttpCookie*
Перечисления	*HttpURLConnection*
Authenticator.RequestorType	*IDN*
Proxy.Type	*Inet4Address*
StandardProtocolFamily	*Inet6Address*
Исключения	*InetAddress*
BindException	*InetSocketAddress*
ConnectException	*InterfaceAddress*
HttpRetryException	*JarURLConnection*
MalformedURLException	*MulticastSocket*
NoRouteToHostException	*NetPermission*
PortUnreachableException	*NetworkInterface*
ProtocolException	*PasswordAuthentication*
SocketTimeoutException	*Proxy*

UnknownHostException	*ProxySelector*
UnknownServiceException	*ResponseCache*
URISyntaxException	*SecureCacheResponce*
Классы	*ServerSocket*
URLClassLoader	*Socket*
URLConnection	*SocketAddress*
URLDecoder	*SocketImpl*
URLEncoder	*SocketPermission*
URLPermission	*StandardSocketOptions*
URLStreamHandler	*URI, URL*

Рассмотрим основные сетевые классы и примеры их применения.

КЛАСС INETADDRESS

Этот класс служит для инкапсуляции как числового *IP*-адреса, так и его доменного имени. Для взаимодействия с классом *InetAddress* используется имя *IP*-хоста, т.е. узла сети, которое намного удобнее и понятнее, чем *IP*-адрес. Числовое значение *IP*-адреса скрывается в классе *InetAddress*. Этот класс может оперировать адресами как по протоколу *IPv4*, так и по протоколу *IPv6*.

В классе *InetAddress* отсутствуют доступные конструкторы. Чтобы создать объект класса *InetAddress*, следует использовать один из доступных в нем фабричных методов. **Фабричные методы просто обозначают соглашение, по которому статические методы класса возвращают экземпляр этого класса.** Это делается вместо перегрузки конструктора с различными списками параметров, когда наличие однозначных имен методов проясняет результат. Ниже приведены общие формы трех наиболее используемых фабричных методов класса *InetAddress*.

Метод *getLocalHost()* возвращает объект типа *InetAddress*, представляющий локальный хост, метод *getByName()* – объект типа *InetAddress*, представляющий хост, имя которого передается в качестве параметра *host*. Если эти методы оказываются не в состоянии получить имя хоста, то они генерируют исключение типа *UnknownHostException*.

```
public static InetAddress getLocalHost() throws
UnknownHostException;

public static InetAddress getByName(String host) throws
UnknownHostException;
```

В Интернете считается обычным явлением, когда одно имя используется для обозначения нескольких машин. Это единственный путь обеспечить масштабируемость веб-серверов. Фабричный метод *getAllByName()* возвращает массив объектов типа *InetAddress*, представляющих все адреса, в которые преобразуется конкретное имя. Этот метод генерирует также исключение типа *UnknownHostException* в том случае, если он не в состоянии преобразовать имя хотя бы в один адрес.

```
public static InetAddress[] getAllByName(String host) throws
UnknownHostException;
```

В состав класса *InetAddress* входит также фабричный метод *getByAddress()*, который принимает *IP*-адрес и возвращает объект типа *InetAddress*. Причем могут использоваться адреса как по протоколу *IPv4*, так и по протоколу *IPv6*.

```
public static InetAddress getByAddress(byte[] addr) throws
UnknownHostException;
```

В следующем примере выводятся адреса и имена локальной машины, а также двух веб-сайтов сети Интернет.

```
//Example №1. Применение класса InetAddress
import java.net.*;
class InetAddressTest {
    public static void main(String args[]) throws
UnknownHostException {
        InetAddress inetAddress = InetAddress.getLocalHost();
        System.out.println(inetAddress);
        inetAddress = InetAddress.getByName("www.gmail.com");
        System.out.println(inetAddress);
        inetAddress = InetAddress.getByName("www.coursera.org");
        System.out.println(inetAddress);
        System.out.println("----------------------------------
----------------------");
        InetAddress addressesArray[] =
InetAddress.getAllByName("www.coursera.org");
        for (int i = 0; i < addressesArray.length; i++)
            System.out.println(addressesArray[i]);
        addressesArray =
InetAddress.getAllByName("www.mail.ru");
        for (InetAddress address : addressesArray)
            System.out.println(address);
    }}
```

Результат работы программы:

```
LAPTOP-63BTM6U2/192.168.100.25
www.gmail.com/142.250.186.165
www.coursera.org/18.66.233.45
--------------------------------
www.coursera.org/18.66.233.45
www.coursera.org/18.66.233.7
www.coursera.org/18.66.233.35
www.coursera.org/18.66.233.36
www.mail.ru/94.100.180.70
www.mail.ru/217.69.139.70
```

```
LAPTOP-63BTM6U2/192.168.100.25
Exception in thread "main" java.net.UnknownHostException Create breakpoint : No such host is known (www..com)
    at java.base/java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)
    at java.base/java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:52)
    at java.base/java.net.InetAddress$PlatformResolver.lookupByName(InetAddress.java:1048)
    at java.base/java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1638)
    at java.base/java.net.InetAddress$NameServiceAddresses.get(InetAddress.java:997)
    at java.base/java.net.InetAddress.getAllByName0(InetAddress.java:1628)
    at java.base/java.net.InetAddress.getAllByName(InetAddress.java:1494)
    at java.base/java.net.InetAddress.getByName(InetAddress.java:1389)
    at ex1.InetAddressTest.main(InetAddressTest.java:8)
```

IP-адреса, которые вы увидите на своей машине, могут отличаться о приведенных.

В классе *InetAddress* имеется ряд методов, наиболее употребительные из них перечислены в таблице.

Таблица 2 – Наиболее употребительные методы класса *InetAddress*

Метод	Описание
`public boolean equals(Object obj)`	Сравнивает текущий объект с указанным. Результат является истинным, когда аргумент не равен *null* и он представляет тот же IP-адрес, что и текущий объект
`public byte[] getAddress()`	Возвращает массив байтов, представляющий *IP*-адрес в порядке следования байтов в сети
`public String getHostAddress()`	Возвращает символьную строку, представляющую адрес хоста, связанного с объектом типа *InetAddress*
`public String getHostName()`	Возвращает символьную строку, представляющую имя хоста, связанного с объектом типа *InetAddress*
`public boolean isMulticastAddress()`	Метод для проверки того, является ли *InetAddress IP*-адресом многоадресной рассылки
`public String toString()`	Возвращает символьную строку, перечисляющую имя и *IP*-адрес хоста

Поиск адресов в Интернете осуществляется в последовательном ряде иерархически кэшированных серверов. Это означает, что компьютер может автоматически сопоставить конкретное имя с его *IP*-адресом, как в отношении себя, так и в отношении ближайших серверов. А в отношении всех прочих имен он может обращаться к *DNS*-серверам, откуда получаются сведения об *IP*-адресах. Если на таком сервере отсутствуют сведения об определенном адресе, то он может обратиться к следующему удаленному сайту и запросить у него эти сведения. Этот процесс может продолжаться вплоть до корневого сервера и потребовать немало времени. Поэтому структуру прикладного кода строят таким образом, чтобы сведения об *IP*-адресах локально кэшировались и их не приходилось искать каждый раз заново.

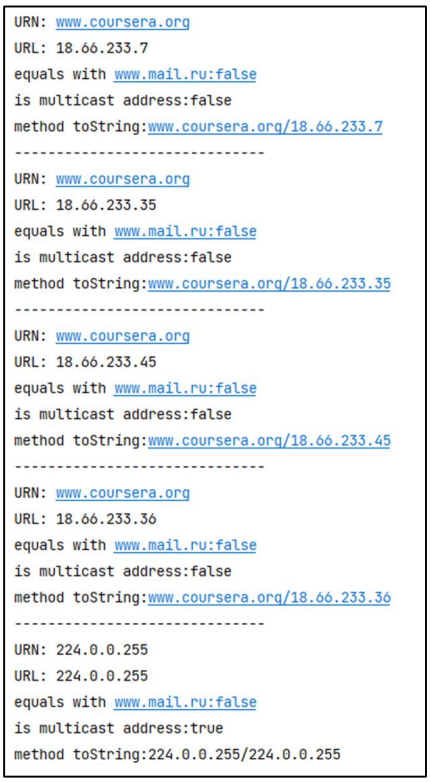
Для ускорения реализации сетевых запросов операционная система *Windows* кэширует ответы *DNS* серверов. Сразу после прихода ответа на определение числового *IP*-адреса по имени с сервера *DNS*, *Windows* автоматически помещает этот адрес в локальное хранилище. Когда браузер запрашивает адрес по имени, *Windows* вначале ищет его в хранилище, и, если находит его, то сразу же возвращает результат, не обращаясь к *DNS* серверам провайдера. Локальный кэш увеличивает скорость работы и экономит используемый трафик.

Кэширование в работе компьютера – это способ увеличения быстродействия компьютера за счет хранения часто используемых данных и кодов в «кэш-памяти 1-го уровня» (быстрой памяти), находящейся внутри микропроцессора. Кэш-память – очень быстрое запоминающее устройство. В общем, кэширование – это некий промежуточный буфер, в котором хранятся данные для более быстрого доступа к ним.

```
//Example №2. Использование методов класса InetAddress
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.UnknownHostException;
class InetAddressDemo {
```

```
    public static void main(String[] args) {
        try {
            InetAddress[] allByName =
InetAddress.getAllByName("www.coursera.org");
            for (InetAddress inetAddress : allByName) {
                if (inetAddress instanceof Inet4Address) {
                    System.out.println("URN: " +
inetAddress.getHostName());
                    System.out.println("URL: " +
inetAddress.getHostAddress());
                    System.out.println("equals with
www.mail.ru:" + inetAddress.equals("www.mail.ru"));
                    System.out.println("is multicast address:" +
inetAddress.isMulticastAddress());
                    System.out.println("method toString:" +
inetAddress);
                    System.out.println("-----------------------
------");
                }            }
            InetAddress group =
InetAddress.getByName("224.0.0.255");
            System.out.println("URN: " + group.getHostName());
            System.out.println("URL: " +
group.getHostAddress());
            System.out.println("equals with www.mail.ru:" +
group.equals("www.mail.ru"));
            System.out.println("is multicast address:" +
group.isMulticastAddress());
            System.out.println("method toString:" + group);
        } catch (UnknownHostException e) {
            System.out.println("Data about host isn't obtained."
+ e);
        }   }}
```

Результаты работы программы:



ПРОТОКОЛ TCP/IP

Сокеты по протоколу *TCP/IP* служат для реализации **надежных двунаправленных постоянных двухточечных потоковых соединений** между хостами в Интернете. **Сокет может служить для подключения системы ввода-вывода в *Java* к другим программам, которые могут находиться как на локальной машине, так и на любой другой машине в сети Internet.**

В *Java* поддерживается **две разновидности сокетов по протоколу *TCP/IP*: один – для серверов, другой – для клиентов**. Класс *ServerSocket* на серверной стороне приложения служит "приемником" (слушателем, *listener*), ожидая подключения клиентов прежде, чем выполнять какие-либо действия. Другими словами, класс *ServerSocket* предназначен для серверов, тогда как класс *Socket* – для серверов и для клиентов. Класс *Socket* служит для подключения к серверным сокетам и инициирования обмена данными по сетевому протоколу *TCP/IP*.

При создании объекта типа *Socket* неявно устанавливается соединение клиента с сервером. Выявить это соединение нельзя никакими методами или конструкторами. В таблице ниже перечислены публичные конструкторы класса *Socket*, предназначенные для создания клиентских сокетов.

Таблица 3– Конструкторы класса *Socket*

Конструктор	Описание
`public Socket()`	Создает неподключенный сокет
`public Socket(Proxy proxy)`	Создает неподключенный сокет, указывая тип прокси-сервера, если таковой имеется, который следует использовать независимо от любых других настроек
`public Socket(InetAddress`	Создает потоковый сокет и подключает его к указанному номеру порта по указанному IP-адресу

address, int port) throws IOException	
public Socket(String host, int port)throws UnknownHostException, IOException	Создает потоковый сокет и подключает его к указанному номеру порта на именованном хосте

В классе *Socket* определяется набор методов, например, объект типа *Socket* может быть проанализирован для получения сведений о связанных с ним *IP*-адресе и порте. Для этого применяются методы, перечисленные в таблице ниже.

Таблица 4– Методы класса *Socket*

Метод	Описание
public InetAddress getInetAddress()	Возвращает удаленный *IP*-адрес, к которому подключен этот сокет, или *null*, если сокет не подключен.
public int getPort()	Возвращает номер удаленного порта, к которому подключен этот сокет, или 0, если сокет еще не подключен
public InetAddress getLocalAddress()	Возвращает локальный адрес, к которому привязан сокет, адрес *loopback*, если диспетчер безопасности отказал, или *wildcard* адрес, если сокет закрыт или еще не привязан
public int getLocalPort()	Возвращает номер локального порта, к которому привязан этот сокет, или -1, если сокет еще не привязан

Для доступа к потокам ввода-вывода, связанным с классом *Socket*, можно воспользоваться методами *getInputStrueam()* и *getOuptutStream()*. Каждый из этих методов может сгенерировать исключение типа *IOException*, если сокет оказался недействительным из-за потери соединения. Эти потоки ввода-вывода используются для передачи и приема данных таким же образом, как и несетевые потоки ввода-вывода.

Таблица 5– Методы класса Socket для доступа к потокам ввода-вывода

Метод	Описание
public InputStream getInputStream() throws IOException	Возвращает входной поток для чтения байтов из этого сокета. Создает исключение *IOException* – если при создании входного потока возникает ошибка ввода-вывода, сокет закрыт, сокет не подключен или вход сокета был отключен с помощью *shutdownInput()*
public OutputStream getOutputStream() throws IOException	Возвращает выходной поток для записи байтов в этот сокет. Выдает исключение *IOException* – если при создании выходного потока возникает ошибка ввода-вывода или если сокет не подключен

Имеется набор других методов, в том числе метод *void connect (SocketAddress endpoint)*, позволяющий создать новое соединение; метод *boolean isConnected()*, возвращающий логическое значение *true*, если сокет подключен к серверу; метод *boolean isBound()*, возвращающий логическое значение *true*, если сокет привязан к адресу; а также метод *boolean isClosed()*, возвращающий логическое значение *true*, если сокет закрыт.

```
//Example №3. Использование метода connect() класса Socket
import java.net.InetAddress;
import java.net.InetSocketAddress;
```

```
import java.net.Socket;
import java.net.SocketAddress;
class MethodConnectDemo {
    public static void main(String[] argv) throws Exception {
        InetAddress addr =
InetAddress.getByName("java.sun.com");
        int port = 80;
        SocketAddress sockaddr = new InetSocketAddress(addr,
port);
        Socket sock = new Socket();
        sock.connect(sockaddr);
        System.out.println("Соединение установлено:
"+sock.isConnected());
    }
}
```

Результаты работы программы:

```
Соединение установлено: true
```

Чтобы закрыть сокет, необходимо вызвать метод *close()*. Закрытие сокета приводит к закрытию связанных с ним потоков ввода-вывода. Начиная с версии *JDK* 7 класс *Socket* реализует также интерфейс *AutoCloseable*. Это означает, что управление сокетом можно организовать в блоке оператора *try* с ресурсами.

Приведенная ниже программа служит примером использования класса *Socket*. В этой программе устанавливается соединение с портом "*whois*" (основное применение протокола *whois* – получение регистрационных данных о владельцах доменных имён, *IP*-адресов и автономных систем) на *Verisign-grs*-сервере, посылается аргумент командой строки через сокет, а затем выводятся возвращаемые данные. *Google*-сервер пытается интерпретировать аргумент как зарегистрированное доменное имя Интернет, а затем возвращает *IP*-адрес и контактную информацию с веб-сайта, найденного по этому доменному имени.

```
//Example №4. Демонстрация обращения с сокетами. САМОСТОЯТЕЛЬНО.
ЗАПУСТИТЬ ПРИЛОЖЕНИЕ, ПЕРЕДАВ АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ
import java.net.*;
import java.io.*;
class Whois {
    public static void main(String args[]) throws Exception {
        int c;
        //создать сокетное соединение с веб-сайтом verisign-
grs.com через порт 43
        Socket s = new Socket("whois.verisign-grs.com", 43);
        // получить потоки ввода-вывода
        InputStream socketInputStream = s.getInputStream();
        OutputStream socketOutputStream = s.getOutputStream();
        // сформровать строку запроса
        String str=(args.length==0 ? "GOOGLE.COM":args[0]) + "
\n";
        byte buf[] = str.getBytes();//преобразовать строку в
```

```
байты
            socketOutputStream.write(buf); //отправить запрос
            // прочитать ответ и вывести его на экран
            while ((c = socketInputStream.read()) != -1) {
                System.out.print((char) c);
            }
            s.close();
        }}
```

Если запросить, например, сведения об адресе *google.com*, то будет такой результат:

```
   Domain Name: GOOGLE.COM
   Registry Domain ID: 2138514_DOMAIN_COM-VRSN
   Registrar WHOIS Server: whois.markmonitor.com
   Registrar URL: http://www.markmonitor.com
   Updated Date: 2019-09-09T15:39:04Z
   Creation Date: 1997-09-15T04:00:00Z
   Registry Expiry Date: 2028-09-14T04:00:00Z
   Registrar: MarkMonitor Inc.
   Registrar IANA ID: 292
   Registrar Abuse Contact Email: abusecomplaints@markmonitor.com
   Registrar Abuse Contact Phone: +1.2086851750
   Domain Status: clientDeleteProhibited https://icann.org/epp#clientDeleteProhibited
   Domain Status: clientTransferProhibited https://icann.org/epp#clientTransferProhibited
   Domain Status: clientUpdateProhibited https://icann.org/epp#clientUpdateProhibited
   Domain Status: serverDeleteProhibited https://icann.org/epp#serverDeleteProhibited
   Domain Status: serverTransferProhibited https://icann.org/epp#serverTransferProhibited
   Domain Status: serverUpdateProhibited https://icann.org/epp#serverUpdateProhibited
   Name Server: NS1.GOOGLE.COM
   Name Server: NS2.GOOGLE.COM
   Name Server: NS3.GOOGLE.COM
   Name Server: NS4.GOOGLE.COM
   DNSSEC: unsigned
   URL of the ICANN Whois Inaccuracy Complaint Form: https://www.icann.org/wicf/
>>> Last update of whois database: 2023-02-21T10:48:10Z <<<

For more information on Whois status codes, please visit https://icann.org/epp
```

Эта программа действует следующим образом. Сначала в ней создается объект типа *Socket*, задающий имя хоста "*whois.verisign-grs.com*" и номер порта 43. Затем на основе сокета формируются потоки ввода-вывода. Далее формируется символьная строка, содержащая имя веб-сайта, сведения о котором требуется получить. Если веб-сайт не указан в командной строке, то выбирается имя хоста "*GOOGLE.COM*". Эта символьная строка преобразуется в массив байт и направляется в сеть через сокет. После этого ответ читается из сокета, а результат выводится на экран. И наконец, сокет закрывается, а вместе с ним и потоки ввода-вывода. В данном примере сокет закрывается вручную в результате вызова метода *close()*.

Для автоматического закрытия сокета можно организовать блок оператора *try* с ресурсами. Ниже приведен другой способ создания метода *main()* для реализации автоматического закрытия сокета.

```
//Example №4.1 Использование блока try with resources.
САМОСТОЯТЕЛЬНО: ОБРАБОТАТЬ ИСКЛЮЧИТЕЛЬНУЮ СИТУАЦИЮ В МЕТОДЕ
MAIN(УБРАТЬ ОПЕРАТОР THROWS)
import java.net.*;
import java.io.*;
class Whois {
    public static void main(String args[]) throws Exception {
        int c;
        // создать сокетное соединение с вес-сайтом
whois.verisign-grs.com через порт 43
        try(Socket s=new Socket("whois.verisign-grs.com", 43)){
            // получить потоки ввода-вывода
            InputStream socketInputStream  = s.getInputStream();
            OutputStream socketOutputStream =
s.getOutputStream();
            // сформровать строку запроса
            String str = (args.length == 0 ? "GOOGLE.COM" :
args[0]) + " \n ";
            byte buf[] = str.getBytes();//преобразовать строку в
байты
            socketOutputStream.write(buf);// послать запрос
            // прочитать ответ и вывести его на экран
            while ((c = socketInputStream.read()) != -1) {
                System.out.print((char) c);
            }}}}
```

В приведенных далее примерах используется явное закрытие сетевых ресурсов. Но в новом прикладном коде следует использовать автоматическое управление сетевыми ресурсами, поскольку это более рациональный и гибкий подход.

СЕРВЕРНЫЕ СОКЕТЫ ПО ПРОТОКОЛУ TCP/IP

В *Java* имеются различные классы сокетов, которые должны применяться при разработке серверных приложений. В частности, класс *ServerSocket* применяется для создания серверов, которые принимают запросы как от локальных, так и от удаленных клиентских программ, желающих установить соединение с ними через открытые порты. Класс *ServerSocket* отличается от обычных классов типа *Socket*. Когда создается объект класса *ServerSocket*, он регистрируется в системе как заинтересованный в соединении с клиентами.

Конструкторы класса *ServerSocket* получают номер порта, через который требуется принимать запросы на соединение, а также (хотя и необязательно) длину очереди, которая организуется для данного порта. Длина очереди сообщает системе, сколько клиентских соединений можно поддерживать, прежде чем отказать в соединении. **По умолчанию задается длина очереди 50 соединений.** При определенных условия конструкторы класса *ServerSocket* могут сгенерировать исключение типа *IOException*. Конструкторы этого класса перечислены в таблице ниже.

Таблица 6– Конструкторы класса *ServerSocket*

Конструктор	Описание

`public ServerSocket() throws IOException`	Создает несвязанный серверный сокет
`public ServerSocket(int port) throws IOException`	Создает ServerSocket, привязанный к указанному порту. Номер порта, равный 0, означает, что номер порта присваивается автоматически, обычно из эфемерного диапазона портов. Затем этот номер порта можно получить, вызвав *getLocalPort()*. Максимальная длина очереди для входящих запросов на создание подключения равной 50. Если уведомление о подключении поступает, когда очередь заполнена, в подключении будет отказано.
`public ServerSocket(int port, int backlog) throws IOException`	Создает серверный сокет и привязывает его к указанному номеру локального порта с указанным отставанием. Номер порта, равный 0, означает, что номер порта присваивается автоматически, обычно из эфемерного диапазона портов. Затем этот номер порта можно получить, вызвав *getLocalPort()*.Максимальная длина очереди для входящих запросов на создание подключения устанавливается параметром *backlog*. Если уведомление о подключении поступает, когда очередь заполнена, в подключении будет отказано.
`public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException`	Создайте сервер с указанным портом, очередью прослушивания и локальным *IP*-адресом для привязки. Аргумент *bindAddr* может использоваться на мультидоменном хосте для *ServerSocket*, который будет принимать запросы на подключение только к одному из своих адресов. Если *bindAddr* имеет значение *null*, то по умолчанию он будет принимать соединения по любым/всем локальным адресам. Порт должен находиться в диапазоне от 0 до 65535 включительно. Номер порта, равный 0, означает, что номер порта присваивается автоматически, обычно из эфемерного диапазона портов. Затем этот номер порта можно получить, вызвав *getLocalPort()*.

В классе *ServerSocket* есть метод *accept()*, который реализует **блокирующий вызов**, ожидая от клиента начала соединения, а затем возвращает объект типа *Socket*, который может далее служить для взаимодействия с клиентом.

```
//Example №5. Сетевое соединение по протоколу TCP/IP. Код программы-сервера
import java.io.*;
import java.net.*;
class ServerFirst {
    public static void main(String[] arg) {
        ServerSocket serverSocket = null;
        Socket clientAccepted = null;
        ObjectInputStream sois = null;
        ObjectOutputStream soos =null;
        try {
            serverSocket = new ServerSocket(2525);
            System.out.println("server starting....");
            System.out.println("at
```

```
IP="+InetAddress.getLocalHost().getHostAddress());
            System.out.println("at
port="+serverSocket.getLocalPort());
            clientAccepted = serverSocket.accept();
            System.out.println("connection established....");
            System.out.println("with IP="+
clientAccepted.getInetAddress());
            System.out.println("at
port="+clientAccepted.getPort());
            sois=new
ObjectInputStream(clientAccepted.getInputStream());
            soos=new
ObjectOutputStream(clientAccepted.getOutputStream());
            String clientMessageRecieved = (String)
sois.readObject();
            while (!clientMessageRecieved.equals("quite")){
                System.out.println("message recieved:" +
clientMessageRecieved);

clientMessageRecieved=clientMessageRecieved.toUpperCase();
                soos.writeObject(clientMessageRecieved);
                clientMessageRecieved = (String)
sois.readObject();
            }
        } catch (IOException | ClassNotFoundException e) {
        } finally {
            try {
                if (sois != null)sois.close();
                if (soos != null) soos.close();
                if (clientAccepted !=
null)clientAccepted.close();
                if (serverSocket != null) serverSocket.close();
            } catch (IOException e) {
                System.out.println("Ресурсы не закрыты!!!");
            }}}}
//Код программы клиента
import java.io.*;
import java.net.Socket;
public class ClientFirst {
    public static void main(String[] arg) throws IOException,
ClassNotFoundException {
        try (Socket clientSocket = new Socket("127.0.0.1",
2525)) {
            System.out.println("connection established....");
            BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
            ObjectOutputStream coos = new
ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream cois = new
ObjectInputStream(clientSocket.getInputStream());
            System.out.println("Enter string to send to server
\n\t('quite' - programme terminate)");
            String clientMessage = stdin.readLine();
```

```
            System.out.println("you've entered: " +
clientMessage);
            while (!clientMessage.equals("quite")) {
                coos.writeObject(clientMessage);
                System.out.println("~server~: " +
cois.readObject());
              System.out.println("----------------------------");
                clientMessage = stdin.readLine();
                System.out.println("you've entered: " +
clientMessage);
            }}}}
```

Результаты работы программы (серверная и клиентская консоль):

```
server starting....
at IP=192.168.100.17
at port=2525
connection established....
with IP=/127.0.0.1
at port=54263
message recieved:Первая строка данных
message recieved:Вторая строка данных
message recieved:Третья строка данных 57984758647
```

```
connection established....
Enter string to send to server
    ('quite' - programme terminate)
Первая строка данных
you've entered: Первая строка данных
~server~: ПЕРВАЯ СТРОКА ДАННЫХ
----------------------------
Вторая строка данных
you've entered: Вторая строка данных
~server~: ВТОРАЯ СТРОКА ДАННЫХ
----------------------------
Третья строка данных 57984758647
you've entered: Третья строка данных 57984758647
~server~: ТРЕТЬЯ СТРОКА ДАННЫХ 57984758647
----------------------------
quite
you've entered: quite
```

ИСПОЛЬЗОВАНИЕ МНОГОПОТОЧНОСТИ ПРИ СОЗДАНИИ КЛИЕНТ-СЕРВЕРНОГО СОЕДИНЕНИЯ

Сервер должен поддерживать многопоточность, иначе он будет не в состоянии обрабатывать несколько соединений одновременно. В этом случае сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс *ServerThread*, расширяющий класс *Thread*, и используется затем для соединений с многими клиентами каждый в своем потоке.

```
//Example №6 Код приложения-сервера
import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
public class MultiThreadServerMain {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(8071, 10,
InetAddress.getLocalHost());
            System.out.println("initialized: " +
server.getInetAddress()+","+server.getLocalPort());
            while (true) {
                Socket socket = server.accept();//ожидание
подключения клиента
                System.out.println(socket.getInetAddress()+"
connected"+","+socket.getPort());
                //создание отдельного потока для обмена данными
```

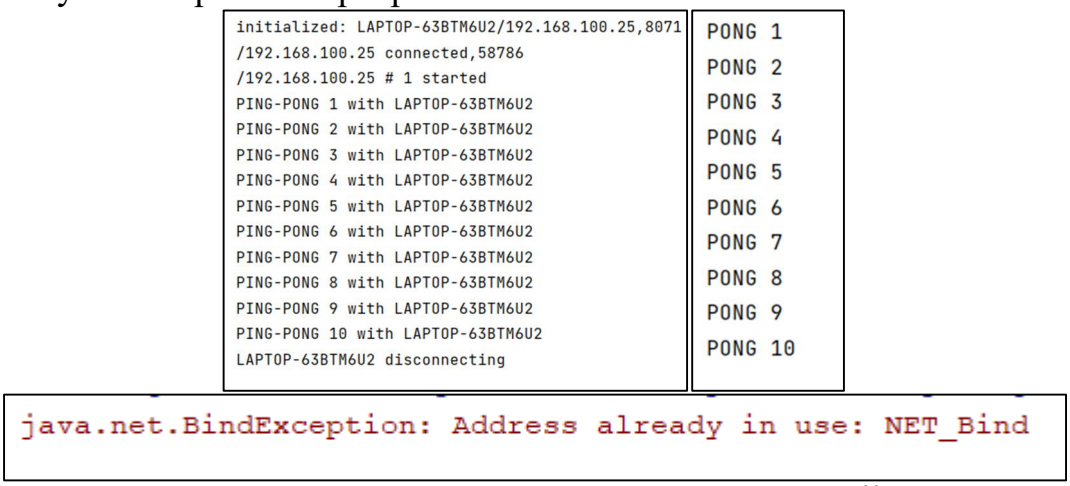
```
с соединившимся клиентом
                ServerThread thread = new ServerThread(socket);
                thread.start();// запуск потока
                System.out.println(socket.getInetAddress()+" №
"+ServerThread.getCounter()+ " started");
            }
        } catch (IOException e) {System.err.println(e);}}}

//Код приложения клиента
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
public class MultiClient {
    public static void main(String[] args) {
        Socket socket = null;
        BufferedReader br = null;
        try {
            // установка соединения с сервером
            socket = new Socket(InetAddress.getLocalHost(),
8071);
            //или Socket socket = new Socket("ИМЯ_СЕРВЕРА",
8071);
            PrintStream ps = new
PrintStream(socket.getOutputStream());
            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            for (int i = 1; i <= 10; i++) {
                ps.println("PING");
                System.out.println(br.readLine());
                Thread.sleep(1_000);
            }
        } catch (UnknownHostException e) {
            // если не удалось соединиться с сервером
            System.err.println("адрес недоступен" + e);
        } catch (IOException e) {
            System.err.println("ошибка I/O потока" + e);
        } catch (InterruptedException e) {
            System.err.println("ошибка потока выполнения" + e);
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (socket != null) {
                try {
                    socket.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                } } } }}
// код создания отдельного потока исполнения
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
class ServerThread extends Thread {
    private PrintStream os;
    private BufferedReader is;
    private InetAddress addr; // адрес клиента
    private static int counter = 0;
    public ServerThread(Socket s) throws IOException {
        os = new PrintStream(s.getOutputStream());
        is = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        addr = s.getInetAddress();
        counter++;
    }
    public static int getCounter() {
        return counter;
    }
    public void run() {
        int i = 0;
        String str;
        try {
            while ((str = is.readLine()) != null) {
                if ("PING".equals(str)) {
                    os.println("PONG " + ++i);
                }
                System.out.println("PING-PONG " + i + " with " +
addr.getHostName());
            }
        } catch (IOException e) {
            // если клиент не отвечает, соединение с ним
разрывается
            System.err.println("Disconnect");
        } finally {
            disconnect(); // уничтожение потока
        }
    }
    public void disconnect() {
        try {
            if (os != null) {
                os.close();
            }
            if (is != null) {
                is.close();
            }
            System.out.println(addr.getHostName() + "
```

```
disconnecting");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            this.interrupt();
        }
    }
}
```

В примере сервер передает сообщение клиенту. Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции. Сервер должен быть инициализирован до того, как клиент попытается осуществить сокетное соединение.

Результаты работы программы:




```
java.net.BindException: Address already in use: NET_Bind
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Что такое *IP*-адрес? В чем различие между версиями интернет-протокола *IPv4* и *IPv6*?
2. В чем отличие понятий *host* (хост) и *domain* (домен)?
3. В чем отличие *URI*, *URL*, *URN*?
4. Что такое сокет? Для чего он предназначен?
5. В чем отличие классов *Socket* и *ServerSocket*?
6. Что такое протокол передачи данных? Приведите примеры протоколов.
7. Для чего предназначен стек протоколов *TCP/IP*?
8. Назовите уровни сетевой модели *OSI*.
9. Для чего предназначены *DNS* сервера?
10. Когда создается исключение *UnknownHostException*?
11. Какой пакет в платформе *Java* содержит основные типы для автоматизации работы в сети?
12. Опишите алгоритм создания клиент-серверного приложения на Java с использованием протокола *TCP/IP*.

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.

2. Ответить на контрольные вопросы лабораторной работы.

3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. диаграмму созданных классов в нотации UML
8. выводы по лабораторной работе

ВО ВСЕХ ЗАДАНИЯХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. ВСЕ РЕШАЕМЫЕ ЗАДАЧИ ДОЛЖНЫ БЫТЬ РЕАЛИЗОВАНЫ, ИСПОЛЬЗУЯ КЛАССЫ И ОБЪЕКТЫ.

В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО:

Необходимо разработать многопоточное клиент-серверное приложение на языке программирования *Java* с использованием протокола передачи данных *TCP/IP*. Смоделировать работу согласно варианту задания. Все параметры заданий необходимо вводить с клавиатуры. Все отправленные и полученный данные записываются в файл на серверной стороне приложения.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Функционирование клиента и сервера реализовать следующим образом: клиент посылает два числа серверу и одну из математических операций: «*», «/», «+», «–»,– сервер соответственно умножает, делит, складывает либо вычитает эти два числа и ответ посылает ответ назад клиенту.

2. Функционирование клиента и сервера реализовать следующим образом: клиент посылает слово серверу, сервер возвращает назад клиенту это слово в обратном порядке следования букв, если встречается спецсимвол, то он заменяется на знак *.

3. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет строку данных серверу, сервер возвращает строку в обратном порядке следования букв, в верхнем регистре, зашифрованную любым способом.

4. Функционирование клиента и сервера реализовать следующим образом: клиент посылает два числа серверу m и n, сервер возвращает m!+n! этих чисел назад клиенту.

5. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет две строки серверу, сервер их сравнивает и возвращает «строки равны», если они одинаковы с учетом регистра, иначе возвращает значение «строки не равны».

6. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет набор латинских букв и спецсимволов (например, !, «, №, $...) серверу и получает назад символы, упорядоченными по алфавиту, не учитывая спецсимволы.

7. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу произвольный набор символов, сервер замещает каждый четвертый символ на «%», каждый пятый на символ «#».

8. Функционирование клиента и сервера реализовать следующим образом: сервер хранит данные о прогнозе погоды. Клиент отправляет дату и получает соответствующий прогноз.

9. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу произвольные дробные числа и получает назад количество чисел, кратных трем, наибольший и наименьший элементы.

10. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу символьную строку, содержащую пробелы, и получает назад ту же строку, но в ней между словами должен находиться только один пробел. Сервер также возвращает количество слов в строке.

11. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу две символьные строки. Сервер редактирует строки, удаляя из них идущие друг за другом одинаковые слова. Сервер определяет самое длинное общее слово в полученных заданных строках.

12. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу ФИО. Сервер определяет, находится ли этот человек в списке, хранящемся на сервере, и возвращает все данные об этом человеке.

13. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу два числа и получает «наибольший общий делитель» (наибольший общий делитель) и «наименьшее общее кратное» этих чисел.

14. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу число от 0 до 10 и получает название этого числа прописью на русском, белорусском и английском языкам.

15. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу русское слово или фразу и получает перевод с русского языка на английский.

16. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу координаты точки X и У в декартовой системе координат. Сервер определяет, в какой координатной четверти находится данная точка и отправляет результат клиенту.

17. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу координаты прямоугольной области и точки в декартовой системе координат. Сервер определяет, лежит ли данная точка в прямоугольной области, и отправляет результат клиенту.

18. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу девятизначный номер счета и pin code. Сервер определяет, существует ли такой номер счета в системе и соответствует ли ему указанный pin code. Результат возвращается клиенту.

19. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу *ID* клиента и кредитный баланс. Сервер определяет, существует ли такой номер счета в системе и анализирует его кредитный баланс. Если кредитный баланс больше указанного клиентом, то возвращается сообщение «Кредит недоступен» иначе «Кредит доступен».

20. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу набор названий в стиле *camelCase*. Сервер возвращает набор слов в стиле *snake_case*.

21. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу текст и ключ «*encryption*» или «*decryption*». Если получен ключ «*encryption*», то сервер реализует шифрование с помощью шифра "Сэндвич": текст разбивается на две одинаковые по количеству символов части, результатом шифрования является текст, в которой символы из первой части чередуются символами из второй части; если получен ключ «*decryption*», то сервер реализует дешифрование. Сервер возвращает полученный зашифрованный текст.

22. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу набор строк, в каждой из которых содержится сначала число, а затем некоторая строка. Выполнить сортировку строк по числу, а строки с одинаковыми числами упорядочить по оставшейся части. Сервер возвращает отсортированный набор.

23. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу набор строк. Сервер возвращает каждую строку в обратном (реверсивном) порядке, изменив расположение символов в каждой строке задом наперёд.

24. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу название алфавита, сервер возвращает клиенту массив символов указанного алфавита (видов алфавитов должны быть обязательно следующие: английский, белорусский, русский).

25. Функционирование клиента и сервера реализовать следующим образом: клиент посылает серверу шестизначный номер билета. Определить, является ли этот билет «счастливым». «Счастливым» называется такой билет, у которого сумма первых трех цифр равна сумме последних трех.

26. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу количество жителей в государстве и площадь его территории. Сервер рассчитывает и возвращает плотность населения в этом государстве.

27. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу предложение, в котором слова разделены одним пробелом. Сервер определяет и возвращает самое «длинное» слово в предложении.

28. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу строк, содержащую наименование e-mail. Сервер определяет корректен ли формат названия e-mail. Корректность e-mail определяется по следующим правилам: название может состоять из 6–30 знаков и содержать буквы, цифры и символы, может содержать буквы латинского алфавита (a–z), цифры (0–9) и точки (.), запрещено использовать амперсанд (&), знаки равенства (=) и сложения (+), скобки (<>), запятую (,), символ подчеркивания (_), апостроф ('), дефис (-) и несколько точек подряд.

29. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу количество человек. Сервер случайным образом заполняет массив этого размера значениями среднего балла этих человек и возвращает их клиенту.

30. Функционирование клиента и сервера реализовать следующим образом: клиент отправляет серверу число 0 или 1. Если сервер получил число 1, то он заполняет массив таким образом, чтобы значения элементов образовывали убывающую последовательность, если клиент отправил 0, то возрастающую последовательность.