

## Тема: Basics of Java Network Programming (Part 1)

### **План занятия:**

1. Основы работы в сети
  - основные понятия
  - протокол, стек протоколов TCP
2. Адреса, порты и сетевые интерфейсы
  - классы Java, управляющие работой
  - класс *InetAddress*
  - класс *NetworkInterface*
3. Программная работа с TCP протоколом
  - серверные сокеты
  - клиентские сокеты
4. Основные этапы создания клиент / серверного приложения на потоковых сокетах
  - простой пример
  - сервер, работающий с многими клиентами: итерационный подход
  - сервер, работающий с многими клиентами: параллельный подход
5. Передача объектов по сети

### **Литература**

1. Кей Хорстманн, Гари Корнелл «*Java. Библиотека профессионала. Том 2*»
2. Harold E. R. *Java Network Programming*: - O'Reilly, 2005 (2013) – 735 p.
3. Trail: Custom Networking:  
<https://docs.oracle.com/javase/tutorial/networking/>
4. Jakob Jenkov Java Networking Tutorial: <http://tutorials.jenkov.com/java-networking/index.html>
5. Jakob Jenkov Multithreaded Servers in Java Tutorial:  
<http://tutorials.jenkov.com/java-multithreaded-servers/index.html>
6. Jan Graba *An Introduction to Network Programming with Java*, 2013
7. Роберт Орфали, Дан Харки. *Java и CORBA в приложениях клиент-сервер*. Издательство: Лори, 2000 - 734 с.
8. Патрик Нимейер, Дэниэл Леук. *Программирование на Java* (Исчерпывающее руководство для профессионалов). М.: Эксмо, 2014

### **Основы работы в сети**

Словосочетание «сетевое программирование» обозначает написание программ, которые выполняются на нескольких устройствах (компьютерах), в

которых все устройства подключены друг к другу с помощью сети. Начиная с данной темы, мы приступаем к работе с такими программами, которые, даже будучи правильно написанными, могут не всегда работать. Важно не только правильно написать, но и правильно запустить приложение.

Следует отметить, что сетевые программы, можно условно разделить на две категории:

- первый вид приложений – передача файлов и данных между хостами;
- второй вид приложений – связан с вызовом одним хостом программ на другом хосте.

При изучении этой темы мы будем, в основном, рассматривать первый тип приложений.

В основном мы будем работать с пакетом `java.net`, содержащим набор классов и интерфейсов, которые позволяют организовать «низкоуровневое» сетевое взаимодействие. Но, прежде чем разбирать особенности этих классов и интерфейсов, очень кратко рассмотрим основные понятия, необходимые для организации взаимодействия компьютеров через сеть.

## Компьютерные сети

*Сеть (network)* - это совокупность компьютеров и других устройств, которые могут отправлять и получать данные друг от друга, более или менее в режиме реального времени. Каждая машина в сети будем называть *узлом (host)*. Большинство узлов - это компьютеры, но узлами могут быть также принтеры, маршрутизаторы, мосты, шлюзы, немые терминалы и т. д. Иногда слово «*узел*» используется для обозначения любого устройства в сети, а слово «*хост*» - для обозначения узла, который является компьютером общего назначения.

Каждый сетевой узел имеет *адрес (address)* – последовательность байт, которые однозначно его идентифицируют. Чем больше байт содержит адрес, тем больше адресов доступно и тем больше устройств можно подключить к сети одновременно. В сетях разных типов адреса назначаются по-разному.

Все современные компьютерные сети являются сетями с коммутацией пакетов (*packet-switched networks*): данные, передаваемые по сети, разбиваются на куски, называемые пакетами, и каждый пакет обрабатывается отдельно. Каждый пакет содержит информацию о том, кто его отправил и куда он идет. Наиболее важным преимуществом разбивки данных на индивидуально адресуемые пакеты является то, что пакеты от многих текущих источников могут передаваться по одному каналу связи, что значительно удешевляет построение сети: многие компьютеры могут без помех использовать один и тот же «провод». Еще одно преимущество использования

пакетов заключается в том, что можно использовать контрольные суммы для определения того, был ли пакет поврежден при передаче.

Для взаимодействия устройств в сети очень важны правила, согласно которым компьютеры передают данные. *Протокол (protocol)* - это точный набор правил, определяющих, как компьютеры взаимодействуют: формат адресов, способ разделения данных на пакеты и т. д. Существует много разных протоколов, определяющих различные аспекты сетевого взаимодействия. Например, можно привести пример протоколы *HTTP (Hypertext Transfer Protocol)*, который определяет, как взаимодействуют веб-браузеры и серверы; и пример «другого уровня» - стандарт *IEEE 802.3*, который определяет правила того, каким образом биты кодируются в виде электрических сигналов на среде передачи определенного типа. Открытые, опубликованные стандарты протоколов позволяют программному обеспечению и оборудованию разных производителей обмениваться данными друг с другом: веб-браузеру совершенно не важно, является ли какой-либо конкретный сервер рабочей станцией *Unix*, *Windows* или *Macintosh*, потому что сервер и браузер, независимо от платформы, используют на один и тот же *HTTP* протокол.

## **Сетевые модели**

Отправка данных по сети - сложная процедура, которая должна быть тщательно настроена и учитывать как физические характеристики сети, так и логический характер отправляемых данных. Программное обеспечение, которое отправляет данные по сети, должно понимать, как избежать коллизий между пакетами, как преобразовывать цифровые данные в аналоговые сигналы, как обнаруживать и исправлять ошибки, как маршрутизировать пакеты от одного хоста к другому и многое другое. Процесс становится еще более сложным, так как необходимо поддерживать различные операционные системы и разнородные сетевые кабели.

Чтобы управлять этой сложностью и скрыть большую ее часть от разработчика приложения и конечного пользователя, различные аспекты сетевого взаимодействия разделены на несколько уровней. Каждый уровень представляет различный уровень абстракции между физическим оборудованием (например, проводами и электричеством) и передаваемой информацией. Каждый слой имеет строго ограниченную функцию. Например, один уровень может отвечать за маршрутизацию пакетов, тогда как уровень над ним отвечает за обнаружение и запрос повторной передачи поврежденных пакетов. Теоретически, каждый слой говорит только со слоями, расположенными непосредственно над и сразу под ним. Разделение сети на уровни позволяет изменять или даже заменять программное обеспечение на

одном уровне, не затрагивая другие, при условии, что интерфейсы между уровнями остаются неизменными.

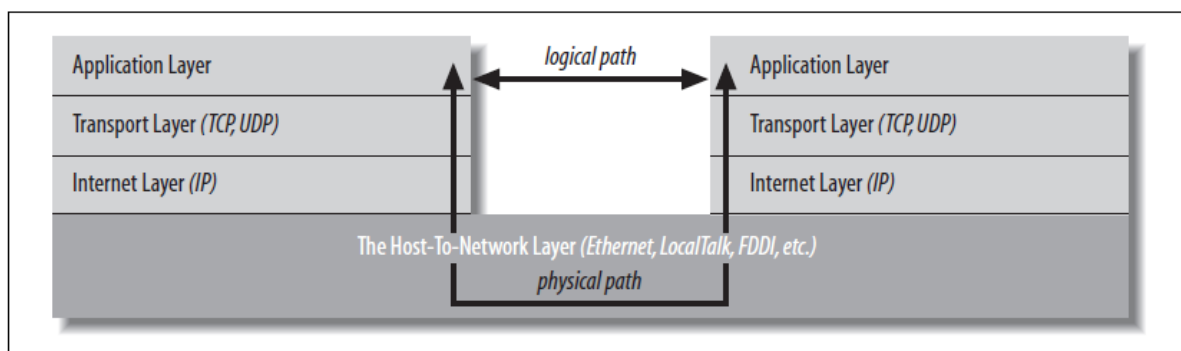
Таким образом, при рассмотрении особенностей сетевого взаимодействия применяют разные модели слоев, например сетевая модель *OSI* — сетевая модель стека сетевых протоколов *OSI/ISO*. Но, для практической работы в рамках технологии *Java*, удобнее использовать сетевую модель *TCP/IP*. Стек протоколов *TCP/IP* включает в себя четыре уровня:

- *прикладной уровень (Application Layer)*,
- *транспортный уровень (Transport Layer)*,
- *межсетевой уровень или сетевой уровень (Internet Layer)*,
- *канальный уровень (Network Access Layer)*.

Протоколы этих уровней полностью реализуют функциональные возможности модели *OSI*. На стеке протоколов *TCP/IP* построено всё взаимодействие пользователей в *IP*-сетях. Стек является независимым от физической среды передачи данных, благодаря чему, в частности, обеспечивается полностью прозрачное взаимодействие между проводными и беспроводными сетями.

В рамках этой модели такие приложения, как, например, *Internet Explorer*, работают на *прикладном уровне* и взаимодействуют только с *транспортным уровнем* на локальном клиентском компьютере. *Транспортный уровень* взаимодействует только с *прикладным уровнем* и *сетевым уровнем*. На *транспортном уровне* запрос разбивается на сегменты *TCP*, добавляет к данным некоторые порядковые номера и контрольные суммы, а затем передает запрос локальному *сетевому уровню*. *Сетевой уровень*, в свою очередь, взаимодействует только с *канальным уровнем* и *транспортным уровнем*. *Сетевой уровень* фрагментирует сегменты в дейтаграммы *IP* необходимого размера для локальной сети и передает их на *канальный уровень* для передачи по проводам. *Канальный уровень* кодирует цифровые данные в виде аналоговых сигналов, подходящих для конкретного физического носителя, и отправляет запрос по проводной линии, где они будут считываться *канальным уровнем* удаленной системы, которой они адресованы. То есть, на *канальном уровне* данные перемещаются по проводам, оптоволоконным кабелям или другому носителю на *канальный уровень* в удаленной системе, а затем данные перемещаются по уровням вверх на удаленной системе. *Канальный уровень* на удаленной системе декодирует аналоговые сигналы в цифровые данные, а затем передает полученные *IP*-дейтаграммы на *сетевой уровень*. *Сетевой уровень* выполняет несколько простых проверок, чтобы убедиться, что дейтаграммы *IP* не повреждены, собирает их, если они были фрагментированы, и передает их на

*транспортный уровень*. Транспортный уровень проверяет, все ли данные получены, и запрашивает повторную передачу любых отсутствующих или поврежденных фрагментов. Как только *транспортный уровень* получает достаточно последовательных дейтаграмм, он собирает их и записывает их в поток, считываемый веб-сервером, работающим на *прикладном уровне*. Такая схема показана на рисунке. Следует отметить различие в логическом и физическом пути движения данных.



Более подробно о каждом уровне можно прочитать в литературе.

## IP адреса и доменные имена

*Java* программист обычно не сталкивается с внутренним устройством *IP*, но нужно знать иметь представление об адресации. Каждый компьютер в сети *IPv4* идентифицируется четырехбайтовым номером. Обычно, этот номер записывается в специальной точечной четырехбайтной нотации, например 199.1.32.90, где каждое из четырех чисел имеет типичное для беззнакового байтового числа значение в диапазоне от 0 до 255. Каждый компьютер, подключенный к сети *IPv4*, имеет уникальный четырехбайтовый адрес.

При передачи данных по сети формируются пакеты, заголовок которых включает в себя адрес машины, для которой предназначен пакет (адрес назначения), и адрес машины, которая отправила пакет (адрес источника). Маршрутизаторы по пути выбирают лучший маршрут для отправки пакета, проверяя адрес назначения. В пакете указывается адрес источника для того, чтобы получатель знал, кому отвечать.

Исходя из способа формирования, понятно, что существует ограниченное количество *IP*-адресов - чуть более четырех миллиардов. Получается, что их меньше, чем людей на планете, а тем более, меньше, чем компьютеров. Сейчас идет переход к адресации *IPv6*, где используются 16-байтовые адреса. Это обеспечивает достаточное количество *IP*-адресов для идентификации каждого человека и каждого компьютера (даже каждого атома на планете). Адреса *IPv6* обычно записываются в восьми блоках из четырех шестнадцатеричных цифр, разделенных двоеточиями: BA98:7654:3210:FEDC:BA98:7654:3210. Если

две и более групп подряд равны 0000, то они могут быть опущены и заменены на двойное двоеточие (::). Незначащие старшие нули в группах могут быть опущены. Например, 2001:0db8:0000:0000:0000:0000:ae21:ad12 может быть сокращён до 2001:db8::ae21:ad12, а 0000:0000:0000:0000:0000:0000:ae21:ad12 может быть сокращен до ::ae21:ad12. Сокращению не могут быть подвергнуты две разделенные нулевые группы из-за возникновения неоднозначности.

Существует специальная нотация для отображения адресов *IPv4* на *IPv6*.

Компьютерам очень удобно работать с числами, но людям не очень удобно их запоминать. Для удобства работы была разработана *система доменных имен (DNS)* для перевода имен хостов, которые удобно запоминать людям (например, `www.univer.kharkov.ua`), в числовые адреса (например, `194.44.181.169`). Для работы с адресами в архитектуре *Java* есть специальные классы (напр., `java.net.InetAddress`).

Следует отметить, что в сети существуют как компьютеры, которые имеют фиксированные адреса (напр., серверы), так и компьютеры, которые получают разные адреса при каждой загрузке (напр., клиенты в локальных сетях). Технически, это никак не должно влиять на программы, написанные на *Java*. Важно только помнить, что *IP*-адреса могут измениться со временем (иногда, даже во время работы программы), и соответствующим образом писать программы (например, не сериализовать локальный *IP*-адрес при сохранении состояния приложения).

## Порты

Если бы каждый компьютер, присутствующий в сети, одновременно выполнял не более одной задачи, то только адреса вполне хватило бы для организации сетевой работы. Однако современные компьютеры выполняют много разных задач «одновременно». Такой режим сетевой реализуется с помощью такой абстракции, как *порт (port)*. Каждый компьютер с присвоенным *IP*-адресом содержит несколько тысяч логических портов (65 535 для протокола транспортного уровня). Это именно абстракция, не представляющая собой никакого реального, физического устройства, такого как, например, последовательный или параллельный порт. Каждый порт идентифицируется номером от 1 до 65535. Каждый порт может быть выделен для какой-либо конкретной службы (задачи).

Например, *HTTP*, базовый протокол *Интернета*, обычно использует порт 80. Компьютер - получатель данных анализирует каждый пакет, который приходит по его адресу; из полученных выбирает те, которые предназначены

для указанного порта; и отправляет данные программе, которая прослушивает указанный порт. Так сортируются разные типы трафика.

Номера портов от 1 до 1023 зарезервированы для служебного использования. В операционных системах семейства *Unix* только программы, работающие от имени суперпользователя *root*, могут получать данные из этих портов, но все программы могут отправлять данные в них. В *Windows* любая программа может использовать эти порты без особых ограничений. Существуют таблицы портов, которые обычно назначаются разным службам. Однако эти таблицы не всегда справедливы; в частности, веб-серверы часто работают на портах, отличных от 80, либо из-за того, что несколько серверов должны работать на одном компьютере, либо из-за того, что тот, кто установил сервер, не обладает правами *root*, необходимыми для его запуска на порту 80.

### Основные классы и интерфейсы

Рассмотрим некоторые основные классы и интерфейсы, которые используются при написании сетевых приложений.

#### Класс **InetAddress**

Для высокоуровневого представления *IP*-адреса в языке *Java* существует класс `java.net.InetAddress`. В первых версиях *Java* он был финальным (`final`), а в настоящее время имеет два подкласса для представления *IPv4* и *IPv6* адресов. Данный класс используется большинством других сетевых классов. Как правило, он инкапсулирует имя хоста (*hostname*), и *IP*-адрес.

У данного класса нет открытых (`public`) конструкторов. Вместо этого класс `InetAddress` предоставляет три статических (`static`) метода, которые возвращают корректно инициализированные объекты типа `InetAddress`. Приведем их:

```
public static InetAddress getByName(String hostName)
                                throws UnknownHostException

public static InetAddress[] getAllByName(String hostName)
                                throws UnknownHostException

public static InetAddress getLocalHost( ) throws UnknownHostException
```

При необходимости, каждый из этих методов может устанавливать соединение с локальным *DNS*-сервером для того, чтобы получить информацию, которая должна быть инкапсулирована в объекте `InetAddress`.

Поэтому, эти методы могут генерировать исключения, если подключение к *DNS*-серверу запрещено. Кроме того, в ряде случаев, вызов одного из этих методов может привести к установлению соединения с провайдером. Важно то, что эти методы не просто устанавливают значения внутренних полей, используя значения своих аргументов. Они, фактически, используя значения своих аргументов, устанавливают сетевые подключения для получения всей необходимой информации. Другие методы, существующие в этом классе, работают с той информацией, которая была собрана одним из этих трех методов.

Так как поиск *DNS* может быть относительно долгим (порядка нескольких секунд для запроса, который должен проходить через несколько промежуточных серверов, или того, который пытается разрешить недоступный хост), то класс *InetAddress* кэширует результаты поиска. Получив адрес определенного хоста, он не будет искать его снова, даже если вы создадите новый объект *InetAddress* для того же хоста.

В случае отрицательного результата (*host not found error*), когда первоначальная попытка разрешить хост не удалась, то сразу же следует следующая. Часто это происходит из-за того, информация еще не пришла с удаленного *DNS*-сервера. Как только адрес прибыл, так он сразу становится доступным для следующего запроса. Из-за этого *Java*, обычно, кэширует неудачные *DNS*-запросы в течение 10 секунд. В современных версиях *Java* этим временем можно управлять с помощью системных свойств *networkaddress.cache.ttl* и *networkaddress.cache.negative.ttl*. Параметр *networkaddress.cache.ttl* определяет время (количество секунд), в течение которых успешный поиск *DNS* будет оставаться в кэше *Java*. Параметр *networkaddress.cache.negative.ttl* определяет время (количество секунд), в течение которых неудачный поиск будет кэширован. Попытка найти тот же хост снова в течение этого времени приведет к тому же самому значению. Значение параметров -1 интерпретируется как «никогда не истекает».

Кроме такого локального кэширования классом *InetAddress*, еще и локальный хост, локальный сервер доменных имен и другие *DNS*-серверы в других частях *Интернета* могут также кэшировать результаты различных запросов. Архитектура *Java* не предоставляет возможности контролировать это. В результате распространение информации об изменении *IP*-адреса через *Интернет* может занять несколько часов. Тем временем программа может столкнуться с различными исключениями, в том числе *UnknownHostException*, *NoRouteToHostException* и *ConnectException*, в зависимости от изменений, внесенных в *DNS*.

Начиная с *Java 1.4*, в классе появились еще два фабричных метода, которые не проверяют свои адреса на локальном *DNS*-сервере. Первый создает



объект `InetAddress` с заданным *IP*-адресом и без указания имени хоста. Второй создает объект `InetAddress` с *IP*-адресом и именем хоста.

```
public static InetAddress getByAddress(byte[] address) throws
UnknownHostException
public static InetAddress getByAddress(String hostName, byte[] address)
throws UnknownHostException
```

В отличие от трех других фабричных методов, эти два метода не гарантируют, что хост с созданным адресом действительно существует, или что имя хоста правильно сопоставлено с *IP*-адресом. Они генерируют исключение `UnknownHostException`, только если в качестве аргумента адреса передан байтовый массив недопустимого размера (ни 4, ни 16 байтов).

Рассмотрим немного подробнее эти методы.

**`public static InetAddress getByName(String hostName) throws UnknownHostException`**

Метод `InetAddress.getByName()` является наиболее часто используемым из этих, ранее описанных фабричных методов. Это статический метод, который принимает в качестве аргумента имя искомого хоста. Метод ищет *IP*-адрес хоста, используя *DNS*. Данный метод генерирует исключение `UnknownHostException`, если хост не может быть найден, поэтому при вызове метода необходимо обработать это исключение, или его суперкласс, `IOException`, например, так:

```
try {
    InetAddress address =
        InetAddress.getByName("www.univer.kharkov.ua");
    System.out.println(address);
} catch (UnknownHostException ex) {
    System.out.println("Could not find www.univer.kharkov.ua");
}
```

Иногда бывает необходимо подключиться к машине, у которой нет имени хоста. В этом случае вы можете передать `InetAddress.getByName()` строку, содержащую четырехточечную или шестнадцатеричную форму *IP*-адреса с точками:

```
InetAddress address = InetAddress.getByName("194.44.181.169");
```

Когда метод `getByName()` вызывается с *IP*-адресом в виде строки в качестве аргумента, он создает объект `InetAddress` для указанного *IP*-адреса без проверки с помощью *DNS*. Это означает, что можно создавать объекты `InetAddress` для хостов, которые не существуют на самом деле, и к которым невозможно подключиться. Имя хоста для объекта `InetAddress`, созданного из строки, содержащей *IP*-адрес, изначально устанавливается на эту строку. *DNS* поиск для определения фактического имени хоста выполняется только при явном запросе имени хоста через вызов `getHostName()`, или неявно через вызов `toString()`. Если при запросе имени хоста, во время выполнения *DNS* поиска, хост с указанным *IP*-адресом не может быть найден, тогда имя хоста остается в исходной точечной строкой. При этом, однако, исключение `UnknownHostException` не генерируется.

В новых версиях *Java* (начиная с *Java 1.4*) метод `toString()` возвращает немного другой результат, чем в более ранних версиях. Метод не выполняет повторный поиск имени; таким образом, если во время вызова метода имя хоста неизвестно, то оно и не печатается.

Имена хостов (*hostname*) являются намного более стабильными, чем *IP*-адреса. Есть сетевые службы, которые годами имели одно и то же имя хоста, но много раз меняли *IP*-адреса. При написании сетевых программ, при наличии выбора между использованием имени хоста, например `www.univer.kharkov.ua`, или *IP*-адресом, например `194.44.181.169`, рекомендуется всегда выбирать имя хоста. Использовать *IP*-адрес рекомендуется только в крайних случаях, когда имя хоста неизвестно.

**public static InetAddress[] getAllByName(String hostName)  
throws UnknownHostException**

Некоторые компьютеры могут иметь более одного *Интернет*-адреса. Принимая в качестве параметра имя хоста, метод `InetAddress.getAllByName()` возвращает массив, который содержит все адреса, соответствующие этому имени. Данный метод, также как и рассмотренный ранее метод `InetAddress.getByName()`, может выбрасывать исключение `UnknownHostException`, которое необходимо или передать дальше, или обработать на месте.

Рассторим фрагмент кода, показывающий применение метода:

```
try {
```

```

InetAddress[] hosts =
    InetAddress.getAllByName("www.google.com");
    System.out.println("Hosts: " +
        java.util.Arrays.toString(hosts));

} catch (UnknownHostException ex) {
    System.err.println(ex);
}

```

Вообще-то, хосты с более чем одним адресом являются скорее исключением, чем правилом. Большинство хостов с несколькими *IP*-адресами являются веб-серверами. Но, даже для этих компьютеров редко бывает нужно знать более одного адреса.

```

public static InetAddress getByAddress(byte[ ] address)
throws UnknownHostException // Java 1.4
public static InetAddress getByAddress(
String hostName, byte[] address)
throws UnknownHostException // Java 1.4

```

Начиная с *Java 1.4* метод `getByAddress()` может принять в качестве аргумента байтовый массив и, кроме того, необязательное имя хоста. В этом случае поиск доменного имени не выполняется. Однако, если байтовый массив имеет неверную длину (отличную от 4 или 16 байт), и, поэтому, массив не может быть адресом ни *IPv4*, ни *IPv6* формата, то в этом случае генерируется исключение `UnknownHostException`. Такая форма вызова бывает полезна, в случае если сервер доменных имен или недоступен, или содержит неточную информацию. С помощью этих методов можно создавать объекты `InetAddress`, которые не соответствуют ни одному реально существующему адресу.

```

public static InetAddress getLocalHost()
throws UnknownHostException

```

В классе `InetAddress` есть еще один метод для получения объекта `InetAddress`. Это статический метод `InetAddress.getLocalHost()`, возвращающий объект `InetAddress` для компьютера, на котором он вызван. Так же, как и ранее описанные методы, этот метод генерирует исключение `UnknownHostException`. Это исключение выбрасывается в случае, когда

метод не может найти адрес локальной машины (хотя эта ситуация никогда не должна произойти):

```
try {
    InetAddress localhost = InetAddress.getLocalHost();
    System.out.println("Local host address: " + localhost);
} catch (UnknownHostException ex) {
    System.err.println(ex);
}
```

В результате, можно получить разные результаты:

```
//Local host address: TITAN/127.0.0.1
//Local host address: Sporov/192.168.205.53
```

То, что будет выведено на экран в результате работы программы зависит от ситуации. Можно увидеть или полное, или частичное имя, например, *Sporov*; это зависит от того, что именно локальный *DNS*-сервер вернет для хоста. Если компьютер к Интернету не подключен, и не имеет фиксированного *IP*-адреса или имени, то, скорее всего, будет выведено *localhost* в качестве имени хоста и *127.0.0.1* в качестве *IP*-адреса.

## Проблемы с безопасностью

Отметим кратко проблемы, сопровождающие создание нового объекта *InetAddress*. Такая операция по созданию объекта *InetAddress* по имени хоста считается потенциально небезопасной операцией, поскольку требует *DNS* поиска. Недоверенному коду, полученному по сети (напр. *апплету*) под управлением диспетчера безопасности по умолчанию будет разрешено получать только *IP*-адрес хоста, с которого он получен (его кодовая база), и, возможно, локальный хост. Недоверенный код не позволяет создавать объект *InetAddress* из любого другого имени хоста, т.к. недоверенный код не может выполнять поиск *DNS* для сторонних хостов из-за запрета на сетевые подключения к хостам, отличным от кодовой базы. Сейчас эти проблемы рассматривать не будет. Немного об этих проблемах поговорим при рассмотрении простых распределенных приложений.

## Методы доступа

В классе *InetAddress* содержится три метода-геттера, которые возвращают имя хоста в строковом виде, *IP*-адрес в строковом виде и в виде байтового массива:

```
public String getHostName()
public byte[] getAddress()
```

```
public String getHostAddress()
```

При этом, в классе нет соответствующих методов-сеттеров. Это значит, что после создания не получится изменить значение полей созданного объекта что делает `InetAddress` неизменным и, следовательно, потокобезопасным.

### **public String getHostName()**

Метод `getHostName()` возвращает строку, содержащую имя хоста с *IP*-адресом, представленным этим объектом `InetAddress`. Если у рассматриваемой машины нет имени хоста или если `SecurityManager` препятствует определению имени, то возвращается числовой точечный формат *IP*-адреса. Вид результата зависят от того, как локальный сервер *DNS* ведет себя при разрешении локальных имен.

### **public String getHostAddress()**

Данный метод возвращает строку, содержащую числовой точечный формат *IP*-адреса.

### **public byte[] getAddress()**

Данный метод возвращает *IP*-адрес компьютера в виде массива байтов, расположенных в сетевом порядке. Самый старший байт (первый байт в точечной нотации) - это первый байт в массиве (его нулевой элемент). Так как возможно возвращение как адресов типа *IPv4*, так и адресов формата *IPv6*, то не нужно делать предположений о длине этого массива. Если нужно узнать длину массива, то пользуйтесь стандартным подходом.

Байты, возвращенные методом, вообще-то относятся к беззнаковому типу, и это проблема, так как в *Java* нет примитивных беззнаковых типов. Поэтому, по умолчанию, значения больше 127 воспринимаются как отрицательные числа. Поэтому, прежде чем что-то делать с полученными байтами, нужно выполнить операцию преобразования типа: от `byte` к `int`. Например так:

```
int unsignedByte = signedByte < 0 ? signedByte + 256 : signedByte;
```

Данный метод может применяться для определения типа адреса: *IPv4* или *IPv6*.

```
public static int getVersion(InetAddress inetAddress) {
    byte[] address = inetAddress.getAddress();
```

```

    if (address.length == 4) return 4;
    else if (address.length == 16) return 6;
    else return -1;
}

```

## Типы адресов

Некоторые сетевые *IP*-адреса и их группы имеют особое значение. Например, адрес 127.0.0.1 является локальным адресом обратной связи (англ. *loopback*). Адреса формата *IPv4* в диапазоне от 224.0.0.0 до 239.255.255.255 - это многоадресные адреса (*multicast addresses*). Класс *InetAddress* содержит целый набор методов для проверки типа адреса объекта *InetAddress*:

```

public boolean isAnyLocalAddress( )
public boolean isLoopbackAddress( )
public boolean isLinkLocalAddress( )
public boolean isSiteLocalAddress( )
public boolean isMulticastAddress( )
public boolean isMCGlobal( )
public boolean isMCNodeLocal( )
public boolean isMCLinkLocal( )
public boolean isMCSiteLocal( )
public boolean isMCOrgLocal( )

```

### **public boolean isAnyLocalAddress( )**

Этот метод возвращает *true*, если адрес является *wildcard address*, в противном случае - *false*. Такой адрес соответствует любому локальному адресу. Такой адрес бывает важен для компьютеров, имеющих несколько сетевых интерфейсов, что характерно для серверов. В *IPv4 wildcard address* равен 0.0.0.0. В *IPv6* этот адрес составляет 0:0:0:0:0:0:0:0 (или : :).

### **public boolean isLoopbackAddress( )**

Этот метод возвращает значение *true*, если адрес является *loopback*, в противном случае - значение *false*. С помощью такого адреса происходит подключение к тому же самому компьютеру напрямую на *IP* уровне, без использования какого-либо физического оборудования. Обычно используется для тестирования. Подключение к *loopback* адресу отличается от подключения

к обычному *IP*-адресу. В *IPv4* этот адрес 127.0.0.1. В *IPv6* этот адрес равен 0:0:0:0:0:0:0:1 (или :: 1).

### **public boolean isLinkLocalAddress( )**

Этот метод возвращает *true*, если адрес является *link-local address IPv6*, в противном случае - *false*. Этот адрес используется для самоконфигурации сетей *IPv6 (IPv6 networks self-configure)*, аналогично *DHCP* в сетях *IPv4*, но без использования сервера. При этом маршрутизаторы не пересылают такие пакеты за пределы локальной подсети. Все *link-local address* адреса начинаются с восьми байтов FE80:0000.0000:0000. Следующие восемь байт содержат локальный адрес, который часто копируется с *MAC*-адреса *Ethernet*.

### **public boolean isSiteLocalAddress( )**

Этот метод возвращает значение *true*, если адрес является *link-local address IPv6*, в противном случае - значение *false*. *Site-local* адреса аналогичны *link-local* адресам, за исключением того, что они могут пересылаться маршрутизаторами кампуса, но не должны выходить за пределы этой сети. *Site-local* адреса начинаются с восьми байтов FEC0:0000.0000:0000. Следующие восемь байт заполняются адресом, который часто копируется с *MAC*-адреса *Ethernet*.

### **public boolean isMulticastAddress( )**

Этот метод возвращает *true*, если адрес является многоадресным (*multicast*), в противном случае - *false*. Многоадресная рассылка транслирует контент не на один конкретный компьютер, а на все подписанные компьютеры. В *IPv4* все многоадресные адреса находятся в диапазоне от 224.0.0.0 до 239.255.255.255. В *IPv6* все они начинаются с байта FF. Многоадресную рассылку рассмотрим чуть позже.

### **public boolean isMCGlobal( )**

Этот метод возвращает *true*, если адрес является глобальным многоадресным адресом (*global multicast address*), в противном случае - *false*. Такой адрес может иметь «подписчиков» по всему миру. Все многоадресные адреса начинаются с FF. В *IPv6 global multicast address* начинаются с FF0E или FF1E в зависимости от того, является ли адрес многоадресной рассылки общеизвестным постоянно назначенным адресом (*permanently assigned address*) или временным (*transient*) адресом. В *IPv4* все *multicast* адреса имеют глобальную область.

### **public boolean isMCOrgLocal( )**

Этот метод возвращает *true*, если адрес является *organization-wide multicast address*, в противном случае - *false*. Такие адреса могут иметь подписчиков на всех сайтах компании или организации, но не за пределами этой организации. Адреса такого типа начинаются с FF08 или FF18, в зависимости от того, является ли адрес многоадресной рассылки *well known permanently assigned address* или *transient address*.

### **public boolean isMCSiteLocal( )**

Этот метод возвращает *true*, если адрес является *site-wide multicast address*, в противном случае - *false*. Пакеты, направленные на этот адрес, будут передаваться только в пределах локального сайта. Такие адреса начинаются с FF05 или FF15, в зависимости от того, является ли адрес многоадресной рассылки *well known permanently assigned address* или *transient address*.

### **public boolean isMCLinkLocal( )**

Этот метод возвращает *true*, если адрес является *subnet-wide multicast address*, в противном случае - *false*. Пакеты, адресованные на этот адрес, будут передаваться только в пределах их собственной подсети. Такие адреса начинаются с FF02 или FF12, в зависимости от того, является ли адрес многоадресной рассылки *a well known permanently assigned address* или *a transient address*.

### **public boolean isMCNodeLocal( )**

Этот метод возвращает *true*, если адрес является *an interface-local multicast address*, в противном случае - *false*. Пакеты, направленные по этому адресу, не выходят за пределы сетевого интерфейса, который их выбросил, даже на другой сетевой интерфейс на того же самого узла. Эта возможность полезна для отладки и тестирования сети. *Interface-local multicast address* начинаются с двух байтов FF01 или FF11 в зависимости от того, является ли *multicast address is a well known permanently assigned address* или *a transient address*.

## **Проверка доступности**

В *Java 1.5* в класс *InetAddress* было добавлено два новых метода, которые позволяют проверить, доступен ли конкретный узел с текущего хоста; то есть, может ли быть установлено сетевое соединение. Соединения могут быть заблокированы по многим причинам: брандмауэры, прокси-серверы, ненадлежащее поведение маршрутизаторов и поврежденные кабели,



удаленный хост не включен и т.д. Два метода `isReachable()` позволяют проверить соединение:

```
public boolean isReachable(int timeout) throws IOException
public boolean isReachable(
    NetworkInterface interface, int ttl, int timeout)
    throws IOException
```

Эти методы пытаются установить подключение к эхо-порту на удаленном хост-сайте, чтобы выяснить, достижим ли он. (для *TCP* и *UDP* протоколов это порт с номером 7). Если хост отвечает в течение заданного времени (в миллисекундах), то методы возвращают `true`; в противном случае они возвращают `false`. В случае возникновения сетевой ошибки будет выброшено исключение `IOException`. Второй вариант метода позволяет указать локальный сетевой интерфейс, из которого устанавливается соединение, и «время жизни» (*time-to-live*) (максимальное количество «сетевых прыжков», (*network hops*) которые будет выполнено соединением, прежде чем оно будет сброшено).

Вообще-то, обычно, эти методы дают не очень надежные результаты для глобальной сети *Интернет*. Как правило, брандмауэры, мешают сетевым протоколам, которые *Java* использует для определения доступности хоста. При работе в локальной сети эти методы надежны.

### Метод, унаследованный от `Object`

Как и любой другой класс, `java.net.InetAddress` является наследником от `java.lang.Object`. Таким образом, он имеет доступ ко всем методам этого класса. Он переопределяет три метода для обеспечения более специализированного поведения:

```
public boolean equals(Object o)
public int hashCode( )
public String toString( )
```

### **public boolean equals(Object o)**

Объекты `InetAddress` равны, только если они оба являются экземплярами класса `InetAddress` и имеет тот же *IP*-адрес. При этом хостам не нужно иметь одно и то же имя. Таким образом, объект `InetAddress` для *www.ibiblio.org* равен объекту `InetAddress` для *www.cafeaulait.org*, поскольку оба имени ссылаются на один и тот же *IP*-адрес.

Рассмотрим небольшой пример использования этих методов:

```
package themel;

import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class InetAddress1 {
    public static void main(String[] args) {
        InetAddress localhost = null;
        InetAddress host = null;
        InetAddress[] hosts = null;
        InetAddress host1 = null;
        try {
            localhost = InetAddress.getLocalHost();
            host = InetAddress.getByName("www.univer.kharkov.ua");
            hosts = InetAddress.getAllByName("www.microsoft.com");
            host1 = InetAddress.getByAddress(new byte[] {(byte)8, (byte)8, (byte)8,
(byte)8});
        } catch (UnknownHostException ex) {
            System.err.println(ex);
        }
        System.out.println("Local host address: " + localhost);
        //Without Internet Connection:
        //Local host address: TITAN/127.0.0.1
        //Host: null
        //Hosts: null
        System.out.println("Host: " + host);
        System.out.println("Hosts: " + java.util.Arrays.toString(hosts));
        System.out.println("Hosts:");
        for (int i = 0; i < hosts.length; i++) {
            System.out.println("\t" + (i+1) + ") . " + hosts[i]);
        }
        System.out.println("Host1: " + host1);
        //With internet connection:
        //Local host address: TITAN/192.168.1.4
        //Host: www.univer.kharkov.ua/194.44.181.173
        //Hosts: [www.google.com/173.194.122.176,
//www.google.com/173.194.122.177, www.google.com/173.194.122.178,
//www.google.com/173.194.122.179, www.google.com/173.194.122.180]
        //Host1: /8.8.8.8

        //Some getters
        if (host1 != null) {
            System.out.println("Host1: " + host1.getHostName()); //google-public-dns-
a.google.com
            System.out.println("Host1: " + host1.getHostAddress()); //8.8.8.8
            System.out.println("Host1: " +
java.util.Arrays.toString(host1.getAddress())); //[8, 8, 8, 8]
        }

        //Type of Address
        InetAddress address = localhost;
        if (address.isAnyLocalAddress()) {
            System.out.println(address + " is a wildcard address.");
        }
        if (address.isLoopbackAddress()) {
            System.out.println(address + " is loopback address.");
        }
        if (address.isLinkLocalAddress()) {
            System.out.println(address + " is a link-local address.");
        } else if (address.isSiteLocalAddress()) {
```

```

        System.out.println(address + " is a site-local address.");
    } else {
        System.out.println(address + " is a global address.");
    }
    if (address.isMulticastAddress()) {
        if (address.isMCGlobal()) {
            System.out.println(address + " is a global multicast address.");
        } else if (address.isMCOrgLocal()) {
            System.out.println(address
                + " is an organization wide multicast address.");
        } else if (address.isMCSiteLocal()) {
            System.out.println(address + " is a site wide multicast address.");
        } else if (address.isMCLinkLocal()) {
            System.out.println(address + " is a subnet wide multicast address.");
        } else if (address.isMCNodeLocal()) {
            System.out.println(address
                + " is an interface-local multicast address.");
        } else {
            System.out.println(address + " is an unknown multicast address type.");
        }
    } else {
        System.out.println(address + " is a unicast address.");
    }
}

try {
    System.out.println("is univer reachable: " + host.isReachable(100) ); //10
- false, 100 - true
//public boolean isReachable(NetworkInterface interface, int ttl, int timeout)
//throws IOException
} catch (IOException ex) {
    System.err.println(ex);
}

try {
    InetAddress ibiblio = InetAddress.getByName("www.ibiblio.org");
    InetAddress helios = InetAddress.getByName("helios.metalab.unc.edu");
    if (ibiblio.equals(helios)) {
        System.out.println("www.ibiblio.org is the same as
helios.metalab.unc.edu");
    } else {
        System.out.println("www.ibiblio.org is not the same as
helios.metalab.unc.edu");
    }
} catch (UnknownHostException ex) {
    System.out.println("Host lookup failed.");
}
}
}

```

## Класс java.net.NetworkInterface

Архитектура *Java 1.4* включает класс `NetworkInterface`, который также представляет локальный *IP*-адрес. Это может быть либо физический интерфейс, такой как дополнительная карта *Ethernet*, либо виртуальный интерфейс, связанный с тем же физическим оборудованием, что и другие *IP*-адреса машины. Класс `NetworkInterface` содержит методы для перечисления всех локальных адресов независимо от интерфейса и для создания объектов

`InetAddress` из них. Эти объекты `InetAddress` могут затем использоваться для создания сокетов, серверных сокетов и так далее.

### Фабричные методы

Поскольку объекты типа `NetworkInterface` представляют физическое оборудование и виртуальные адреса, они не могут быть просто созданы. Как и в случае класса `InetAddress`, для этого предназначены статические фабричные методы, которые возвращают объект `NetworkInterface`, связанный с конкретным сетевым интерфейсом. Можно создать объект класса `NetworkInterface` по *IP*-адресу или по имени.

#### **public static NetworkInterface getBy\_name(String name) throws SocketException**

Метод `getBy_name()` возвращает объект `NetworkInterface`, представляющий сетевой интерфейс с конкретным именем. Если интерфейса с таким именем нет, то метод возвращает `null`. Если базовый сетевой стек сталкивается с проблемой при поиске соответствующего сетевого интерфейса, то генерируется исключение `SocketException`, но обычно эта ситуация маловероятна.

Формат имен сетевых интерфейсов зависит от платформы. В типичной системе *Unix* имена интерфейсов *Ethernet* обычно такие: `eth0`, `eth1` и т.д. Локальный адрес обратной свяи обычно называется «`lo`». В операционной системе *Windows* имена могут формироваться как строки, полученные из имени поставщика и модели оборудования в данном конкретном сетевом интерфейсе.

#### **public static NetworkInterface getByInetAddress( InetAddress address) throws SocketException**

Метод `getByInetAddress()` возвращает объект типа `NetworkInterface`, представляющий сетевой интерфейс, связанный с указанным *IP*-адресом. Если ни один сетевой интерфейс не связан с этим *IP*-адресом на локальном хосте, то метод возвращает `null`. В случае какой-либо проблем генерируется исключение `SocketException`.

#### **public static Enumeration getNetworkInterfaces( ) throws SocketException**

Метод `getNetworkInterfaces()` возвращает `java.util.Enumeration`, перечисляющий все сетевые интерфейсы на локальном хосте. Рассмотрим небольшой пример, демонстрирующий применение метода:

```
import java.net.*;
import java.util.*;

public class InterfaceLister {
    public static void main(String[] args) throws Exception {
        Enumeration interfaces =
            NetworkInterface.getNetworkInterfaces( );
        while (interfaces.hasMoreElements( )) {
            NetworkInterface ni =
                (NetworkInterface) interfaces.nextElement( );
            System.out.println(ni);
        }
    }
}
```

Посмотрите на результат выполнения программы в Вас на компьютере.

### Методы доступа

Получив объект `NetworkInterface`, вы можете узнать его *IP*-адрес и имя.

#### **public Enumeration getInetAddresses( )**

Бывает так, что один сетевой интерфейс связан с более чем одним *IP*-адресом. Метод `getInetAddresses()` возвращает `java.util.Enumeration`, содержащий объект `InetAddress` для каждого *IP*-адреса, с которым связан этот интерфейс. Рассмотрим фрагмент кода перечисляет все *IP*-адреса для интерфейса `eth0`:

```
NetworkInterface eth0 = NetworkInterface.getBy-name("eth0");
Enumeration addresses = eth0.getInetAddresses( );
while (addresses.hasMoreElements( )) {
    System.out.println(addresses.nextElement( ));
}
```

#### **public String getName( )**

Метод `getName()` возвращает имя объекта `NetworkInterface`, например `eth0` или `lo`.

## **public String getDisplayName( )**

Метод `getDisplayName()` возвращает имя интерфейса конкретного `NetworkInterface`, в виде, более понятном для человека - что-то вроде «*Ethernet Card 0*». Эта строка сильно зависил от операционной системы.

## **Методы, унаследованные от Object**

Два объекта `NetworkInterface` считаются равными, если они представляют один и тот же физический сетевой интерфейс (например, оба указывают на один и тот же порт *Ethernet*, модем или беспроводную карту) и имеют один и тот же *IP*-адрес. В противном случае они не равны.

Ниже приведены некоторые примеры, показывающие особенности применения рассмотренных методов.

```
package theme1;

import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Enumeration;

public class InetAddress2 {

    public static void main(String[] args) {
        try {
            NetworkInterface ni = NetworkInterface.getBy_name("eth0");
            if (ni == null) {
                System.err.println("No such interface: eth0");
            } else {
                System.out.println("ni: " + ni); //ni: name:eth0 (WAN Miniport (IPv6))
            }
        } catch (SocketException ex) {
            System.err.println("Could not list sockets.");
        }

        ///////////////////////////////////////////////////
        try {
            InetAddress local = InetAddress.getByName("127.0.0.1");
            NetworkInterface ni = NetworkInterface.getByInetAddress(local);
            if (ni == null) {
                System.err.println("That's weird. No local loopback address.");
            } else {
                System.out.println("ni: " + ni); //ni: name:lo (Software Loopback
Interface 1)
            }
        } catch (SocketException ex) {
            System.err.println("Could not list sockets.");
        } catch (UnknownHostException ex) {
            System.err.println("That's weird. No local loopback address.");
        }

        //list all network interfaces
```

```

try {
    Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
    while (interfaces.hasMoreElements()) {
        NetworkInterface ni = (NetworkInterface) interfaces.nextElement();
        System.out.println(ni);
    }
} catch (SocketException ex) {
    System.err.println(ex);
}

/*
name:net0 (WAN Miniport (SSTP))
name:net1 (WAN Miniport (L2TP))
name:net2 (WAN Miniport (PPTP))
name:ppp0 (WAN Miniport (PPPOE))
name:eth0 (WAN Miniport (IPv6))
name:eth1 (WAN Miniport (Network Monitor))
name:eth2 (WAN Miniport (IP))
name:ppp1 (RAS Async Adapter)
name:eth3 (Realtek PCIe GBE Family Controller)
name:net3 (Teredo Tunneling Pseudo-Interface)
name:net4 (WAN Miniport (IKEv2))
name:eth4 (VirtualBox Host-Only Ethernet Adapter)
name:net5 (Адаптер Microsoft ISATAP #2)
name:net6 (Адаптер Microsoft ISATAP)
name:eth5 (Realtek PCIe GBE Family Controller-VirtualBox NDIS Light-Weight Filter-0000)
name:eth6 (Realtek PCIe GBE Family Controller-QoS Packet Scheduler-0000)
name:eth7 (Realtek PCIe GBE Family Controller-WFP LightWeight Filter-0000)
name:eth8 (WAN Miniport (IPv6)-QoS Packet Scheduler-0000)
name:eth9 (WAN Miniport (IP)-QoS Packet Scheduler-0000)
name:eth10 (WAN Miniport (Network Monitor)-QoS Packet Scheduler-0000)
name:eth11 (VirtualBox Host-Only Ethernet Adapter-VirtualBox NDIS Light-Weight Filter-0000)
name:eth12 (VirtualBox Host-Only Ethernet Adapter-QoS Packet Scheduler-0000)
name:eth13 (VirtualBox Host-Only Ethernet Adapter-WFP LightWeight Filter-0000)
*/

//interface getter methods
try {
    NetworkInterface eth4 = NetworkInterface.getBy_name("eth4");
    System.out.println("name: " + eth4.getName()); // name: eth4
    System.out.println("display name: " + eth4.getDisplayName()); // display
name: VirtualBox Host-Only Ethernet Adapter
    Enumeration addresses = eth4.getInetAddresses();
    while (addresses.hasMoreElements()) {
        System.out.println(addresses.nextElement());
    }
} catch (SocketException ex) {
    System.err.println(ex);
}

// 192.168.56.1
// fe80:0:0:0:1ca4:7952:36b8:ad5d%eth4
}
}

```

Рассмотрим пример программы, перечисляющей все сетевые интерфейсы и, в случае наличия, их под-интерфейсы.

```

package test1;

import java.net.*;
import java.util.*;

```

```

class GetNetworkInterfaces {
    public static void main(String args[]) throws Exception {
        Enumeration<NetworkInterface> intfs =
            NetworkInterface.getNetworkInterfaces();
        while (intfs.hasMoreElements()) {
            NetworkInterface intf = intfs.nextElement();
            System.out.println("\nInterface: " + intf.getName());
            System.out.println("Display name: " + intf.getDisplayName());
            Enumeration<NetworkInterface> subIfs = intf.getSubInterfaces();
            for (NetworkInterface subIf : Collections.list(subIfs)) {
                System.out.printf("\tSub Interface : " + subIf.getName());
                System.out.printf("\tSub Interface Display
name: "+subIf.getDisplayName());
            }
        }
    }
}

```

Каждому сетевому интерфейсу присваивается один или несколько *IP*-адресов. *IP*-адрес программно представлен классом `InetAddress`. Иногда необходимо знать *IP*-адрес интерфейса. Это можно сделать с помощью метода `getInetAddresses()`, который возвращает перечисление `InetAddress`. Выведем на экран *IP*-адреса тех интерфейсов, которым назначен хотя бы один *IPv4*-адрес.

```

package test1;

import java.net.*;
import java.util.*;

class GetInterfaceAddresses {
    public static void main(String args[]) throws Exception {
        System.setProperty("java.net.preferIPv4Stack", "true");
        Enumeration<NetworkInterface> intfs =
            NetworkInterface.getNetworkInterfaces();
        while (intfs.hasMoreElements()) {
            NetworkInterface intf = intfs.nextElement();
            Enumeration<InetAddress> addresses = intf.getInetAddresses();
            if (addresses.hasMoreElements()) {
                System.out.println("\nName: " + intf.getName());
                System.out.println("Display name: " + intf.getDisplayName());
                while (addresses.hasMoreElements()) {
                    InetAddress addr = addresses.nextElement();
                    System.out.println("Address: " + addr);
                }
            }
        }
    }
}

```

Чтобы получить *IPv6* адреса, нужно закомментировать строку

```
System.setProperty("java.net.preferIPv4Stack", "true");
```



Как уже было отмечено, класс `NetworkInterface` содержит методы для получения информации о сетевом интерфейсе. Рассмотрим простую программу, отображающую информацию о сетевых интерфейсах, имя которых указано в качестве аргумента командной строки.

```
package test1;

import java.net.*;
import java.util.*;

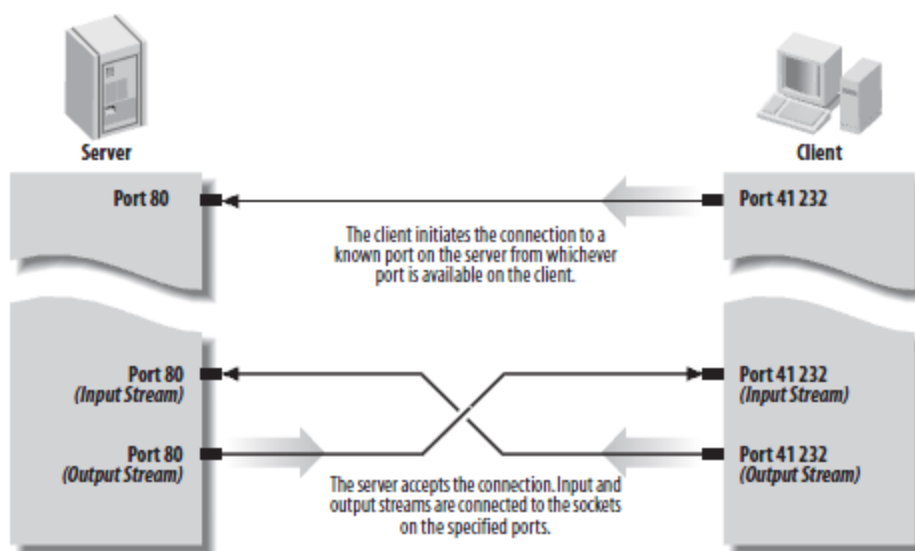
class GetInterfaceParameters {
    public static void main(String args[]) throws Exception {
        NetworkInterface intf = NetworkInterface.getByByName(args[0]);
        System.out.println("\nName : " + intf.getName());
        System.out.println("Display name : " + intf.getDisplayName());
        System.out.println("Up : " + intf.isUp());
        System.out.println("Loopback : " + intf.isLoopback());
        System.out.println("PointToPoint : " + intf.isPointToPoint());
        System.out.println("Supports multicast : " + intf.supportsMulticast());
        System.out.println("Virtual : " + intf.isVirtual());
        byte[] mac1 = intf.getHardwareAddress();
        if (mac1 != null) {
            System.out.print("Hardware Address : ");
            for (int k = 0; k < mac1.length; k++)
                System.out.format("%02X%s", mac1[k], (k < mac1.length - 1) ? "-" : "");
            System.out.println();
        }
        System.out.println("MTU : " + intf.getMTU());
    }
}
```

## Программная работа с TCP протоколом

Для организации сетевого взаимодействия часто используется модель «клиент—сервер» (англ. *client-server*) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми *серверами*, и заказчиками услуг, называемыми *клиентами*. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных или в виде сервисных функций. Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, ее также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

*Java* содержит классы, которые позволяют устанавливать соединение с удаленным компьютером, используя протокол *TCP*. Протокол *TCP* является протоколом, ориентированным на установление соединения, которое гарантирует надежную связь между приложениями клиента и сервера. Взаимодействие с использованием протокола *TCP*, начинается после установления соединения между сокетами клиента и сервера. Сокет сервера «слушает» запросы на установление соединения, отправленные сокетами клиентов, и устанавливает соединение. После установления соединения между приложениями клиента и сервера, они могут взаимодействовать друг с другом.

*Java* упрощает сетевое программирование, за счет инкапсуляции функциональности соединения сокета *TCP* в классы сокета, в которых класс *Socket* предназначен для создания сокета клиента, а класс *ServerSocket* для создания сокета сервера. Схема взаимодействия представлена на рисунке:



## Класс **ServerSocket**

Для написания серверов *Java* содержит класс *ServerSocket*. Серверный сокет создается на сервере и предназначен для прослушивания входящих *TCP*-соединений. Каждый серверный сокет прослушивает определенный порт на сервере. Когда клиент, запущенный на удаленном хосте, пытается подключиться к этому порту компьютера, серверный сокет «активируется» - выходит из состояния ожидания клиентов, согласовывает соединение между клиентом и сервером и возвращает объект типа *Socket*, предназначенный для работы с клиентом. Таким образом, серверные сокеты ожидают подключения, а клиентские сокеты иницируют подключения. Как только *ServerSocket* установил соединение, сервер использует обычный объект типа *Socket* для отправки данных клиенту.

Класс `ServerSocket` содержит все необходимое для написания серверов на *Java*. Мы рассмотрим самые часто используемые методы, а более подробно с этим классом можно познакомиться в документации:

<https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html>.

Необработанные входящие запросы на соединение, адресованные определенному порту, сохраняются в очереди, длина которой по умолчанию обычно составляет 50. После того, как очередь заполняется необработанными соединениями, хост отказывает в дополнительных подключениях к этому порту, пока не освободится место в очереди. Класс `ServerSocket` содержит конструкторы, которые позволяют регулировать длину очереди; однако нельзя увеличить длину очереди сверх максимального размера, поддерживаемого операционной системой.

### Конструкторы

Рассмотрим некоторые открытые конструкторы, которые содержатся в классе `ServerSocket`.

#### **`public ServerSocket(int port) throws BindException, IOException`**

Этот конструктор создает серверный сокет, и привязывает его к порту, указанному аргументом `port`. Если в качестве номера порта будет указан 0, то система сама выберет свободный порт. Такой порт, автоматически выбранный системой, иногда называют *анонимным портом* (*anonymous port*), поскольку его номер заранее неизвестен. Эта возможность не очень полезна, поскольку клиенты должны заранее знать, к какому порту следует подключаться.

Конструктор может выбрасывать `IOException` (в частности, `BindException`), если сокет не может быть создан и привязан к указанному порту. Исключение `IOException` при создании `ServerSocket` означает проблемы с портом: либо он уже используется другим серверным сокетом, возможно, из совершенно другой программы, либо в операционной системе с ограничениями доступа программа пытается подключиться к порту от 1 до 1023 без привилегий *суперпользователя* (*root*).

#### **`public ServerSocket(int port, int queueLength)` **`throws IOException, BindException`****

Этот конструктор создает серверный сокет, привязанный к указанному порту с выбранной длиной очереди. Если машина имеет несколько сетевых интерфейсов или *IP*-адресов, она прослушивает этот порт на всех этих интерфейсах и *IP*-адресах. Аргумент `queueLength` устанавливает длину очереди для входящих запросов на соединение (количество входящих

соединений ожидающих подключения, прежде чем хост начнет отказывать в соединении). Если при создании сокета указать длину больше, чем поддерживает операционная система, то, будет использована максимальная длина очереди. Если в качестве номера порта будет указан 0, то будет выбран какой-то из свободных портов.

**public ServerSocket(int port, int queueLength,  
InetAddress bindAddress)  
throws BindException, IOException**

Этот конструктор создает серверный сокет, который привязан к указанному порту `port` с указанной длиной очереди `queueLength`. Он отличается от двух других конструкторов тем, что созданный сокет привязывается только к указанному локальному *IP*-адресу. Этот конструктор полезен для серверов, работающих в системах с несколькими *IP*-адресами, поскольку с его помощью можно выбрать адрес, только который будет прослушиваться. Предыдущие два конструктора создавали сокеты, которые были привязаны по умолчанию со всеми локальными *IP*-адресами.

### Методы принятия и закрытия соединений

Объект типа `ServerSocket` обычно используется в цикле приема соединений. На каждой итерации цикла вызывается метод `accept()`. Этот вызов возвращает объект `Socket`, представляющий соединение между удаленным клиентом и локальным сервером. Взаимодействие с этим клиентом происходит через этот объект `Socket`. Когда взаимодействие завершено, сервер должен вызвать метод `close()` на объекте `Socket`. Если клиент закрывает соединение, в то время когда сервер с ним еще работает, то *поток ввода / вывода*, которые соединяют сервер с клиентом, выбрасывают исключение `InterruptedIOException` при следующей операции чтения / записи. При этом серверный сокет готов к приему и обработке следующих подключений. Когда серверу нужно завершить работу и не обрабатывать последующие входящие подключения, необходимо вызвать метод `close()` объекта `ServerSocket`.

**public Socket accept( ) throws IOException**

После создания и настройки серверного сокета нужно быть готовым к приему соединений от клиентов. Для этого нужно вызвать метод `accept()` объекта `ServerSocket`. Это блокирующий метод, то есть он останавливает поток выполнения и ожидает подключения клиента. Когда клиент подключается, метод `accept()` возвращает объект `Socket`. Для взаимодействия

с клиентом используются потоки, возвращенные методами этого объекта `Socket`: `getInputStream()` и `getOutputStream()`.

Когда обрабатываются исключения, код становится достаточно громоздким. При этом важно различать исключения, которые, должны завершать работу сервера и вывести сообщение об ошибке, и исключения, которые должны просто закрывать активное соединение. Исключения, выдаваемые методом `accept()`, а также *входными* и *выходными* потоками, обычно не должны завершать работу сервера. Большинство других исключений, скорее всего, должны. Для придется использовать вложенные *try*-блоки.

При завершении работы сервера необходимо закрыть все сокеты, соответствующие приняты соединениям. Для этого вызов `close()` также должен быть заключен в *try*-блок, который перехватывает `IOException`. При этом, если перехвачено исключение `IOException` при закрытии сокета, то можно просто игнорировать его. Это означает, что клиент закрыл сокет раньше, чем это сделал сервер. Рассмотрим, схематично, пример:

```
try {
    ServerSocket server = new ServerSocket(5776);
    while (true) {
        Socket connection = server.accept( );
        try {
            Writer out = new
                OutputStreamWriter(
                    connection.getOutputStream( ));
            out.write("You've connected to this server. Bye-bye now.\r\n");
            out.flush( );
            connection.close( );
        } catch (IOException ex) {
            // This tends to be a transitory error for this one connection;
            // e.g. the client broke the connection early. Consequently,
            // you don't want to break the loop or print an error message.
            // However, you might choose to log this exception in an error log.
        } finally {
            // Guarantee that sockets are closed when complete.
            try {
                if (connection != null) connection.close( );
            } catch (IOException ex) {}
        }
    }
}
```

```

    }
} catch (IOException ex) {
    System.err.println(ex);
}

```

### **public void close( ) throws IOException**

По завершении работы с серверным сокетом необходимо его закрыть, особенно если при этом программа не завершается. Заккрытие сокета освобождает порт и делает его свободным для других программ, которые смогут его использовать. Заккрытие серверного сокета отличается от закрытия обычного сокета `Socket`. Следует отметить, что закрытие `ServerSocket` не только освобождает порт на локальном хосте, но также завершает все открытые в настоящее время сокеты, которые были созданы для соединений, которые были приняты `ServerSocket`.

Серверные сокеты автоматически закрываются после завершения работы программы. Поэтому для программ, работа которых заканчивается сразу после того, как `ServerSocket` больше не нужен, явно закрывать серверный сокет не обязательно. Тем не менее, рекомендуется это делать.

После того, как серверный сокет закрыт, он уже не может быть повторно подключен. В *Java* существует метод `isClosed()`, который возвращает значение `true`, если `ServerSocket` был закрыт, и `false`, если нет:

```
public boolean isClosed( )
```

Объекты типа `ServerSocket`, созданные с помощью конструктора без аргументов `ServerSocket()` и еще не связанные с локальным портом, не считаются закрытыми. Вызов `isClosed()` для этих объектов возвращает `false`. Для таких ситуаций *Java* содержит метод `isBound()`, который сообщает, привязан или нет `ServerSocket` к какому-либо порту:

```
public boolean isBound( )
```

Но, следует обратить внимание не то, что метод `isBound()` возвращает `true`, если `ServerSocket` когда-либо был связан с портом, даже если он в данный момент закрыт. Таким образом, для проверки того, открыт ли `ServerSocket` или нет, нужно проверить, что метод `isBound()` возвращает `true`, а метод `isClosed()` возвращает `false`.

### **Методы доступа**

Класс `ServerSocket` содержит два метода-геттера, которые возвращают локальный адрес и порт, занятый сокетом сервера. Эти методы особенно полезны в ситуации, когда серверный сокет открыт на анонимном порту и / или не указан сетевой интерфейс.

### **public InetAddress getInetAddress( )**

Этот метод возвращает локальный адрес, используемый сервером. Если локальный хост имеет один *IP*-адрес, то это будет адресом, возвращаемым методом `InetAddress.getLocalHost()`. Если локальный хост имеет более одного *IP*-адреса, то возвращается один из *IP*-адресов хоста. При этом нет возможности предсказать, какой адрес будет получен. Если `ServerSocket` еще не связан с сетевым интерфейсом, этот метод возвращает `null`.

### **public int getLocalPort( )**

Класс `ServerSocket` содержит конструктор, позволяющий создать серверный сокет, прослушивающий анонимный, неизвестный заранее порт, передавая 0 для номера порта. Этот метод дает возможность узнать, какой конкретно порт прослушивается серверным сокетом. Если `ServerSocket` еще не привязан к порту, этот метод возвращает -1.

## **Параметры серверного сокета**

Архитектура *Java* поддерживает несколько параметров, определяющих работу серверного сокета. Мы рассмотрим только параметр `SO_TIMEOUT`.

Этот параметр определяет время в миллисекундах, в течение которого метод `accept()` ожидает входящее соединение, прежде чем выдать исключение `java.io.InterruptedIOException`. Если параметр `SO_TIMEOUT` равен 0, то метод `accept()` не прерывается исключением (т.е. ждет до «бесконечности»). Это значение параметра по умолчанию.

Обычно параметр `SO_TIMEOUT` для серверного сокета остается в значении по умолчанию (0, ожидание входящего соединения никогда не прерывается). Обычно, изменять его рекомендуется в случае, когда реализуется сложный и безопасный протокол, который требует нескольких соединений между клиентом и сервером, где ответы должны происходить в течение фиксированного периода времени. Для управления этим параметром используются методы:

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout( ) throws IOException
```

Обратный отсчет времени начинается при вызове метода `accept()`. По истечении времени ожидания, если за это время не пришел запрос на обслуживание от клиента, метод `accept()` выбрасывает исключение `InterruptedException`. (В современной *Java* метод генерирует `SocketTimeoutException`, подкласс `InterruptedException`). Эту операцию установки значения тайм-аута нужно устанавливать перед вызовом метода `accept()`; невозможно изменить значение тайм-аута в то время, когда `accept()` ожидает соединения. Аргумент метода установки тайм-аута должен быть больше или равен нулю; если это не так, метод генерирует исключение `IllegalArgumentException`.

## Класс `Socket`

Класс `java.net.Socket` () является основным классом *Java* для выполнения операций по организации *TCP* соединения на стороне клиента. Этот класс использует нативный (*native*) код для связи с локальным стеком *TCP* операционной системы хоста. Методы класса `Socket` устанавливают и разрывают соединения и управляют различными параметрами сокетов. Фактическое чтение и запись данных с помощью сокета осуществляется через потоковые классы ввода/вывода.

Рассмотрим основные возможности класса, для более подробного изучения полную документацию можно посмотреть по адресу: <https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html>.

## Конструкторы

Конструкторы класса `Socket` просты и понятны. При создании объекта `Socket` указывается хост и порт, к которому нужно подключиться. Хосты могут быть указаны либо в виде объекта `InetAddress`, либо `String`. Значения номера порта должно быть в диапазоне от 0 до 65 535. В некоторых конструкторах можно также указать локальный адрес и локальный порт, с которого будут отправляться данные. Кроме того, класс `Socket` содержит также конструкторы, создающие неподключенные сокеты. Это бывает необходимо, когда нужно установить параметры сокета перед установлением первого соединения.

**`public Socket(String host, int port)`**

**`throws UnknownHostException, IOException`**

Этот конструктор создает *TCP*-сокет для взаимодействия с указанным хостом `host` на указанном порте `port` и пытается выполнить подключение к указанному удаленному хосту. Если *DNS*-сервер не может разрешить имя хоста или вообще не работает, то этот конструктор генерирует исключение



`UnknownHostException`. Если сокет не может быть создан по какой-либо другой причине, то конструктор выбрасывает исключение `IOException`. Существует множество причин, по которым попытка подключения будет неудачной: хост, к которому осуществляется подключение, может не принимать подключения; подключение к Интернету через модемное соединение может не работать или возникнут проблемы с маршрутизацией, которые могут препятствовать доставке пакетов к месту назначения.

### **public Socket(InetAddress host, int port) throws IOException**

Как и предыдущий конструктор, этот конструктор создает *TCP*-сокет для установления связи с указанным хостом на указанном порте и пытается реально установить подключение. В отличие от предыдущего конструктора, в этом для указания хоста использует объект `InetAddress`. Данный конструктор может генерировать исключение `IOException`, если подключение не может быть установлено. Этот конструктор не генерирует исключение `UnknownHostException`, если хост неизвестен – об этом станет известно при создании объекта `InetAddress`.

Этот конструктор бывает очень удобен, если создается много сокетов для одного хоста: более эффективно преобразовать имя хоста в `InetAddress`, а затем многократно использовать этот объект для создания сокетов.

### **public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException**

Этот конструктор создает *TCP*-сокет для установления соединения с удаленным хостом. Он подключается к тому хосту и порту, которые указаны в первых двух аргументах. Он связывается с локальным сетевым интерфейсом и портом, указанным в последних двух аргументах. Сетевой интерфейс может быть физическим или виртуальным. Если для аргумента `localPort` передается значение 0, то *Java* выбирает случайный порт из доступных в диапазоне от 1024 до 65535. Данный конструктор удобен, например, в ситуации, когда на компьютере с несколькими сетевыми интерфейсами входящие соединения будут приниматься на одном интерфейсе, обрабатываться и пересылаться в локальную сеть с другого интерфейса. Этот конструктор может выбрасывать исключение `IOException` по тем же причинам, что указаны в предыдущих конструкторах. Кроме того, может быть выброшено исключение `UnknownHostException`, если удаленный хост не может быть найден. Кроме того, будет выброшено исключение `IOException` (вообще-то, исключение типа `BindException`, подкласс `IOException` и из-за этого не указанного в

предложении `throws` этого метода), если сокет не сможет привязаться к запрошенному локальному интерфейсу, что ограничивает переносимость приложений, использующих этот конструктор.

**public Socket(InetAddress host, int port, InetAddress interface,  
int localPort)  
throws IOException**

Этот конструктор похож на предыдущий, за исключением того, что хост для подключения указывается объектом `InetAddress`. Если создание сокета и подключение к удаленному хосту заканчивается ошибкой, то это приводит к генерации исключения `IOException`.

### **public Socket( )**

Класс `Socket` содержит также конструкторы, которые создают объект сокета без подключения к хосту. Конструктор без параметров `Socket()`, используя реализацию сокета по умолчанию, создает новый объект типа `Socket` без выполнения подключения. Подключения может быть выполнено позже, передав объект типа `SocketAddress` (<https://docs.oracle.com/javase/8/docs/api/java/net/SocketAddress.html>) одному из методов `connect()`.

### **Получение информации о сокете**

Объекты типа `Socket` содержат несколько приватных полей, доступных для программиста с помощью различных методов доступа. Вообще-то, сокеты имеют только одно поле, `SocketImpl` (<https://docs.oracle.com/javase/8/docs/api/java/net/SocketImpl.html>) и поля, которые кажутся принадлежащими `Socket`, фактически соответствуют полям `SocketImpl`. Из-за этого, реализации сокетов могут быть изменены без нарушения работы программы, а реально используемый `SocketImpl` является полностью прозрачным для программиста.

### **public InetAddress getInetAddress( )**

Для данного объекта `Socket` метод `getInetAddress()` возвращает объект типа `InetAddress`, который сообщает о том, к какому удаленному хосту сейчас подключен сокет или, если соединение уже закрыто, к какому хосту сокет был подключен ранее.

### **public int getPort( )**

Метод `getPort()` возвращает номер порта, к которому или был, или подключен сокет на удаленном хосте.

### **public int getLocalPort( )**

У соединения есть два конца: удаленный хост и локальный хост. Чтобы найти номер порта для локального конца соединения, используется метод `getLocalPort()`.

В отличие от известного удаленного порта, локальный порт обычно выбирается системой во время выполнения из всех доступных в наличии неиспользуемых портов.

### **public InetAddress getLocalAddress( )**

Метод `getLocalAddress()` возвращает сетевой интерфейс, с которым связан сокет. Обычно этот метод используется на хосте с несколькими сетевыми интерфейсами.

### **public InputStream getInputStream( ) throws IOException**

Метод `getInputStream()` возвращает входной поток, который может считывать данные из сокета и передавать их в обрабатывающую программу. Обычно, этот `InputStream` оборачивается или потоком-фильтром, или читателем, который предлагает больше функциональных возможностей - например, `DataInputStream` или `InputStreamReader`. Из соображений производительности можно также буферизовать входные данные обернув поток `BufferedInputStream` / `BufferedReader`.

### **public OutputStream getOutputStream( ) throws IOException**

Метод `getOutputStream()` возвращает `OutputStream` для записи данных из приложения в другой конец сокета. Обычно этот поток оборачивается другим потоком, таким как `DataOutputStream` или `OutputStreamWriter`. Из соображений производительности его также можно буферизировать.

## **Закрытие сокета**

### **public void close( ) throws IOException**

Вообще-то, сокет закрывается автоматически, когда закрывается один из двух его потоков, когда заканчивается программа или когда его уберет из памяти сборщик мусора. Но, в долго работающих программах с интенсивным использованием сокетов, лучше их закрывать явно.

Для закрытия сокета нужно вызвать метод `close()`. В идеале вызов метода нужно поместить в блок `finally`, чтобы сокет был закрыт независимо от того, было ли выброшено исключение или нет:

```
Socket connection = null;
try {
    connection = new Socket("www.oreilly.com", 13);
    // interact with the socket...
} // end try
catch (UnknownHostException ex) {
    System.err.println(ex);
} catch (IOException ex) {
    System.err.println(ex);
} finally {
    if (connection != null) connection.close( );
}
```

После закрытия объекта `Socket` его `InetAddress`, номер порта, локальный адрес и номер локального порта по-прежнему остаются доступными с помощью метода `getInetAddress()`, `getPort()`, `getLocalAddress()` и `getLocalPort()`. При этом, хотя на закрытом соquete можно вызывать `getInputStream()` или `getOutputStream()`, попытка чтения данных из `InputStream` или записи данных в `OutputStream` вызывает исключение `IOException`.

*Java* содержит добавляет метод `isClosed()`, который возвращает `true`, если сокет был закрыт, и `false`, если это не так:

### **public boolean isClosed( )**

Если в программе неизвестно состояние сокета, то можно его определить с помощью вызова этого метода, без риска получить `IOException`. Например:

```
if (socket.isClosed( )) {
    // do something...
} else {
    // do something else...
}
```

Однако это не идеальный тест на возможность использования сокета. Если сокет был создан в неподключенном состоянии, метод `isClosed()` возвращает `false`. *Java* содержит метод `isConnected()`:

```
public boolean isConnected( )
```

Этот метод сообщает вам, был ли вообще сокет когда-либо подключен к удаленному хосту. Если сокет был подключиться к удаленному хосту, то этот метод возвращает `true`, даже после того, как этот сокет был закрыт. Чтобы узнать, открыт ли в данный момент сокет, нужно проверить, что метод `isConnected()` возвращает `true`, а метод `isClosed()` возвращает `false`. Например:

```
boolean connected = socket.isConnected() && ! socket.isClosed( );
```

*Java* также содержит метод `isBound()`:

```
public boolean isBound( )
```

Метод `isConnected()` относится к удаленному концу сокета, а метод `isBound()` относится к локальному концу. Он сообщает, успешно ли сокет связан с портом в локальной системе.

### **Полузакрытые сокеты**

Метод `close()` закрывает как входной, так и выходной потоки сокета. Иногда может потребоваться отключить только половину соединения: вход или выход. В *Java* методы `shutdownInput()` и `shutdownOutput()` позволяют закрыть только половину соединения:

```
public void shutdownInput( ) throws IOException
public void shutdownOutput( ) throws IOException
```

Эти методы не закрывают сокет, а просто закрывает один конец потока. Когда клиентская программа передает запрос серверу, сервер должен иметь возможность отследить окончание запроса. По этой причине многие протоколы, используемые в *Интернет*, ориентированы на работу со строками. Другие протоколы предоставляют поле заголовка, которое указывает объем данных в составе запроса. Указать конец передаваемых данных при работе по сети сложнее, чем при записи в файл. Закрыв сокет,

соединение сервером будет немедленно разорвано. Данную проблему решает одностороннее закрытие. Если вы закроете выходной поток, связанный с сокетом, вы тем самым укажете серверу на окончание запроса. При этом входной поток останется открытым и вы сможете прочитать ответ. Чтение закрытого входного потока вернет -1. Дальнейшая запись в закрытый выходной поток выбросит `IOException`. Рассмотрим небольшой схематический пример:

```
Socket connection = null;
try {
    connection = new Socket(".. ... ..", 80);
    Writer out = new OutputStreamWriter(
        connection.getOutputStream( ), "8859_1");
    out.write("GET / HTTP 1.0\r\n\r\n");
    out.flush( );
    connection.shutdownOutput( );
    // read the response...
} catch (IOException ex) {
} finally {
    try {
        if (connection != null) connection.close( );
    } catch (IOException ex) {}
}
```

Важно помнить, что даже отключено половину или даже обе половины соединения, все равно нужно закрыть сокет, когда с ним закончится работа. Эти методы просто влияют на потоки сокета. Они не освобождают ресурсы, связанные с сокетом (например, порт), который он занимает.

*Java* добавляет два метода, с помощью которых можно узнать, открытые или закрытые входной и выходной потоки:

```
public boolean isInputShutdown( )
public boolean isOutputShutdown( )
```

Эти методы можно использовать для определения, можно или нет читать из или записывать в сокет.

## Параметры сокета

Клиентские сокеты имеют достаточно много параметров, из которых мы рассмотрим параметр `SO_TIMEOUT`. Для управления этим параметром используются методы:

```
public void setSoTimeout(int milliseconds) throws SocketException
public int getSoTimeout( ) throws SocketException
```

Обычно, когда данные считываются из сокета, вызов метода `read()` блокирует выполнение потока исполнения, на то времени, которое понадобится для получения нужного количества байтов. Установка параметра `SO_TIMEOUT`, приводит к тому, что вызов не будет блокировать поток исполнения более чем на заданное количество миллисекунд. Когда время ожидания истекает, выбрасывается исключение `InterruptedIOException`, и его нужно или передать дальше, или обработать. Однако при этом сокет все еще является подключенным. Хотя этот вызов `read()` был прерван, можно попытаться снова прочесть данные из сокета. Время ожидания указывается в миллисекундах. Ноль интерпретируется как бесконечный тайм-аут и это значение по умолчанию.

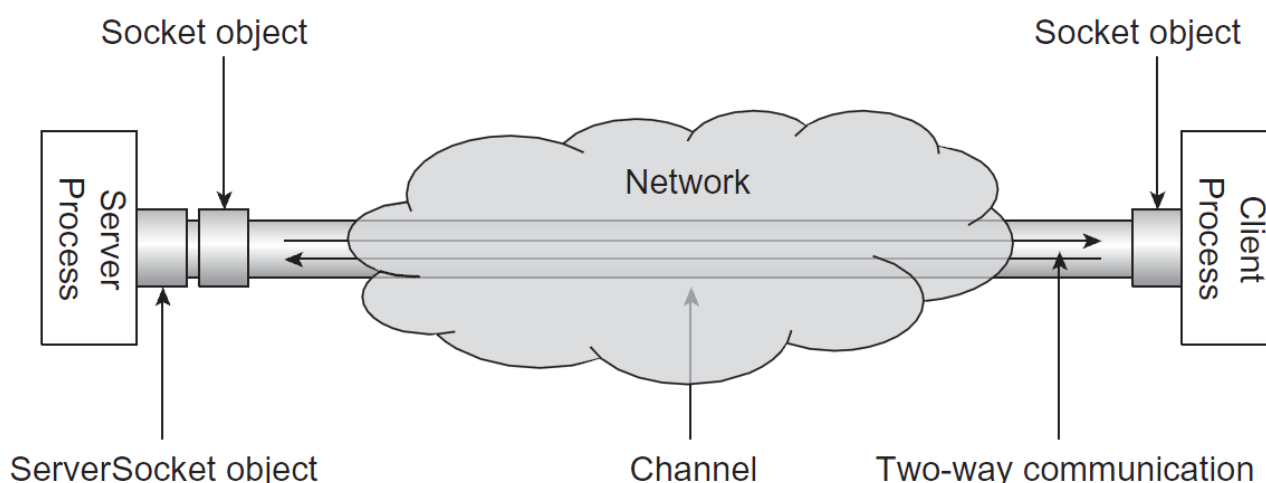
Каждый из этих двух методов может выбросить исключение `SocketException`, если базовая реализация сокета не поддерживает опцию `SO_TIMEOUT`. Метод `setSoTimeout()` может также выбросить исключение `IllegalArgumentException`, если значение времени ожидания указано отрицательным значением.

## **Основные этапы создания клиент / серверного приложения**

Как уже было сказано ранее, архитектура *Java* поддерживает протокол управления передачей (*TCP*). Пакет `java.net` содержит классы `ServerSocket` и `Socket` для поддержки связи по протоколу *TCP*. Перед разбором функциональности этих классов, сначала кратко повторим особенности работы по протоколу *TCP*. При таком способе связи перед обменом данными между двумя взаимодействующими сторонами создается логический канал (с использованием специальной процедуры установления связи). Как только канал создан, данные могут передаваться как непрерывный поток байтов в оба направления одновременно (полнодуплексная связь). Поскольку все данные проходят через один канал, данные принимаются в том же порядке, в котором они были отправлены. Эта упорядоченная доставка данных известна как связь с установлением соединения. Для гарантированной доставки данных *TCP* также использует сложную стратегию подтверждения и повторной передачи. В

конце сеанса связи соединение разрывается (используется специальная процедура разрыва соединения – процедура из 4-х «рукопожатий»).

Чтобы использовать этот способ связи с использованием потокового сокета *Java*, одна программа (обычно называемая *сервером*) сначала создает объект **ServerSocket** и связывает его к свободному номеру порта (см. Рис.). Затем этот серверный сокет получает адрес сокета, который впоследствии будет использовать клиентской частью приложения для установления соединения. Затем серверный сокет начинает прослушивать запросы на подключение от клиентов. Это значит, что серверная часть приложения должна быть запущена первой. Нужно обратить внимание на то, что когда серверный сокет ожидает входящие запросы на подключение, серверная часть приложения блокируется. Для одновременного приема нескольких соединений серверная программа обычно делается многопоточной.



Клиентская часть приложения - это другая программа, запускаемая обычно на другом компьютере после запуска серверной части. В наших занятиях мы будем запускать серверную и клиентскую части приложений на одном компьютере, но на разных виртуальных машинах *Java*. Они будут «общаться» друг с другом по сети. Клиент должен каким-то образом узнать адрес серверного сокета (*IP*-адрес / имя компьютера, на котором запускается серверная программа и номер порта серверного сокета, заданного серверной программой) для установления связи. Сначала клиент отправляет запрос на установление соединения на серверный сокет, создав объект **Socket** и указав для него адрес серверного сокета. Во время установления соединения клиент должен предоставить свой собственный адрес серверному сокету, чтобы при необходимости сервер мог связаться с клиентом. Номер порта клиента обычно автоматически назначается системой.

После получения запроса от клиента, если при этом не возникли проблемы, сервер устанавливает двунаправленный канал связи с клиентом.



Как только канал установлен, оба процесса (клиентский и серверный) могут взаимодействовать одновременно в двух направлениях (см. Рис. выше).

Существует два основных класса для организации такой связи по *TCP*-протоколу: **ServerSocket** и **Socket**. Связь через *TCP*-сокеты состоит из следующих основных этапов:

- серверная сторона взаимодействия создает объект **ServerSocket**, указывая номер порта, который будет прослушиваться сокетом.
- серверная сторона взаимодействия для созданного объекта серверного сокета вызывает метод **accept()**. Это блокирующий метод, заставляющий серверный сокет ожидать, пока от клиента не поступит запрос.
- клиентская сторона взаимодействию создает объект **Socket**, указывая *имя / IP-адрес сервера и номер порта* для подключения.
- конструктор класса **Socket** пытается установить соединение с серверным сокетом, указав адрес сокета (то есть *IP-адрес и номер порта*). Если соединение установлено, то успешно создается объект **Socket**, представляющий клиентскую часть созданного логического соединения. Клиент использует этот объект **Socket** для связи с сервером.
- метод **accept()** объекта **ServerSocket** возвращает объект **Socket**, представляющий серверную сторону канала, который подключен к сокету клиента. Сервер использует этот объект **Socket** для связи с клиентом.

Рассмотрим временной поток управления в серверной и клиентской частях приложения, показанного на рисунке выше:

### Серверная часть

1. Создается объект **ServerSocket**, по номеру порта.
2. Вызывается метод **accept()** для этого объекта. Этот метод заставляет сервер ожидать входящий запрос на соединение

### Клиентская часть

3. Создается объект **Socket**, по имени / *IP-адресу* и номеру порта сервера.
  4. Конструктор класса **Socket** пытается установить соединение с указанным сервером по указанному номеру порта.
  5. Если соединение установлено, то возвращается объект **Socket**, который
5. Метод **accept()** объекта **ServerSocket** также возвращает

объект `Socket`, представляющий соединение на стороне сервера.

6. С помощью объекта `Socket` получают объекты типа `InputStream` и `OutputStream`.

8. Данные, которые были переданы серверу, считываются с помощью `InputStream`.

9. Данные, полученные от клиента обрабатываются и результат передается клиентской стороне с помощью `OutputStream`.

представляет соединение на стороне клиента.

6. С помощью объекта `Socket` получают объекты типа `InputStream` и `OutputStream`.

7. Данные, которые должны быть переданы серверу, передаются с помощью `OutputStream`.

10. Данные, переданные клиенту считываются с помощью `InputStream`.

Представим небольшую экстремально простую, но работающую, программу, которая реализует эту схему. Сервер ожидает подключения клиента. Клиент получает строку от пользователя, отправляет ее на сервер. Сервер преобразует строку к верхнему регистру и отправляет преобразованную строку клиенту. Клиент получает данные и выводит их на экран.

### *Серверная часть приложения:*

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleServer {
    public static void main(String[] args) throws Exception {
        //1.
        ServerSocket serverSocket = new ServerSocket(6789);
        System.out.println("Server is listening on port 6789");
        //2.
        Socket serverEnd = serverSocket.accept();
        //5.
        System.out.println("Request accepted");
        //6.
        BufferedReader fromClient = new BufferedReader(new
        InputStreamReader(serverEnd.getInputStream()));
        PrintWriter toClient = new PrintWriter(serverEnd.getOutputStream(), true);
        //8.
        String inputMessage = fromClient.readLine();
        //9.
        System.out.println("Received from client: " + inputMessage);
        String outputMessage = inputMessage.toUpperCase();
        toClient.println(outputMessage);
        System.out.println("Sent to client: " + outputMessage);
    }
}
```

```

    }
}

```

### *Клиентская часть приложения:*

```

package lection1;

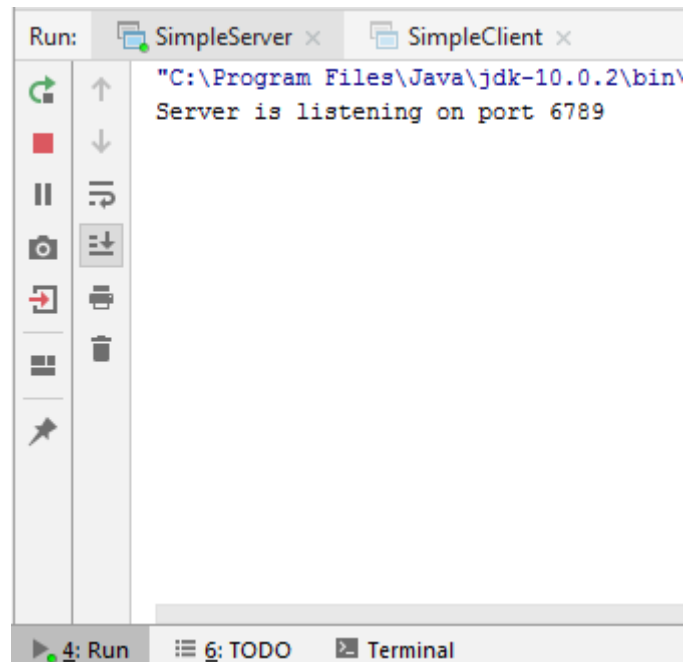
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class SimpleClient {
    public static void main(String argv[]) throws Exception {
        //3., 4., 5.
        Socket clientEnd = new Socket("localhost", 6789);
        System.out.println("connected to localhost at port 6789");
        //6.
        PrintWriter toServer = new PrintWriter(clientEnd.getOutputStream(), true);
        BufferedReader fromServer = new BufferedReader(new
InputStreamReader(clientEnd.getInputStream()));
        Scanner fromUser = new Scanner(System.in);
        System.out.print("Enter the string: ");
        String stringToServer = fromUser.nextLine();
        //7.
        toServer.println(stringToServer);
        System.out.println("Sent to server: " + stringToServer);
        //10.
        String stringFromServer = fromServer.readLine();
        System.out.println("Received from server: " + stringFromServer);
        clientEnd.close();
    }
}

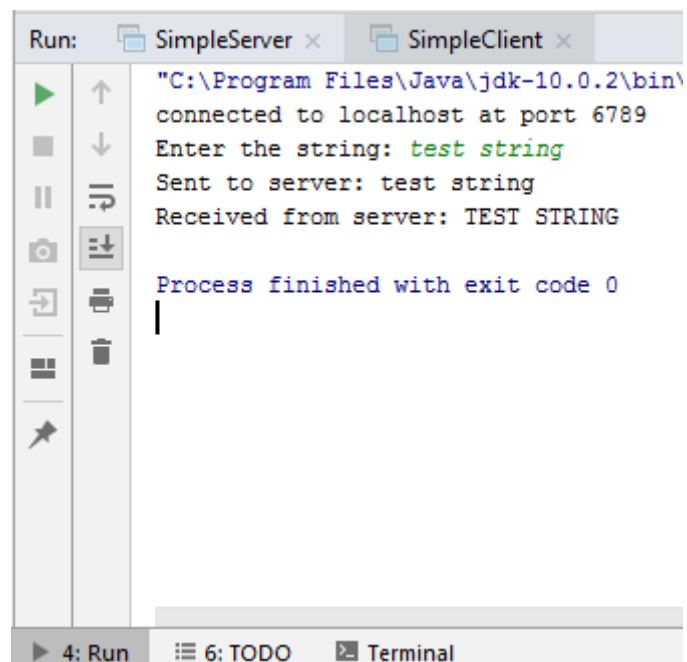
```

Рассмотрим процесс запуска такого простейшего клиент / серверного приложения.

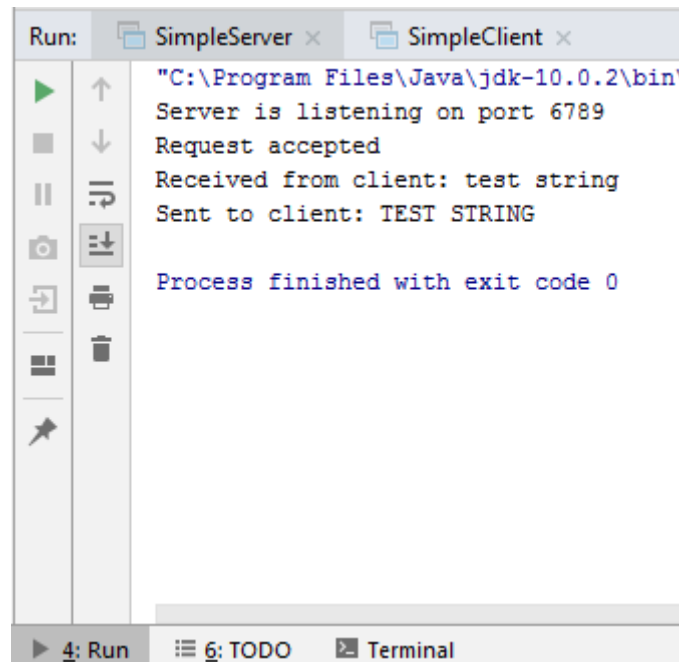
Сначала запускаем серверную часть. Создается клиентский сокет и ожидает подключение клиента:



Запускаем клиентскую часть приложения. Вводим данные и получаем ответ от сервера:



Рассмотрим информацию, выведенную на экран на серверной стороне:



Если запустить клиентскую часть приложения без запуска серверной, то полетят исключения, которые, на самом деле, в этой простой программе не обрабатываются.



## Примеры приложений

Рассмотрим примеры простых демонстрационных приложений с помощью *поточковых (TCP)* сокетов.

Сначала напишем приложение для вычисления факториала целого числа. Клиентская часть приложения взаимодействует с пользователем, получает целые числа, факториалы которых будут вычислены, отправляет эти числа на серверную сторону приложения и выводит полученные результаты на экран. Серверная часть приложения проводит собственно вычисления. Приложение создано в соответствии с предложенной схемой.

```

import java.io.*;
import java.net.*;
public class TCPFactClient {
    public static void main(String argv[]) throws Exception {
        String fact;
        //create a socket to the server
        Socket clientEnd = new Socket("localhost", 6789);
        System.out.println("Connected to localhost at port 6789");
        //get streams
        PrintWriter toServer = new PrintWriter(clientEnd.getOutputStream(), true);
        BufferedReader fromServer = new BufferedReader(new
            InputStreamReader(clientEnd.getInputStream()));
        BufferedReader fromUser = new BufferedReader
            (new InputStreamReader(System.in));
        //get an integer from user
        System.out.print("Enter an integer: ");
        String n = fromUser.readLine();
        //send it to server
        toServer.println(n);
        System.out.println("Sent to server: " + n);
        //retrieve result
        fact = fromServer.readLine();
        System.out.println("Received from server: " + fact);
        //close the socket
        clientEnd.close();
    }
}

```

```
package simple;
```

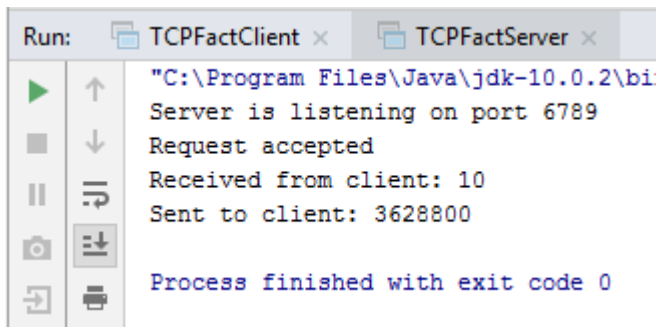
```

import java.io.*;
import java.net.*;

public class TCPFactServer {
    public static void main(String argv[]) throws Exception {
        //create a server socket at port 6789
        ServerSocket serverSocket = new ServerSocket(6789);
        //wait for incoming connection
        System.out.println("Server is listening on port 6789");
        Socket serverEnd = serverSocket.accept();
        System.out.println("Request accepted");
        //get streams
        BufferedReader fromClient = new BufferedReader(new
            InputStreamReader(serverEnd.getInputStream()));
        PrintWriter toClient = new PrintWriter(serverEnd.getOutputStream(), true);
        //receive data from client
        int n = Integer.parseInt(fromClient.readLine());
        System.out.println("Received from client: " + n);
        int fact = 1;
        for (int i = 2; i <= n; i++)
            fact *= i;
        //send result to the client
        toClient.println(fact);
        System.out.println("Sent to client: " + fact);
    }
}

```

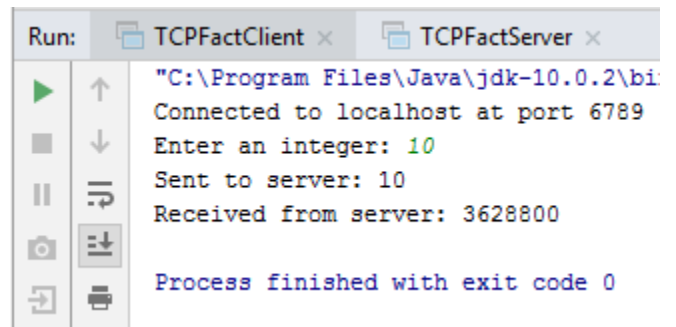
Запуск этого приложения аналогичен предыдущему. Приведем рисунки, демонстрирующие работу серверной и клиентской частей приложения:



```

Run: TCPFactClient x TCPFactServer x
  "C:\Program Files\Java\jdk-10.0.2\bin\java.exe" TCPFactServer
  Server is listening on port 6789
  Request accepted
  Received from client: 10
  Sent to client: 3628800
  Process finished with exit code 0

```



```

Run: TCPFactClient x TCPFactServer x
  "C:\Program Files\Java\jdk-10.0.2\bin\java.exe" TCPFactClient
  Connected to localhost at port 6789
  Enter an integer: 10
  Sent to server: 10
  Received from server: 3628800
  Process finished with exit code 0

```

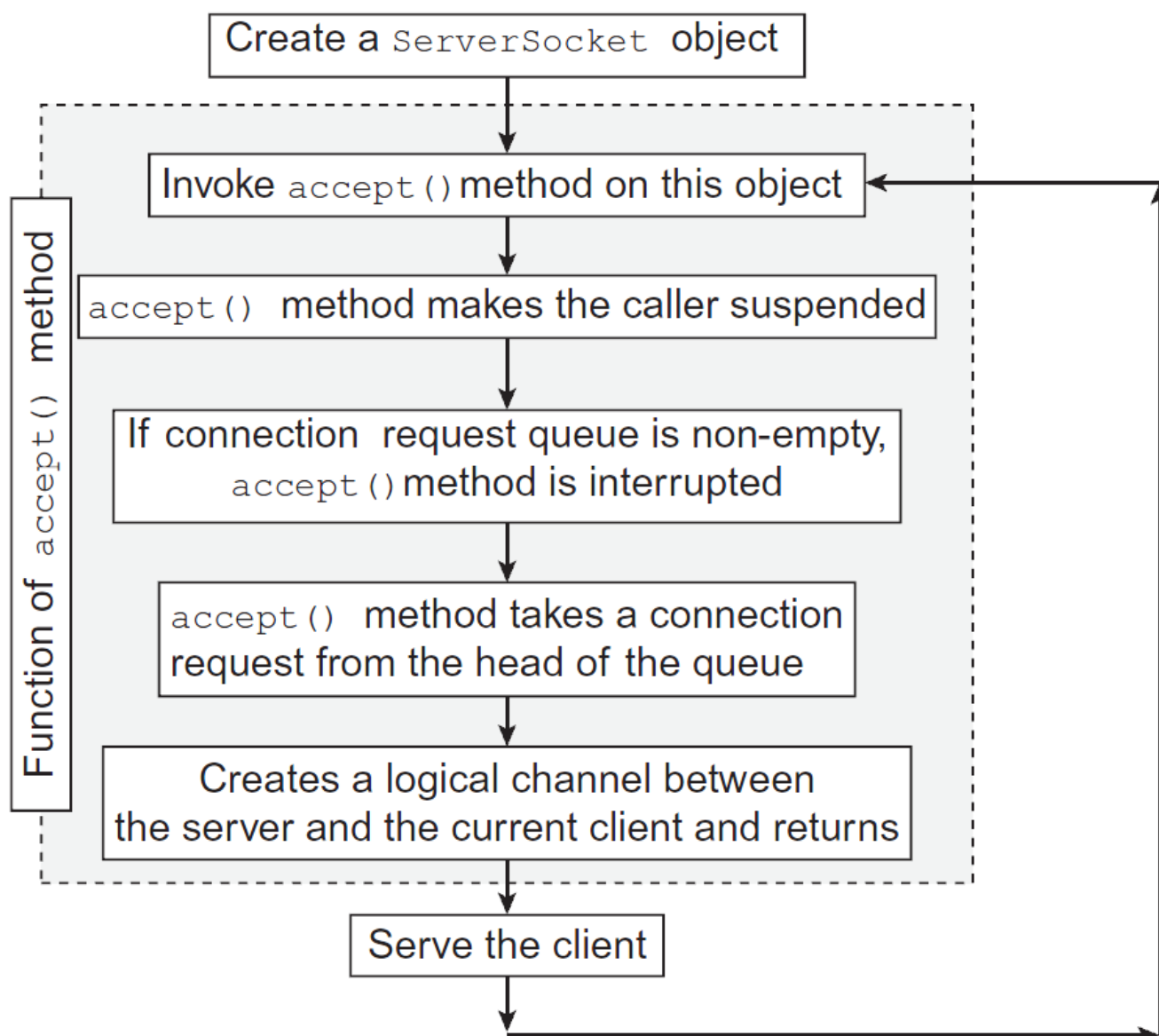
Будем понемногу усложнять приложение. В нашей простой программе серверная часть выполняет вызов метода `accept()` только один раз. Таким образом, наш сервер может обработать только один запрос. Вся работа приложения (состоящего из клиентской и серверной частей) завершается после обработки только одного запроса клиента. Но, вообще-то, серверная часть приложения должна быть спроектирована так, чтобы сервер мог обрабатывать несколько запросов либо от одного, либо от многих клиентов. Рассмотрим варианты усложнения приложения (реализацию сервера) для обработки нескольких запросов, используя:

- итеративный подход;
- параллельный подход.

### Итеративное решение

Возможным решением проблемы работы с многими клиентами является создание сервера, который последовательно обслуживает клиентов: один за другим в порядке поступления запросов. Основная идея создания серверной части приложения заключается в следующем:

Серверная часть приложения создает объект типа `ServerSocket` и вызывает на нем метод `accept()`, выполняющий блокирующее ожидание входящих подключений. Когда приходит запрос на соединение от клиента, метод `accept` создает соединение между сервером и клиентом, и возвращает объект `Socket`, представляющий серверный конец канала. Далее этот объект `Socket` используется для организации взаимодействия с клиентом. Если во время сеанса связи поступают другие запросы на подключение, они сначала ожидают в очереди *First Come First Served*, связанной с объектом типа `ServerSocket`. Когда соединение с текущим клиентом завершено, сервер снова вызывает метод `accept()`, который обрабатывает запрос, находящийся на вершине очереди и заново повторяет обработку запроса (см. Рис.).



Приведем пример кода серверной части приложения:

```

package simple;

import java.io.*;
import java.net.*;

public class TCPSerialFactServer {
    public static void main(String argv[]) throws Exception {
        //create a server socket at port 6789
        //ServerSocket serverSocket = new ServerSocket(6789);
        ServerSocket serverSocket = new ServerSocket(6789, 2);
        while(true) {
            //wait for incoming connection
            System.out.println("Server is listening on port 6789");
            Socket serverEnd = serverSocket.accept();
            System.out.println("Connection from " + serverEnd.getInetAddress() + " from
port: " + serverEnd.getPort());
            System.out.println("Request accepted");
            //get streams
            BufferedReader fromClient = new BufferedReader(new
                InputStreamReader(serverEnd.getInputStream()));
            PrintWriter toClient = new PrintWriter(serverEnd.getOutputStream(),
                true);
        }
    }
}
  
```



```

//receive data from client
    int n = Integer.parseInt(fromClient.readLine());
    System.out.println("Received from client: " + n);
    int fact = 1;
    for (int i = 2; i <= n; i++)
        fact *= i;
//send result to
    //send result to the client
    toClient.println(fact);
    System.out.println("Sent to client: " + fact);
}
}
}

```

Далее приведем немного измененный код клиентской части приложения:

```

package simple;

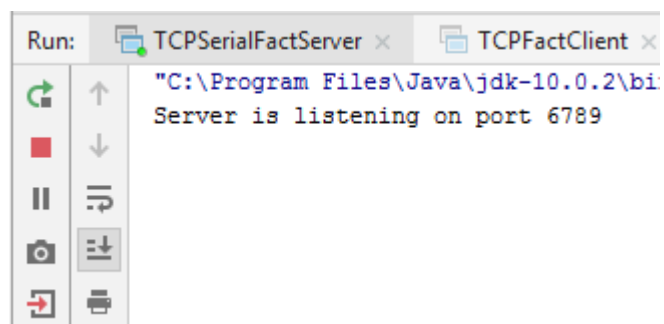
import java.io.*;
import java.net.*;

public class TCPFactClient {
    public static void main(String argv[]) throws Exception {
        String fact;
        //create a socket to the server
        Socket clientEnd = new Socket("localhost", 6789);
        System.out.println("Client port: " + clientEnd.getLocalPort());
        System.out.println("Connected to localhost at port 6789");
        //get streams
        PrintWriter toServer = new PrintWriter(clientEnd.getOutputStream(), true);
        BufferedReader fromServer = new BufferedReader(new
            InputStreamReader(clientEnd.getInputStream()));
        BufferedReader fromUser = new BufferedReader
            (new InputStreamReader(System.in));
        //get an integer from user
        System.out.print("Enter an integer: ");
        String n = fromUser.readLine();
        //send it to server
        toServer.println(n);
        System.out.println("Sent to server: " + n);
        //retrieve result
        fact = fromServer.readLine();
        System.out.println("Received from server: " + fact);
        //close the socket
        clientEnd.close();
    }
}

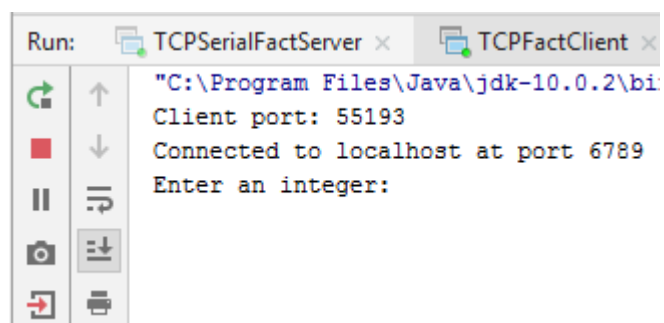
```

Нужно обязательно обратить внимание на создание серверного сокета: использован конструктор с параметром, определяющим максимальную длину очереди для входящих соединения. То есть, в данном примере, пока обрабатываем соединение, еще 2 соединения могут находиться в очереди – ожидать соединения. А еще один клиент (в данном случае 4-й) запустится не сможет – будут выброшены исключения. По умолчанию длина очереди – 50, в примере в демонстрационных целях ее сильно уменьшили.

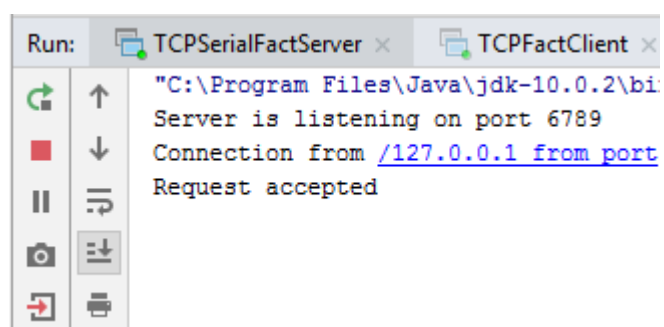
Рассмотрим последовательн работу приложения. Сначала запустим серверную часть приложения. Будет создан серверный сокет, прослушивающий заданный при его создании порт.



На следующем этапе запускаем клиентские части. После запуска первого клиента создается клиентский сокет, при этом устанавливается сетевое соединение и после этого пользователь должен ввести целое число для вычисления.



На серверной стороне приложения будет выполнен метод `accept()`, создан сокет для взаимодействия с только что созданным клиентом. Информация об этом будет выведена на экран.



Не завершая этого сеанса работы с клиентом создадим еще одно приложение – клиент. При создании клиента будет предпринята попытка создать сокет и осуществить подключение к серверу. В это время на сервере идет работа с первым клиентом и метод `accept()` не готов к работе. Данный нормально создается клиент становится первым в очереди на подключение.

```

Run: TCPSerialFactServer x TCPFactClient x
"C:\Program Files\Java\jdk-10.0.2\bin"
Client port: 55216
Connected to localhost at port 6789
Enter an integer:

```

Не завершая работы с уже созданными клиентскими частями, создадим еще одного клиента. Ситуация повторяется.

```

Run: TCPSerialFactServer x TCPFactClient x
"C:\Program Files\Java\jdk-10.0.2\bin"
Client port: 55230
Connected to localhost at port 6789
Enter an integer:

```

Так как при создании серверного сокета был указан второй параметр, равный 2 – длина очереди ожидающих подключения клиентов, то попытка создать еще одного клиента, не завершая работу уже существующих клиентских частей, вызовет ошибку соединения. Сокет не будет создан, будет выброшено исключение `java.net.ConnectionException` и приложение будет аварийно завершено, т.к. это исключения в данной версии программы не оурабатывается.

```

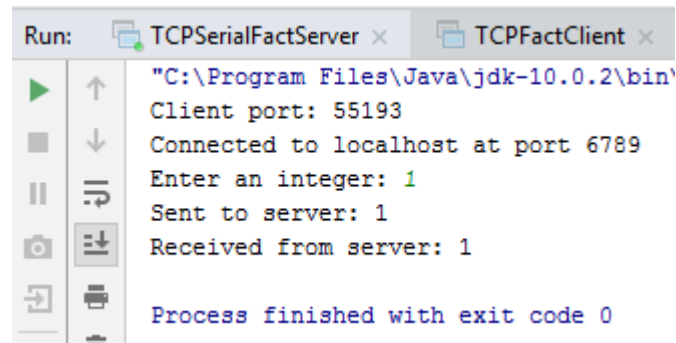
Run: TCPSerialFactServer x TCPFactClient x TCPFactClient x TCPFactClient x TCPFactClient x
Exception in thread "main" java.net.ConnectException: Connection refused: connect
    at java.base/java.net.DualStackPlainSocketImpl.connect0(Native Method)
    at java.base/java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:71)
    at java.base/java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:400)
    at java.base/java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:437)
    at java.base/java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:225)
    at java.base/java.net.PlainSocketImpl.connect(PlainSocketImpl.java:148)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:402)
    at java.base/java.net.Socket.connect(Socket.java:591)
    at java.base/java.net.Socket.connect(Socket.java:540)
    at java.base/java.net.Socket.<init>(Socket.java:436)
    at java.base/java.net.Socket.<init>(Socket.java:213)
    at simple.TCPFactClient.main(TCPFactClient.java:9)

Process finished with exit code 1

```

Переключившись в консоль, где работает первый клиент приложения, введем целое число (в данном случае 1) и передадим его на сервер для

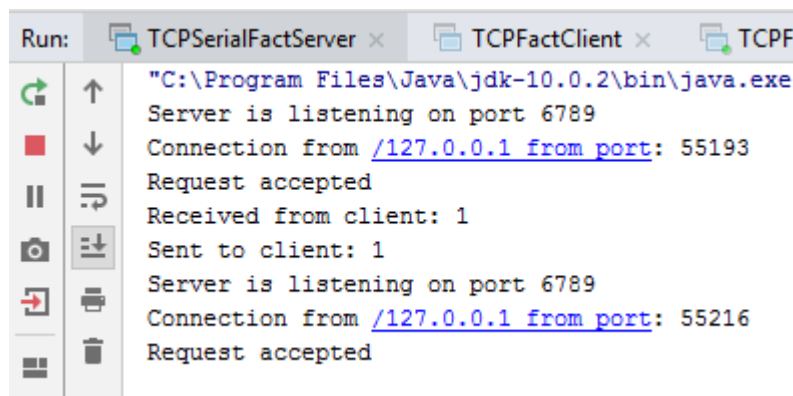
вчисления. Сервер получит число, проведет вычисление и вернет результат назад. Клиентская часть получит ответ, выведет его на экран и закончит работу.



```
Run: TCPSerialFactServer x TCPFactClient x
"C:\Program Files\Java\jdk-10.0.2\bin\
Client port: 55193
Connected to localhost at port 6789
Enter an integer: 1
Sent to server: 1
Received from server: 1

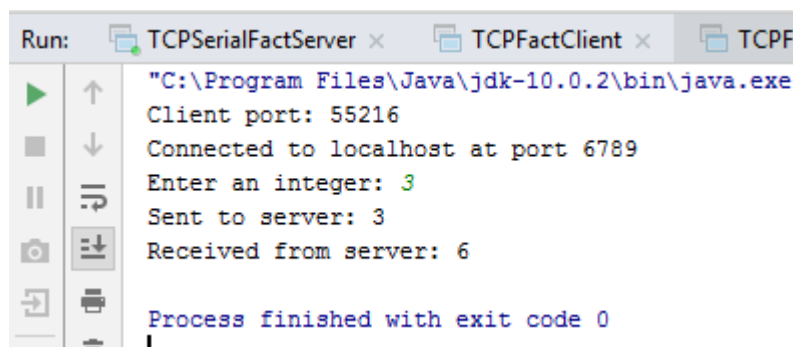
Process finished with exit code 0
```

На серверной части приложение будет выведена отладочная информация о том, с каким клиентом было осуществлено соединение, что от него было получено, какой результат был ему передан, и далее будет выполнен метод `accept()`, создан новый сокет для соединения с первым клиентом, находящимся в очереди на соединение (второй запущенный на выполнение клиент).



```
Run: TCPSerialFactServer x TCPFactClient x TCPF
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55193
Request accepted
Received from client: 1
Sent to client: 1
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55216
Request accepted
```

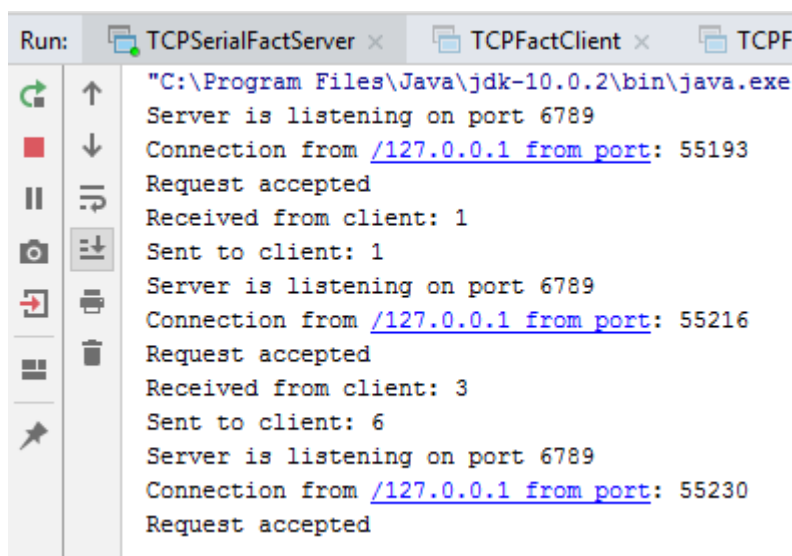
Переключаемся в следующее консольное окно и вводим необходимую информацию. Она будет передана на сервер, обработана, сервером будет сформирован ответ и передан клиенту.



```
Run: TCPSerialFactServer x TCPFactClient x TCPF
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe
Client port: 55216
Connected to localhost at port 6789
Enter an integer: 3
Sent to server: 3
Received from server: 6

Process finished with exit code 0
|
```

Аналогично обработке предыдущего соединения, на сервере будет выполнена обработка этого, будет выведена отладочная информация и, вызван метод `ассепт()` и организована работа со следующим клиентом в очереди.

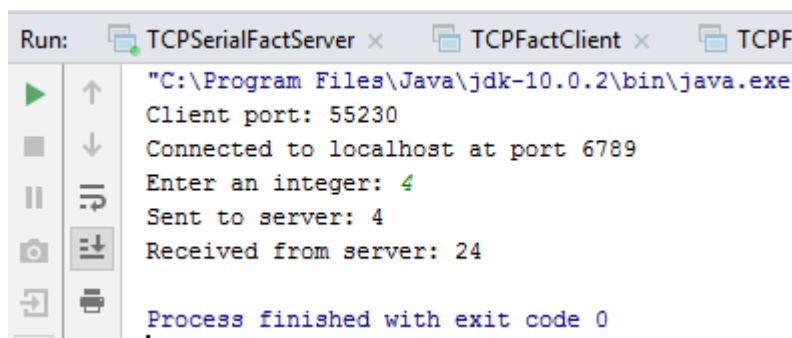


```

Run: TCPSerialFactServer x TCPFactClient x TCPF
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55193
Request accepted
Received from client: 1
Sent to client: 1
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55216
Request accepted
Received from client: 3
Sent to client: 6
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55230
Request accepted

```

Аналогично будет обработан второй клиент очереди (последний клиент).



```

Run: TCPSerialFactServer x TCPFactClient x TCPF
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe
Client port: 55230
Connected to localhost at port 6789
Enter an integer: 4
Sent to server: 4
Received from server: 24
Process finished with exit code 0

```

На серверной части будет выведена необходимая информация, вызван метод `ассепт()`, который заблокирует работу серверной части приложения, так как нет клиентов, ожидающих соединения.

```

Run: TCPSerialFactServer x TCPFactClient x TCPF
Connection from /127.0.0.1 from port: 55193
Request accepted
Received from client: 1
Sent to client: 1
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55216
Request accepted
Received from client: 3
Sent to client: 6
Server is listening on port 6789
Connection from /127.0.0.1 from port: 55230
Request accepted
Received from client: 4
Sent to client: 24
Server is listening on port 6789

```

В таком состоянии серверная часть будет ожидать поступления запросов на подключение от новых клиентов. Завершите работу серверной части средствами интегрированной среды разработки. Все, приложение полностью закончило работу.

Следует отметить, что попытка запуска второго экземпляра серверной части приложения, без закрытия первой приведет к ошибке создания серверного сокета.

```

Run: TCPSerialFactServer TCPSerialFactServer
"C:\Program Files\Java\jdk1.8.0_141\bin\java" ...
Exception in thread "main" java.net.BindException: Address already in use: JVM_Bind
    at java.net.DualStackPlainSocketImpl.bind0(Native Method)
    at java.net.DualStackPlainSocketImpl.socketBind(DualStackPlainSocketImpl.java:106)
    at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
    at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:190)
    at java.net.ServerSocket.bind(ServerSocket.java:375)
    at java.net.ServerSocket.<init>(ServerSocket.java:237)
    at java.net.ServerSocket.<init>(ServerSocket.java:181)
    at simple.TCPSerialFactServer.main(TCPSerialFactServer.java:9)

Process finished with exit code 1

```

Будет выброшено исключение `java.net.BindException`, информирующее о том, что локальный порт, к которому была предпринята попытка подключения, уже занят и прослушивается.

### Параллельное решение

Предыдущее решение – серверная часть приложения, построенная по итерационному принципу, может обрабатывать большое количество клиентов, но только по очереди. Как было показано при тестировании предыдущего

решения, при такой схеме связи между сервером и клиентом может формироваться очередь клиентов: если во время обработки одного запроса поступают дополнительные запросы на соединение; клиенты ждут, пока сервер завершит обработку текущего соединения. Эти запросы будут обработаны в порядке очереди.

Понятно, что такой дизайн приложения имеет серьезные проблемы - обрабатываемый клиент, блокирует обработку других клиентов. При этом, обычно, обрабатываемый запрос от текущего клиента не требует всех ресурсов, имеющихся у сервера. Таким образом, обычно, «эффективность» использование сервера невысока.

Для обеспечения лучшего использования ресурсов сервера и уменьшения времени отклика для всех клиентов, можно спроектировать сервер таким образом, чтобы он мог одновременно обслуживать несколько запросов. Для этого на сервере нужно:

- прослушивать серверный сокет, ожидая входящее соединение;
- принять новый запрос на подключение и создать канал для взаимодействия;
- создать новый поток исполнения (*thread*); передать этому потоку созданный для взаимодействия канал;
- снова перейти в режим прослушивания, чтобы немедленно принять запросы на подключение;
- поток исполнения независимым образом, используя канал взаимодействия, обрабатывает клиентский запрос и завершается после его обработки.

Идея, лежащая в основе параллельного сервера, очень проста: сервер в основном потоке исполнения прослушивает сокет и ожидает входящие запросы от клиентов. Когда запрос от клиента поступает на сервер, он устанавливает сокетное соединение. Однако, вместо того, чтобы обрабатывать этот запрос в основном потоке исполнения, сервер создает новый поток исполнения и передает это соединение созданному потоку. Этот новый поток независимо обрабатывает запрос клиента и завершается после его обработки. Параллельно с этим в основном потоке исполнения сервер снова переходит в режим прослушивания, чтобы можно было также обработать поступившие запросы (создание нового потока исполнения и т.д.). Такая схема построения сервера называется «*thread per request*» (каждому запросу – свой поток). Таким образом, главный поток отвечает только за принятие входящих запросов, а клиентские запросы обрабатываются соответствующими потоками исполнения. Время создания потока достаточно мало, потоки выполняются «одновременно», а значит и клиенты обслуживаются «одновременно». Даже



если потоку исполнения потребуется много времени для обслуживания какого-то клиента, другим потокам, а следовательно, и другим клиентам, не нужно будет ожидать окончания обработки запроса.

Для реализации такой идеи обычно создается класс – обработчик клиентских запросов: класс `Handler`, например расширяющий класс `java.lang.Thread`. Объекту класса `Handler` нужно какое-то средство, с помощью которого он сможет работать с запросом клиента. Для этого используется объект типа `Socket`, который представляет один конец канала связи, и может использоваться для обмена данными через этот канал. Обычно объект типа `Socket` передается конструктору при создании объекта – обработчика. Поскольку класс `Handler` расширяет класс `java.lang.Thread`, то необходимо реализовать метод `run()` в котором выполняется обработка запроса клиента. Новый поток можно запустить на исполнение в самом конструкторе класса `Handler`; как только объект-обработчик запроса будет создан, запустится новый поток исполнения, в рамках которого сразу будет выполняться обработка запроса клиента.

В первых версиях наша серверная программа вычисляет факториал одного целого числа, переданного на обработку клиентом и закрывает сокетное соединение. Таким образом, каждый раз, когда клиент хочет вычислить факториал целого числа, ему необходимо установить новое соединение. Установление соединения по *TCP* протоколу достаточно долгая процедура (3-х этапное рукопожатие), а завершение соединения – еще более долгая процедура (4-х этапное рукопожатие). Для повышения производительности с точки зрения времени отклика можно не разрывать соединение сразу после вычисления факториала заданного целого числа. Клиент сможет использовать это же соединение чтобы найти факториал еще некоторого количества целых чисел. Соединение будет разорвано только в том случае, когда обе стороны, участвующие в обмене, решат, что соединение больше не требуется. С учетом этого спроектируем метод `run()`, обслуживающий клиента, и немного изменим клиентскую программу, чтобы она могла последовательно посылать нужное количество целых чисел. Договоримся, что для завершения работы клиент должен отправить серверу число -1. Приведем исходный код для сервера и клиента.

Код серверной части приложения:

```
package simple;

import java.io.*;
import java.net.*;

public class TCPMTFactServer {
    public static void main(String argv[]) throws Exception {
```



```

//create a server socket at port 6789
    ServerSocket welcomeSocket = new ServerSocket(6789);
    System.out.println("Server ready");
    while (true) {
//wait for incoming connection
        Socket serverEnd = welcomeSocket.accept();
        System.out.println("Connection from " + serverEnd.getInetAddress() + " from
port: " + serverEnd.getPort());
        System.out.println("Request accepted");
//hand over this connection request to Handler
        new Handler(serverEnd);
    }
}

class Handler implements Runnable {
    Socket serverEnd;
    Handler(Socket s) {
        this.serverEnd = s;
        new Thread(this).start();
        System.out.println("A thread created");
    }
    public void run() {
        try {
//get streams
            BufferedReader fromClient = new BufferedReader(new
                InputStreamReader(serverEnd.getInputStream()));
            PrintWriter toClient = new PrintWriter(serverEnd.getOutputStream(),
                true);
            while (true) {
//receive data from client
                int n = Integer.parseInt(fromClient.readLine());
                System.out.println("Received " + n);
                if (n == -1) {
                    serverEnd.close();
                    break;
                }
                int fact = 1;
                for (int i = 2; i <= n; i++)
                    fact *= i;
//send result to the client
                toClient.println(fact);
                System.out.println("Sent: " + fact);
            }
        } catch (IOException e) { }
    }
}

```

### Код клиентской части приложения:

```

package simple;

import java.io.*;
import java.net.*;

public class TCPMTFactClient {
    public static void main(String argv[]) throws Exception {
        String fact;
//create a socket to the server
        Socket clientEnd = new Socket("localhost", 6789);
        System.out.println("Client port: " + clientEnd.getLocalPort());
        System.out.println("connected to localhost at port 6789");
//get streams
        PrintWriter toServer = new PrintWriter(clientEnd.getOutputStream(), true);

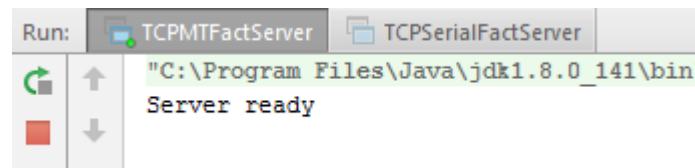
```

```

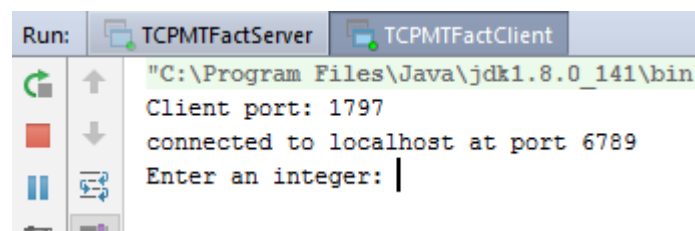
BufferedReader fromServer = new BufferedReader(new
    InputStreamReader(clientEnd.getInputStream()));
BufferedReader fromUser = new BufferedReader(new
    InputStreamReader(System.in));
while (true) {
//get an integer from user
    System.out.print("Enter an integer: ");
    String n = fromUser.readLine();
//send it to server
    toServer.println(n);
    System.out.println("Sent to server: " + n);
    if (n.equals("-1"))
        break;
//retrieve result
    fact = fromServer.readLine();
    System.out.println("Received from server: " + fact);
}
//close the socket
clientEnd.close();
}
}

```

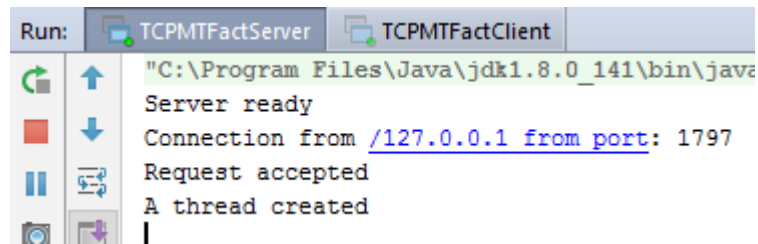
Запустим программу на выполнение. Как и раньше сначала запустим серверную часть приложения. Будет создан серверный сокет и сервер начнет ожидать запросы от клиентов.



На следующем этапе запустим клиентскую часть приложения. Будет создан клиентский сокет и установлено соединение с сервером. Клиент будет ждать ввода целого числа пользователем.



В это время сервер примет запрос на обслуживание от клиента, создаст клиентский сокет для работы с этим клиентом и создаст отдельный поток исполнения, в рамках которого будет осуществляться обработка запроса. После этого сервер перейдет в режим ожидания клиентов.

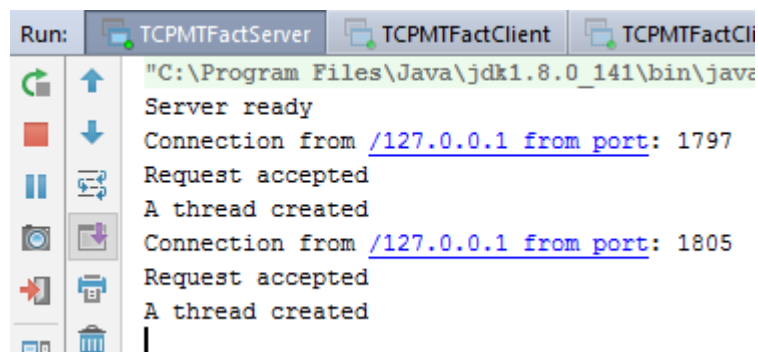


```

Run: TCPMTFactServer TCPMTFactClient
"C:\Program Files\Java\jdk1.8.0_141\bin\java"
Server ready
Connection from /127.0.0.1 from port: 1797
Request accepted
A thread created
|

```

Еще раз создадим и запустим нового клиента. Клиент будет вести себя аналогично первому, а сервер сразу примет запрос, создаст и клиентский сокет для работы с этим клиентом, и отдельный поток исполнения для обработки запроса этого клиента, и опять перейдет в режим ожидания клиентов.

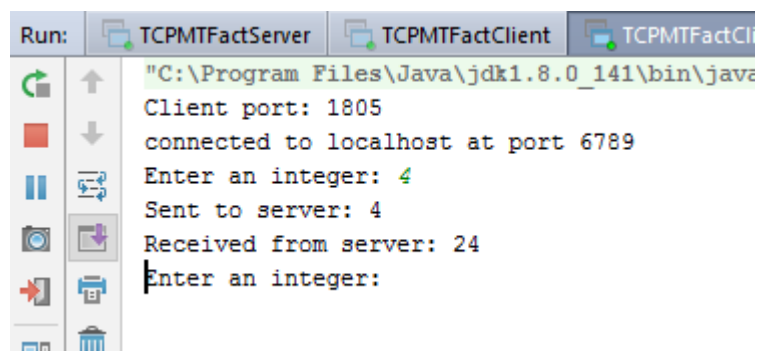


```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactClient
"C:\Program Files\Java\jdk1.8.0_141\bin\java"
Server ready
Connection from /127.0.0.1 from port: 1797
Request accepted
A thread created
Connection from /127.0.0.1 from port: 1805
Request accepted
A thread created
|

```

Когда один из клиентов (в данном случае второй, который был создан позже) передаст целое число, сервер сразу обработает его и вернет результат клиенту.

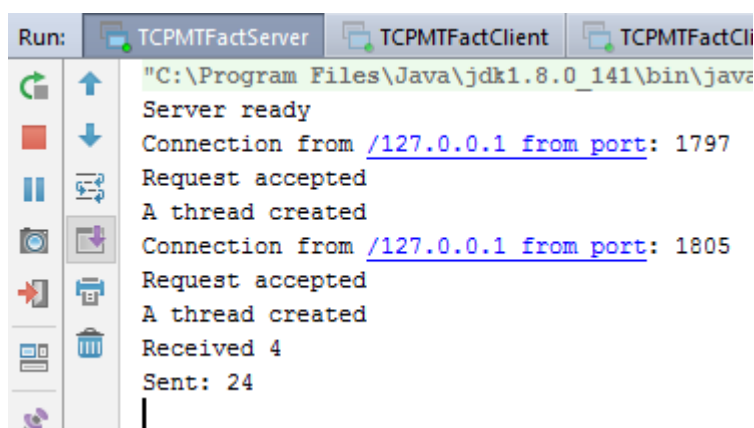


```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactClient
"C:\Program Files\Java\jdk1.8.0_141\bin\java"
Client port: 1805
connected to localhost at port 6789
Enter an integer: 4
Sent to server: 4
Received from server: 24
Enter an integer:
|

```

При этом на сервере будут выведены соответствующие диагностические сообщения.

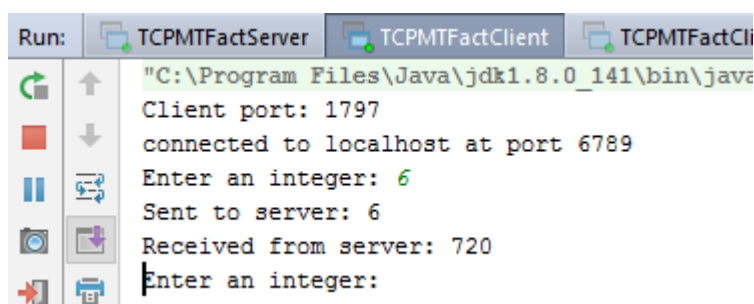


```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactCli
"C:\Program Files\Java\jdk1.8.0_141\bin\java
Server ready
Connection from /127.0.0.1 from port: 1797
Request accepted
A thread created
Connection from /127.0.0.1 from port: 1805
Request accepted
A thread created
Received 4
Sent: 24
|

```

Первый клиент тоже может передать числа на обработку; они также сразу будут обработаны. Теперь очередь подключений на сервере пуста.

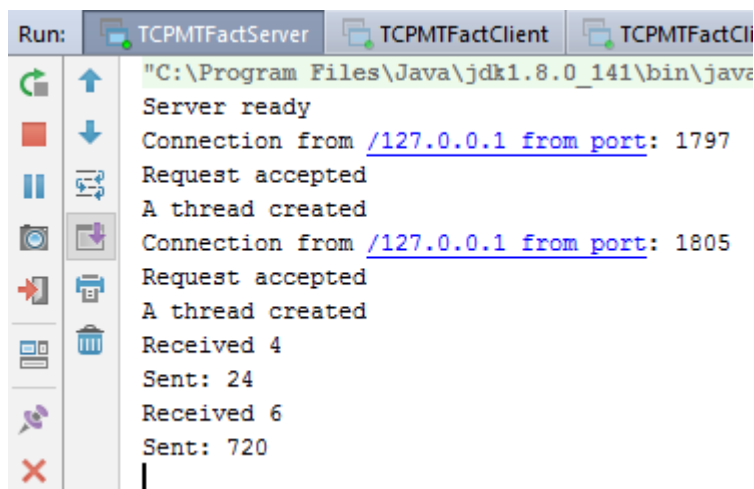


```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactCli
"C:\Program Files\Java\jdk1.8.0_141\bin\java
Client port: 1797
connected to localhost at port 6789
Enter an integer: 6
Sent to server: 6
Received from server: 720
Enter an integer:

```

Серверная часть выводит соответствующие сообщения.



```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactCli
"C:\Program Files\Java\jdk1.8.0_141\bin\java
Server ready
Connection from /127.0.0.1 from port: 1797
Request accepted
A thread created
Connection from /127.0.0.1 from port: 1805
Request accepted
A thread created
Received 4
Sent: 24
Received 6
Sent: 720
|

```

Для завершения работы с сервером клиент вводит -1. Это число передается на сервер и данный клиент завершает свою работу.

```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactCli
"C:\Program Files\Java\jdk1.8.0_141\bin\java
Client port: 1797
connected to localhost at port 6789
Enter an integer: 6
Sent to server: 6
Received from server: 720
Enter an integer: -1
Sent to server: -1

Process finished with exit code 0
|

```

После получения от клиента числа -1 на сервере завершается поток исполнения, который отвечает за работу с этим клиентом и, соответственно, закрывается соединение.

```

Run: TCPMTFactServer TCPMTFactClient TCPMTFactCli
"C:\Program Files\Java\jdk1.8.0_141\bin\java
Server ready
Connection from /127.0.0.1 from port: 1797
Request accepted
A thread created
Connection from /127.0.0.1 from port: 1805
Request accepted
A thread created
Received 4
Sent: 24
Received 6
Sent: 720
Received -1

```

### Отправка и прием объектов

Разработанные нами демонстрационные приложения обменивались строками и числами. В принципе так можно работать, передавая между компьютерами строки и числа, и на нужной стороне взаимодействия из этого конструировать объекты, которые будут нужны для работы. Но, будет гораздо удобнее, если приложения могут отправлять и получать объекты вместо потока необработанных байтов. Как мы уже разобрали, *Java* предоставляет мощную концепцию, известную как *сериализация*, которая может использоваться для отправки / получения объекта через сокет целиком. Приведем небольшой схематический демонстрационный пример, показывающий использование этой возможности.

Класс, определяющий объекты, которыми будут обмениваться стороны взаимодействия.

```
package object;
```

```

class Message implements java.io.Serializable {

    private String subject, text;

    public Message(String s, String t) {
        this.subject = s;
        this.text = t;
    }

    public String getSubject() {
        return subject;
    }

    public String getText() {
        return text;
    }
}

```

Серверный класс – приемник сообщения.

```

package object;

import java.io.*;
import java.net.*;

public class TCPObjReceiver {
    public static void main(String argv[]) throws Exception {
        //create a server socket at port 6789
        ServerSocket serverSocket = new ServerSocket(6789);
        //wait for incoming connection
        System.out.println("Server is listening on port 6789");
        Socket socket = serverSocket.accept();
        System.out.println("Request accepted");
        //Create an ObjectInputStream Object
        ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
        //Restore the object
        Message msg = (Message) in.readObject();
        //Print the message
        System.out.println("Received a message:");
        System.out.println("\tsubject : " + msg.getSubject()+"\n\tbody: " +
msg.getText());
        //Create the answer
        msg = new Message("Answer", "OK.");
        //Create an ObjectOutputStream object
        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
        //Send answer
        out.writeObject(msg);
        //Close application
        socket.close();
        serverSocket.close();
    }
}

```

Клиентская часть – отправка первого сообщения.

```

package object;

import java.io.*;
import java.net.*;

public class TCPObjSender {
    public static void main(String argv[]) throws Exception {
        String fact;

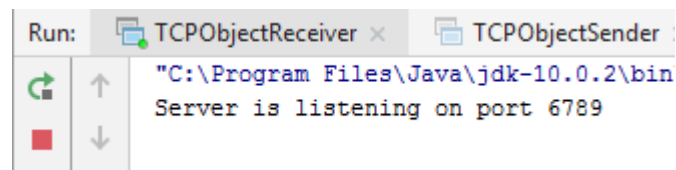
```

```

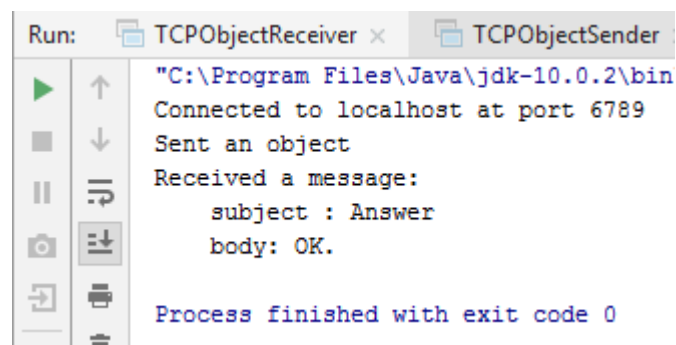
//create a socket to the server
    Socket socket = new Socket(InetAddress.getLocalHost(), 6789);
    System.out.println("Connected to localhost at port 6789");
//Create a Message object to be sent
    Message msg = new Message("Remainder", "Return my book on Monday");
//Create an ObjectOutputStream object
    ObjectOutputStream oos = new
        ObjectOutputStream(socket.getOutputStream());
//Serialize and send over TCP
    oos.writeObject(msg);
    System.out.println("Sent an object");
//Create an ObjectInputStream Object
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
//Restore the object
    msg = (Message) in.readObject();
//Print the message
    System.out.println("Received a message:");
    System.out.println("\tsubject : " + msg.getSubject()+"\n\tbody: " +
msg.getText());
    socket.close();
}
}

```

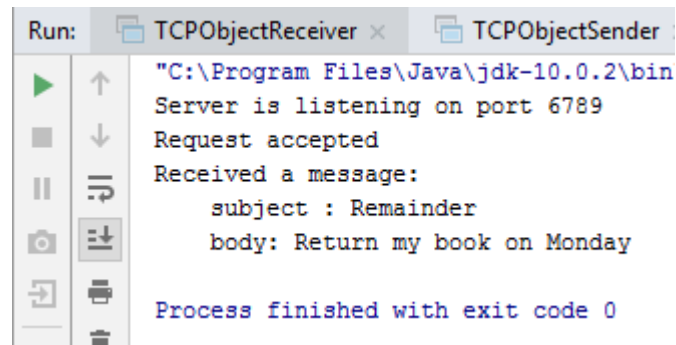
Запускаем приложение. Сначала запускается серверная часть, создается серверный сокет и ожидается подключение клиентов.



Создаем клиента, отправляем сообщение и получаем ответ.



На сервере выводятся сообщения о текущей активности.



## Пример

Рассмотрим еще один пример, показывающий создание многопоточного *TCP* сервера, работающего в отдельном потоке исполнения.

```

package simple;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class TCPConnectionRunnable implements Runnable {

    protected Socket clientSocket = null;

    public TCPConnectionRunnable(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        System.out.println("The connection from: " + clientSocket.getInetAddress() + "
host from " +
            clientSocket.getPort() + " port");
        try {
            ObjectInputStream input = new
ObjectInputStream(clientSocket.getInputStream());
            ObjectOutputStream output = new
ObjectOutputStream(clientSocket.getOutputStream());
            while(true) {
                System.out.println("Server thread is waiting for info from client");
                int n = Integer.parseInt((String) input.readObject());
                System.out.println("\tUser request n = " + n);
                if (n < 0) break;
                int fact = 1;
                for (int i = 2; i <=n; i++)
                    fact *= i;
                String res = String.valueOf(fact);
                output.writeObject(res);
                output.flush();
            }
            clientSocket.close();
            System.out.println("Request processed: ");
        } catch (Exception e) {
            //e.printStackTrace();
        } finally{
            System.out.println("Finish task server thread");
        }
    }
}

```



```

    }
}

package simple;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class TCPServer implements Runnable {

    private ServerSocket serverSocket = null;
    private boolean isStopped = false;

    public TCPServer(int port) {
        this.createServerSocket(port);
    }

    public TCPServer() {
        this.createServerSocket(6789);
    }

    private void createServerSocket(int port) {
        try {
            this.serverSocket = new ServerSocket(port, 2);
        } catch (IOException e) {
            throw new RuntimeException("Cannot open port " + port, e);
        }
    }

    private synchronized boolean isStopped() {
        return this.isStopped;
    }

    public synchronized void stop() {
        this.isStopped = true;
        try {
            this.serverSocket.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing server", e);
        }
    }

    @Override
    public void run() {
        System.out.println("Server Started");
        while(! isStopped()){
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(isStopped()) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            Thread thread = new Thread(new TCPConnectionRunnable(clientSocket));
            thread.setDaemon(true);
            thread.start();
        }
    }
}

```

```

        System.out.println("Server Stopped.") ;
    }
}

package simple;

public class TCPServerApplication {
    public static void main(String[] args) {
        TCPServer server = new TCPServer();
        Thread thread = new Thread(server);
        thread.start();
        System.out.println("The TCP Server has started. Print stop to finish");
        //try {
        //    Thread.sleep(20 * 1000);
        //} catch (InterruptedException e) {
        //    e.printStackTrace();
        //}
        java.util.Scanner in = new java.util.Scanner(System.in);
        do {
            String answer = in.next();
            if (answer.equalsIgnoreCase("stop")) break;
        } while (true);
        System.out.println("Stopping Server");
        server.stop();
    }
}

```

```

package simple;

import java.io.*;
import java.net.Socket;

public class TCPClient {
    public static void main(String argv[]) throws Exception {
        String fact;
        Socket clientEnd = new Socket("localhost", 6789);
        System.out.println("Client port: " + clientEnd.getLocalPort());
        System.out.println("connected to localhost at port 6789");
        ObjectOutputStream toServer = new
ObjectOutputStream(clientEnd.getOutputStream());
        ObjectInputStream fromServer = new
ObjectInputStream(clientEnd.getInputStream());
        BufferedReader fromUser = new BufferedReader(new
        InputStreamReader(System.in));
        while (true) {
            System.out.print("Enter an integer: ");
            String n = fromUser.readLine();
            toServer.writeObject(n);
            System.out.println("Sent to server: " + n);
            if (n.equals("-1"))
                break;
            fact = (String)fromServer.readObject();
            System.out.println("Received from server: " + fact);
        }
        clientEnd.close();
    }
}

```

Запустите приложение и разберите его работу.