

Тема: Basics of Java Network Programming (Part 2)

План занятия:

1. [Шаблон проектирования Команда](#)
2. [Пример простого распределенного приложения](#)
3. [Особенности работы с транспортным протоколом UDP](#)
 - [Класс *DatagramPacket*](#)
 - [Класс *DatagramSocket*](#)
4. [Основные этапы создания клиент / серверного приложения на UDP сокетах](#)
 - [Простой пример](#)
 - [Примеры приложений](#)
5. [Отправка и прием объектов по сети](#)

Литература

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

Шаблон проектирования Команда

При создании сетевых приложений для решения некоторых задач бывает удобно использовать подход, реализованный в шаблоне проектирования **Команда**. Хорошая статья про шаблон размещена в Википедии:

Укр:

[https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_\(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F\)](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F))

Рус:

[https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_\(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F\)](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))

Шаблон проектирования **Команда** (англ. **Command**) относится к классу поведенческих шаблонов. Известен также под такими именами, как *Действие* (англ. *Action*), *Транзакция* (англ. *Transaction*). Этот подход используется в объектно-ориентированном программировании для представления действия. Объект команды заключает в себе само действие и его параметры. Назначение этого шаблона – создания такой структуры, в которой класс-отправитель и класс-получатель не зависят друг от друга напрямую, а также организация обратного вызова к классу, который включает в себя класс-отправитель. Инкапсулирует запрос в форме объекта, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмена операций.

В объектно-ориентированном программировании шаблон проектирования Команда является поведенческим шаблоном, в котором объект используется для инкапсуляции всей информации, необходимой для выполнения действия или вызова события в более позднее время. Эта информация включает в себя имя метода, объект, который является владельцем метода и значения параметров метода.

Четыре термина всегда связаны с шаблоном **Команда**: *команды* (*command*), *приёмник команд* (*receiver*), *вызывающий команды* (*invoker*) и *клиент* (*client*). Объект *Command* знает о приёмнике и вызывает метод приемника. Значения параметров приёмника сохраняются в команде. *Вызывающий объект* (*invoker*) знает, как выполнить команду и, возможно, делает учёт и запись выполненных команд. *Вызывающий объект* (*invoker*) ничего не знает о конкретной команде, он знает только об интерфейсе. Оба объекта (*вызывающий объект* и несколько объектов команд) принадлежат объекту *клиента* (*client*). *Клиент* решает, какие команды выполнить и когда. Чтобы выполнить команду он передает объект команды *вызывающему объекту* (*invoker*).

Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода. Использование вызывающего объекта (*invoker*) позволяет ввести учет выполненных команд без необходимости знать клиенту об этой модели учета (такой учет может пригодиться, например, для реализации отмены и повтора команд).

Следует использовать шаблон *Команда* когда:

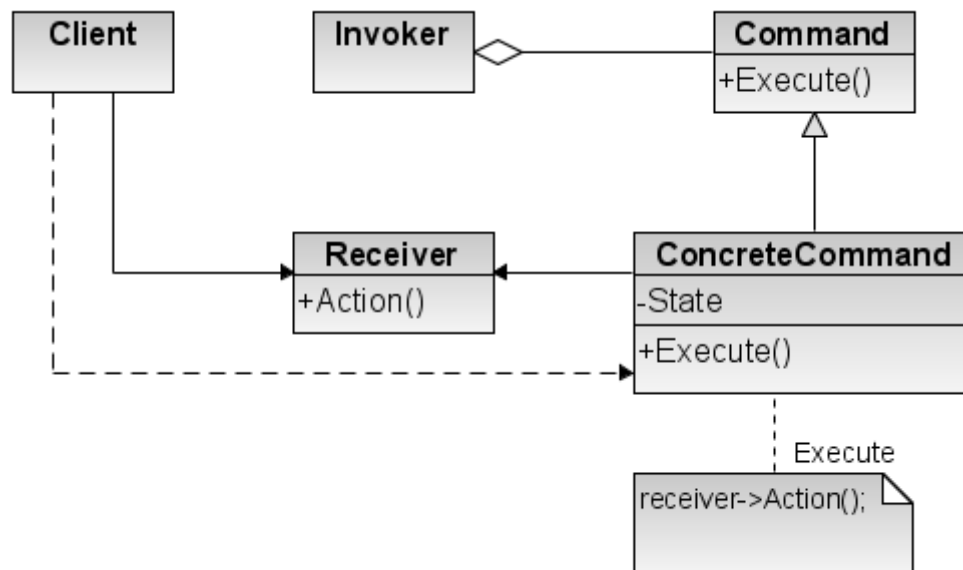
- надо параметризовать объекты действием. В процедурном языке такую параметризацию можно выразить с помощью функции обратного

вызова, то есть такой функцией, которая регистрируется, чтобы быть вызванной позже. Команды являются объектно-ориентированной альтернативой функциям обратного вызова.

- определять, ставить в очередь и выполнять запросы в разное время. Срок жизни объекта Команда не обязательно зависит от срока жизни начального запроса. Если получателя удастся реализовать таким образом, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
- нужна поддержка отмена операций. Операция *Execute* объекта *Команда* может сохранить состояние, необходимое для отмены действий, выполненных *Командой*. В этом случае в интерфейсе класса *Command* должна быть дополнительная операция *Unexecute*, которая отменяет действия, выполненные предыдущим вызовом операции *Execute*. Выполненные команды сохраняются в журнале. Для реализации произвольного количества уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента операцию *Unexecute* или *Execute*;
- поддерживать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса *Command* операциями хранения и загрузки, можно вести протокол изменений во внутренней памяти. Для восстановления после сбоя надо будет загрузить сохранения команды с диска и повторно выполнить их с помощью операции *Execute*;
- надо структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует множество изменений данных. Шаблон Команда позволяет моделировать транзакции. Во всех команд есть общий интерфейс, позволяет работать одинаково с любыми транзакциями. С помощью этого шаблона можно легко добавлять в систему новые виды транзакций.

Есть много вариантов применения. В сетевом программировании шаблон *Команда* может быть полезен для отправки объектов команд по сети для выполнения на другой машине.

Рассмотрим структуру шаблона.



Command - команда:

- объявляет интерфейс для выполнения операции;

ConcreteCommand - конкретная команда:

- определяет связь между объектом-получателем **Receiver** и действием
- реализует операцию **Execute** путем вызова соответствующих операций объекта **Receiver**;

Client - клиент:

- создает объект класса **ConcreteCommand** и устанавливает его получателя;

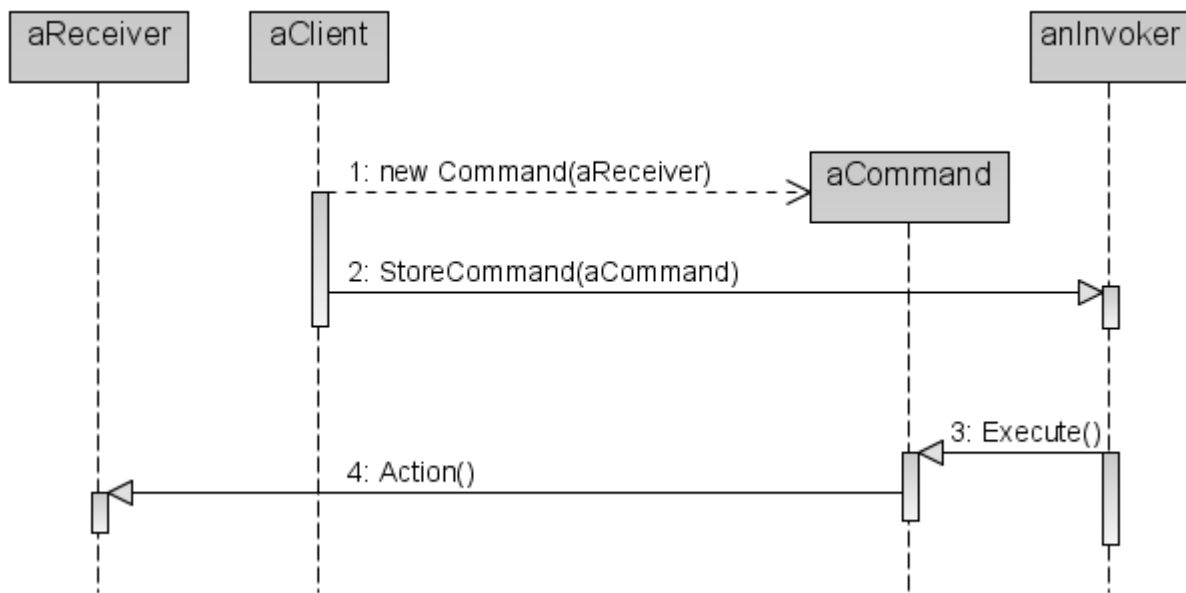
Invoker - вызывающего абонента:

- обращается к команде чтобы она исполнила запрос;

Receiver - получатель:

- располагает всю информацию о способах выполнения операций, необходимых для удовлетворения запроса. В качестве получателя может выступать любой класс.

Рассмотрим взаимоотношения



1. клиент создает объект ConcreteCommand и устанавливает для него получателя;
2. «вызыватель» Invoker сохраняет объект ConcreteCommand;
3. «вызыватель» посылает запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отмены;
4. объект ConcreteCommand вызывает операции получателя для выполнения запроса

На диаграмме видно, как Command разрывает связь между вызывателем и получателем (а также запросом, который должен быть выполнен последним).

Преимущества:

- Отделяет классы, которые вызывают операцию от объекта, который умеет выполнять операцию
- Позволяет создавать последовательность команд с помощью системы очереди
- Расширение для добавления новой команды просто и могут быть выполнены без изменения существующего кода
- Вы также можете определить систему отката с командным шаблоном, например, в примере мастера, мы можем написать метод отката

Недостатки:

Увеличение количества классов для каждой отдельной команды

Рассмотрим пример реализации шаблона

```
package command;
```

```
/** The Command interface */
interface Command {
    void execute();
}
```

```
package command;
```

```
/** The Receiver class */
class Light {
    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}
```

```
package command;
```

```
import java.util.HashMap;
```

```
/** The Invoker class */
class Switch {
    private final HashMap<String, Command> commandMap = new HashMap<>();

    public void register(String commandName, Command command) {
        commandMap.put(commandName, command);
    }

    public void execute(String commandName) {
        Command command = commandMap.get(commandName);
        if (command == null) {
            throw new IllegalStateException("no command registered for " +
commandName);
        }
        command.execute();
    }
}
```

```
package command;
```

```
/** The Command for turning off the light - ConcreteCommand #2 */
class SwitchOffCommand implements Command {
    private final Light light;

    public SwitchOffCommand(Light light) {
        this.light = light;
    }

    @Override // Command
```

```

        public void execute() {
            light.turnOff();
        }
    }

package command;

/** The Command for turning on the light - ConcreteCommand #1 */
class SwitchOnCommand implements Command {
    private final Light light;

    public SwitchOnCommand(Light light) {
        this.light = light;
    }

    @Override // Command
    public void execute() {
        light.turnOn();
    }
}

```

```

package command;

public class CommandDemo {
    public static void main(final String[] arguments) {
        //Old
        Light lamp = new Light();

        Command switchOn = new SwitchOnCommand(lamp);
        Command switchOff = new SwitchOffCommand(lamp);

        Switch mySwitch = new Switch();
        mySwitch.register("on", switchOn);
        mySwitch.register("off", switchOff);

        mySwitch.execute("on");
        mySwitch.execute("off");

        //New style
        /*
        Light lamp = new Light();

        Command switchOn = lamp::turnOn;
        Command switchOff = lamp::turnOff;

        Switch mySwitch = new Switch();
        mySwitch.register("on", switchOn);
        mySwitch.register("off", switchOff);

        mySwitch.execute("on");
        mySwitch.execute("off");
        */
    }
}

```

Пример простого распределенного приложения

Сейчас мы разработаем простое демонстрационное распределенное приложение с использованием *TCP* сокетов. Приложение будет состоять из

двух согласованно работающих частей: клиентской и серверной. Сервер принимает произвольные задания от клиентов, локально выполняет эти задания, вычисляет время выполнения и, в конце, возвращает клиенту результат вычисления и время расчета. Клиент готовит задачи для вычисления, отправляет их серверу и ожидает результат расчетов. Таким образом, клиенты могут выполнять свои задания удаленно на специально для этого выделенном компьютере.

Особенностью этого приложения является то, что сервер предварительно не должен знать те задания, которые будет выполнять. У клиентов есть возможность создавать свои собственные задания и отправлять их на сервер для выполнения. Для реализации этой возможности класс задания должен реализовывать интерфейс, а сервер должен быть готовым к обработке любого класса, который реализовывает этот интерфейс. Так как в *Java* необходимые для работы классы загружаются не сразу все при старте приложения, а только по необходимости, то клиент перед отправкой объекта-задания может через этот же сокет передать на сервер *class* - файл задания. После того, как файл с байт-кодом класса задания передан, сервер может выполнить задание локально.

Так как тип возвращаемого сервером результата не является встроенным, то для отправки результатов расчетов обратно клиенту можно выполнить ту же самую процедуру. Клиент может загрузить файл с байт-кодом класса результата с сервера и затем обработать результат. Требование к классу результата такое же, как и к классу задания: класс результата реализует интерфейс, известный и клиенту, и серверу.

Таким образом, сервер может выполнять любые, заранее неизвестные ему задания. Клиенты также могут получить результат без предварительного знания определения класса результата. Сокет используется и для отправки определений классов, и для последующей отправки объектов этих классов. Таким образом, можно создать объект (задания / результата) и динамически установить его на удаленной машине.

Отсюда можно сделать вывод о структуре приложения: серверная часть приложения, клиентская часть приложения и набор интерфейсов, определяющих обязанности задания и результата.

При разработке этого распределенного приложения необходимо смоделировать ситуацию, когда разработка ведется двумя командами разработчиков. Можно использовать два одновременно запущенных экземпляра интегрированной среды разработки. Но в лекции, для упрощения, будем работать без *IDE*.

Начнем работу над приложением с разработки интерфейсов. Потом разработанные интерфейсы передадим командам разработчиков, котоые

работают над реализацией клиента и сервера. В интерфейсе, описывающем задание (Executable), будет только один метод, определяющий задание.

```
public interface Executable {  
    Object execute();  
}
```

Все параметры, которые нужны для работы задания будут храниться в виде полей класса, реализующего интерфейс Executable. Объекты заданий будут переданы с клиентов на сервер при помощи механизма объектной сериализации. Сервер де-сериализует полученный по сети объект и вызовет на этом объекте единственный метод интерфейса execute(). Метод определен так, что может вернуть любой результат, а клиент знает, что должно быть получено в результате выполнения задания.

Интерфейс Result описывает структуру результата, который будет возвращен клиенту после выполнения задания.

```
public interface Result {  
    Object output();  
    double computeTime();  
}
```

В интерфейсе определено два метода: метод output(), возвращающий вычисленный результат и метод computeTime(), возвращающий время, затраченное на вычисление задания на сервере. Объект класса, реализующего этот интерфейс, будет сформирован на сервере, при помощи механизма сериализации передан по сети обратно клиенту, восстановлен и использован требуемым образом.

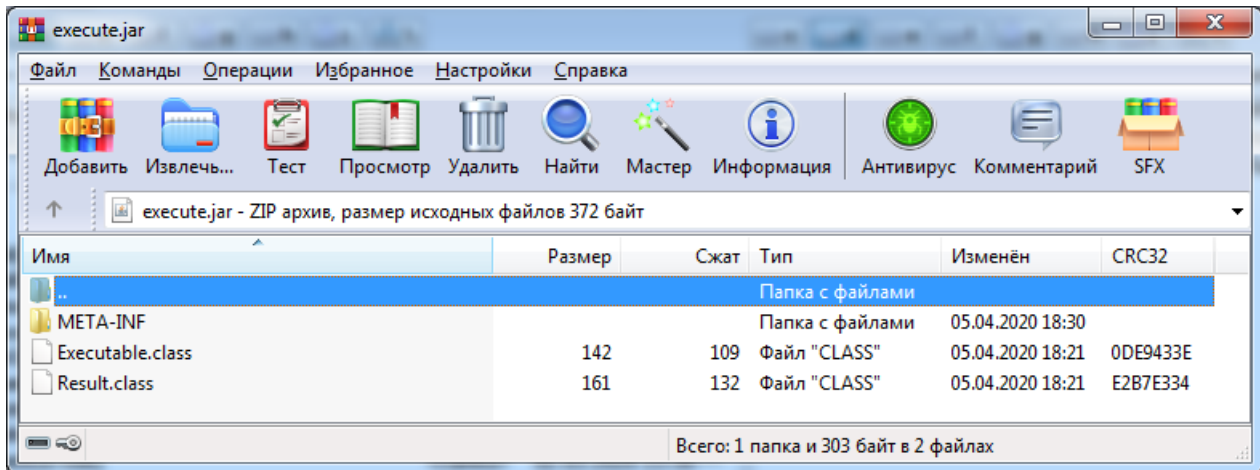
При разработке вне *IDE* создадим директорию Interface, в ней создадим два исходных java файла с указанными интерфейсами. Находясь в заданной директории, выполним компиляцию исходных файлов:

```
javac -cp . *.java
```

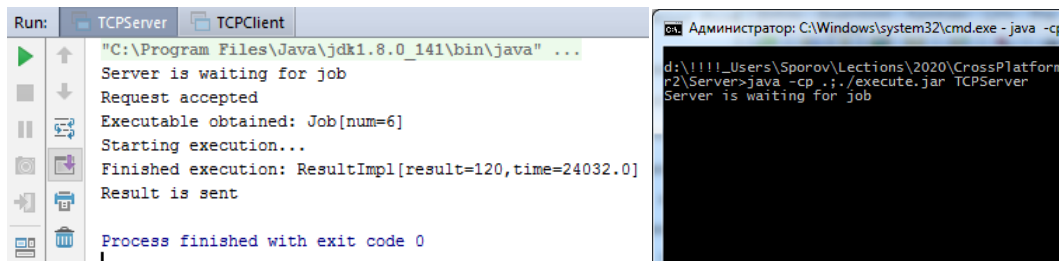
В результате успешной компиляции в текущей директории будут созданы два class файла. На следующем шаге нужно упаковать эти файлы с байт-кодами в jar архив. Для этого, находясь в директории с откомпилированными файлами нужно выполнить следующую команду архивации:

```
jar cvf execute.jar *.class
```

В результате, в данной директории будет сформирован архив `execute.jar`, с таким содержимым:



Интерфейс, описывающий результат,



Это архив «передадим» командам разработчиков, которые работают над сервером и клиентом.

Для создания сервера рядом с директорией `Interface` создадим директорию `Server` и перейдем в нее. Запишем в эту директорию файл с интерфейсами `execute.jar` и в этой директории разместим исходные файлы с определением класса сервера и класса результата.

Сначала определим класс результата (`ResultImpl`), объект которого будет создан на сервере после завершения вычислений и передан обратно тому клиенту, от которого было получено задание. Данный класс должен имплементировать интерфейс результата `Result` и интерфейс `Serializable`:

```
import java.io.Serializable;
```

```

public class ResultImpl implements Serializable, Result {
    private Object result;
    private double time;

    public ResultImpl(Object result, double time) {
        this.result = result;
        this.time = time;
    }

    @Override
    public Object output() {
        return result;
    }

    @Override
    public double computeTime() {
        return time;
    }

    @Override
    public String toString() {
        return
this.getClass().getSimpleName()+"[result="+result+",time="+time+"]";
    }
}

```

Так как объект, представляющий результат вычислений, будет сериализован, передан по сети и должен быть успешно восстановлен на клиентском компьютере, то нужно сделать доступным для клиента (разместить в одной из тех директорий, где он ищет классы) определение (файл с байт-кодом) класса **ResultImpl**. Это можно сделать, передав по сети от сервера на клиентский компьютер через сокет сначала полное имя, а затем и содержимое файла **ResultImpl.class**. Клиент, после того, как получил полное имя файла и его содержимое, должен сохранить полученное в правильной директории. Только после этого клиент будет готов получить по сети сериализованный образ объекта-результата и десериализовать этот объект. Аналогично, клиент и сервер должны обработать задание для расчета.

Создадим файл с исходным кодом сервера:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class TCPServer {
    public static void main(String[] args) throws Exception {
        ServerSocket serverSocket = new ServerSocket(6789);
        System.out.println("Server is waiting for job");
        Socket socket = serverSocket.accept();
        System.out.println("Request accepted");
        //obtain job
        ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());
        //class name
        String classFileName = (String) in.readObject();
        //class file
        byte[] classFileContent = (byte[])in.readObject();
        FileOutputStream fos = new FileOutputStream(classFileName);
        fos.write(classFileContent);
        //job
        Executable exec = (Executable) in.readObject();
        //calculation
        System.out.println("Executable obtained: "+exec);
        System.out.println("Starting execution...");
        double startTime = System.nanoTime();
        Object result = exec.execute();
        double endTime = System.nanoTime();
        ResultImpl answer = new ResultImpl(result, endTime-startTime);
        System.out.println("Finished execution: "+answer);
        //send answer
        ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
        //class name
```

```

        //className =
        ".\\out\\production\\JavaNetLec2\\lec_test0\\ResultImpl.class";
        className = "ResultImpl.class";
        out.writeObject(className);
        //class file
        FileInputStream fis = new FileInputStream(className);
        classFileContent = new byte[fis.available()];
        fis.read(classFileContent);
        out.writeObject(classFileContent);
        //answer
        out.writeObject(answer);
        System.out.println("Result is sent");
        socket.close();
        serverSocket.close();
    }
}

```

В данном примере закомментированная строка `className =`, указывает на месторасположение передаваемого файла с байт кодом в случае, если разработка ведется в *IntelliJ IDEA*.

Находясь в заданной директории, в консольном окне вызовем команду компиляции *.java файлов.

```
javac -cp .;execute.jar *.java
```

Следует обратить внимание, что в `classpath` включается не только текущая директория, но и определения интерфейсов, хранящиеся в архиве `execute.jar`. Они понадобятся не только при компиляции приложения, но и при его запуске. В результате успешной компиляции будут созданы `class` файлы класса результата и сервера.

Для создания клиента рядом с директориями `Interface` и `Server` создадим директорию `Client` и перейдем в нее. Запишем в эту директорию файл с интерфейсами `execute.jar` и в этой директории разместим исходные файлы с определением класса клиента и класса задания.

Структура этих файлов похожа на структуру аналогичных файлов на серверной стороне.

Класс задание:

```
import java.io.Serializable;

public class Job implements Serializable, Executable {
    private int num;

    public Job() {
        this(0);
    }

    public Job(int num) {
        this.num = num;
    }

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    @Override
    public Object execute() {
        int res = 1;
        for (int i = 2; i < this.num; i++) {
            res *= i;
        }

        return new Integer(res);
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + "[num=" + num + "];"
    }
}
```

Класс с определением клиента:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;

public class TCPClient {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket(InetAddress.getLocalHost(), 6789);
        System.out.println("Connected to localhost at port 6789");
        //send job
        ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
        //class name
        //String classFileName =
"./out/production\\JavaNetLec2\\lec_test0\\Job.class";
        String classFileName = "Job.class";
        out.writeObject(classFileName);
        //class file
        FileInputStream fis = new FileInputStream(classFileName);
        byte[] classFileContent = new byte [fis.available()];
        fis.read(classFileContent);
        out.writeObject(classFileContent);
        //job
        Job job = new Job(6);
        out.writeObject(job);
        System.out.println("Submitted the job for execution");
        //obtain answer
        ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());
        //class name
        classFileName = (String) in.readObject();
        //class file
        classFileContent = (byte[])in.readObject();
        FileOutputStream fos = new FileOutputStream(classFileName);
        fos.write(classFileContent);
```

```

//answer
Result res = (Result)in.readObject();
System.out.println("Obtained result object: "+res);
System.out.println("\tValue: "+res.output());
System.out.println("\tCalculation time: "+res.computeTime());
socket.close();
}
}

```

Как и для сервера, в данном примере закомментированная строка `classFileName =`, указывает на месторасположение передаваемого файла с байт кодом в случае, если разработка ведется в *IntelliJ IDEA*.

Так же, как и при компиляции сервера, находясь в данной директории нужно выполнить команду компиляции с указанным параметром `classpath`:

```
javac -cp .;execute.jar *.java
```

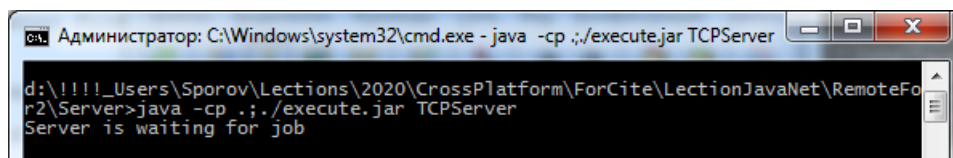
В случае успеха будут сгенерированы все `class` файлы, требуемые для запуска приложения.

Выполним запуск программы не из среды.

Для запуска сервера перейдем в директорию `Server` и запустим серверную сторону приложения при помощи команды:

```
java -cp .;./execute.jar TCPServer
```

Запустится сервер, который будет готов к запросам клиентов:



Для запуска клиента перейдем в директорию `Client` и запустим клиентскую сторону приложения при помощи команды:

```
java -cp .;./execute.jar TCPClient
```

Будет сформирован объект-задание, передан на сервер и после вычислений получен объект – результат:


```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaNet\RemoteFo
r2\Client>java -cp ../execute.jar TCPClient
Connected to localhost at port 6789
Submitted the job for execution
Obtained result object: ResultImpl[result=120,time=146499.0]
Value: 120
Calculation time: 146499.0

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaNet\RemoteFo
r2\Client>

```

Аналогичная последовательность действий был выполнена и на сервере.

```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaNet\RemoteFo
r2\Server>java -cp ../execute.jar TCPServer
Server is waiting for job
Request accepted
Executable obtained: Job[num=6]
Starting execution...
Finished execution: ResultImpl[result=120,time=146499.0]
Result is sent

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaNet\RemoteFo
r2\Server>

```

Для упрощения работы все команды можно сохранить в соответствующих командных *.bat файлах.

Нужно обратить внимание на передачу *.class файлов по сети. На рисунках показано содержимое директории сервера (слева – до взаимодействия, справа – после). Видно, что *.class файл задания (Job.class) действительно был передан по сети.

```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForC
r2\Server>dir
Том в устройстве D не имеет метки.
Серийный номер тома: 7773-B210

Содержимое папки d:\!!!!_Users\Sporov\Lectons\2020\
onJavaNet\RemoteFor2\Server

04.04.2020 23:36 <DIR> .
04.04.2020 23:36 <DIR> ..
04.04.2020 21:31      823 execute.jar
04.04.2020 21:34      953 ResultImpl.class
04.04.2020 21:05      568 ResultImpl.java
04.04.2020 21:34      1 969 TCPServer.class
04.04.2020 21:34      2 044 TCPServer.java
                5 файлов        6 357 байт
                2 папок      61 281 136 640 байт свободно

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForC
r2\Server>

```

```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForC
r2\Server>dir
Том в устройстве D не имеет метки.
Серийный номер тома: 7773-B210

Содержимое папки d:\!!!!_Users\Sporov\Lectons\2020\
onJavaNet\RemoteFor2\Server

04.04.2020 23:37 <DIR> .
04.04.2020 23:37 <DIR> ..
04.04.2020 21:31      823 execute.jar
04.04.2020 23:37      995 Job.class
04.04.2020 21:34      953 ResultImpl.class
04.04.2020 21:05      568 ResultImpl.java
04.04.2020 21:34      1 969 TCPServer.class
04.04.2020 21:34      2 044 TCPServer.java
                6 файлов        7 352 байт
                2 папок      61 281 128 448 байт свободно

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForC
r2\Server>

```

На приведенных ниже рисунках показано содержимое директории клиента (слева – до взаимодействия, справа – после). Видно, что *.class файл результата (ResultImpl.class) действительно был передан по сети.

```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForC
r2\Client>dir
Том в устройстве D не имеет метки.
Серийный номер тома: 7773-B210

Содержимое папки d:\!!!!_Users\Sporov\Lections\2020\
onJavaNet\RemoteFor2\Client

04.04.2020  23:36    <DIR>          .
04.04.2020  23:36    <DIR>          ..
04.04.2020  21:31             823 execute.jar
04.04.2020  21:34             995 Job.class
04.04.2020  21:06             661 Job.java
04.04.2020  21:34             1 956 TCPClient.class
04.04.2020  21:34             1 749 TCPClient.java
                    5 файлов             6 184 байт
                    2 папок             61 281 136 640 байт свободно

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForC
r2\Client>

```

```

Администратор: C:\Windows\system32\cmd.exe

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForC
r2\Client>dir
Том в устройстве D не имеет метки.
Серийный номер тома: 7773-B210

Содержимое папки d:\!!!!_Users\Sporov\Lections\2020\
onJavaNet\RemoteFor2\Client

04.04.2020  23:37    <DIR>          .
04.04.2020  23:37    <DIR>          ..
04.04.2020  21:31             823 execute.jar
04.04.2020  21:34             995 Job.class
04.04.2020  21:06             661 Job.java
04.04.2020  23:37             953 ResultImpl.class
04.04.2020  21:34             1 956 TCPClient.class
04.04.2020  21:34             1 749 TCPClient.java
                    6 файлов             7 137 байт
                    2 папок             61 281 128 448 байт свободно

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForC
r2\Client>

```

Данное приложение допускает простое расширение. Можно определить новый класс – задание и, согласно указанной выше процедуре, передать его на сервер. На сервере можно подготовить новый тип для объекта-результата, более подходящего для хранения результата нового задания, и так же передать его клиенту.

Особенности работы с транспортным протоколом UDP

На прошлой лекции мы обсуждали сетевые приложения, использующие протокол *TCP*. Этот протокол предназначен для надежной передачи данных. Если данные будут потеряны или повреждены при передаче, протокол *TCP* гарантирует, что данные будут отправлены повторно; если пакеты данных приходят не в должном порядке, *TCP* возвращает их в правильном порядке; если данные поступают слишком быстро для соединения, *TCP* снижает скорость, чтобы пакеты не были потеряны. Программе не нужно специально беспокоиться о получении данных, которые пришли не в правильном порядке или повреждены. Однако, за такую надежность передачи данных имеет свою цену - скорость. Установление и разрыв соединений *TCP* может занять достаточно много времени.

Для некоторых задач (например, передача потокового видео, аудио и т.д.) может быть полезно использовать протокол пользовательских датаграмм (англ. *User Datagram Protocol* — *протокол пользовательских датаграмм*. см. <https://docs.oracle.com/javase/tutorial/networking/datagrams/index.html>). От протокола *TCP* он отличается тем, что работает без установления соединения. *UDP* - это один из самых простых протоколов транспортного уровня, который выполняет обмен сообщениями (называемые *датаграммами* - англ. *Datagram*) без подтверждения и гарантии доставки. То есть, когда вы отправляете данные в соответствии с протоколом *UDP*, нет возможности узнать, поступили ли они вообще, тем более, поступили ли разные фрагменты

данных в том порядке, в котором были отправлены. Тем не менее, доставка, как правило, происходит быстро. Таким образом, *UDP* предоставляет ненадежный сервис. Протокол *UDP* подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Протокол был разработан Дэвидом П. Ридом в 1980 году и официально определен в *RFC 768*. Он рекомендуется к использованию для чувствительных ко времени приложений, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени. Кроме того, протокол *UDP* также может быть полезен для серверов, отвечающих на небольшие запросы от огромного числа клиентов, например *DNS* и потоковые мультимедийные приложения вроде *IPTV*, *Voice over IP*, протоколы туннелирования *IP* и многие онлайн-игры.

Сравнение UDP и TCP протоколов

Кратко приведем основные сравнительные характеристики *TCP* и *UDP* протоколов.

TCP — протокол, ориентированный на соединение; это значит, что хосты должны выполнить последовательность «рукопожатия» (англ. *handshake*) для установки. Как только соединение установлено, пользователи могут отправлять данные в обоих направлениях. Его основные характерные черты:

Надежность — протокол *TCP* сам управляет подтверждением, повторной передачей и тайм-аутом сообщений. Производятся многочисленные попытки доставить сообщение. Если оно потеряется на пути, сервер вновь запросит потерянную часть. При использовании *TCP* протокола нет пропавших данных.

Упорядоченность — протокол *TCP* гарантирует правильный порядок данных: если два сообщения последовательно отправлены, первое сообщение достигнет приложения-получателя первым. Если участки данных прибывают в неверном порядке, *TCP* отправляет неупорядоченные данные в буфер до тех пор, пока все данные не могут быть упорядочены и переданы приложению.

Тяжеловесность — протоколу *TCP* необходимо три пакета для установки сокет-соединения перед тем, как отправить данные. *TCP* следит за надежностью и перегрузками.

Потоковость — при использовании *TCP* протокола данные рассматриваются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

UDP — более простой, основанный на сообщениях протокол без установления соединения. Протоколы такого типа не устанавливают выделенного соединения между двумя хостами. Связь достигается путем передачи информации в одном направлении от источника к получателю без

проверки готовности или состояния получателя. Его основные характерные черты:

Ненадежный — когда сообщение посылается, неизвестно, достигнет ли оно своего назначения — оно может потеряться по пути. Нет таких понятий, как подтверждение, повторная передача, тайм-аут.

Неупорядоченность — если два сообщения отправлены одному получателю, то порядок их достижения цели не может быть предугадан.

Легковесность — никакого упорядочивания сообщений, никакого отслеживания соединений и т. д. Это небольшой транспортный уровень, работающий непосредственно поверх *IP*.

Датаграммы — пакеты посылаются по отдельности и проверяются на целостность только если они прибыли. Пакеты имеют определенные границы, которые соблюдаются после получения, то есть операция чтения на сокете-получателе выдаст сообщение таким, каким оно было изначально послано.

Нет контроля перегрузок — *UDP* сам по себе не избегает перегрузок. Для приложений с большой пропускной способностью это может быть проблемой, если только они самостоятельно не реализуют меры контроля на прикладном уровне.

Программная реализация протокола *UDP*

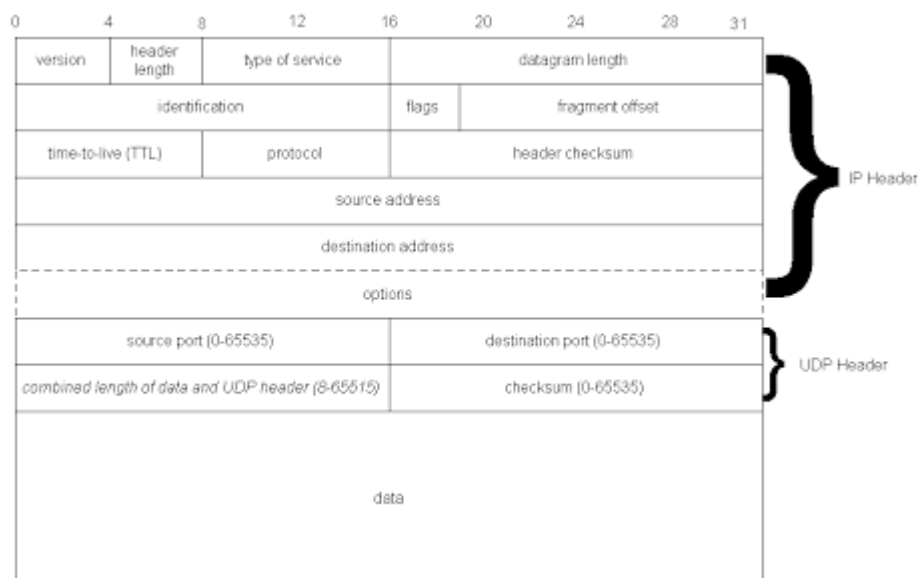
В *Java* реализация протокола *UDP* состоит из двух классов: *DatagramPacket* и *DatagramSocket*. Класс *DatagramPacket* представляет данные, передаваемые по сети — пакеты *UDP*, называемые *датаграммами*. Класс *DatagramSocket* предоставляет средства для отправки и получения датаграмм. Для того, чтобы отправить данные, нужно поместить их в объект *DatagramPacket* и затем отправить пакет при помощи объекта *DatagramSocket*. Для того, чтобы получить данные, нужно получить объект *DatagramPacket* средствами *DatagramSocket*, а затем извлечь содержимое пакета. Датаграммные сокеты очень простые. В рамках *UDP* протокола все, что касается датаграммы, включая адрес назначения, инкапсулировано в пакете; сокету нужно знать только локальный порт для получения или отправки датаграм.

Такое разделение ответственности отличается от модели *TCP*, реализованной классами *Socket* и *ServerSocket*. Во-первых, при использовании *UDP* нет такого понятия, как соединение между двумя хостами. Один *UDP* сокет получает все пакеты, направленные на прослушиваемый им порт или отправляет их с него, не беспокоясь о том, с каким удаленным хостом осуществляется взаимодействие. Один объект *DatagramSocket* может взаимодействовать (и отправлять, и получать данные) с произвольным количеством независимых хостов. Определение того, какой же хост отправил

данные, является ответственностью приложения. Во-вторых, в отличие от *TCP* сокетов, которые работают с сетевым соединением как потоком (отправка и получение данных происходит посредством потоков ввода и вывода, которые нужно получить с помощью сокета), *UDP* сокет всегда работает с отдельными пакетами датаграмм. Все данные, которые помещены в одну датаграмму, отправляются в одном пакете и либо принимаются получателем, либо теряются по дороге. Один пакет не обязательно должен быть связан со следующим. При последовательном получении двух пакетов невозможно определить, какой пакет был отправлен первым, а какой - вторым. Вместо упорядоченной очереди данных, характерной для потока, датаграммы стараются как можно быстрее добраться до получателя. При этом разные датаграммы могут добираться до одного получателя по разным маршрутам.

Класс **DatagramPacket**

Датаграммы *UDP* добавляют очень мало к *IP* датаграммам, поверх которых они работают. На рисунке чуть ниже показана типичная структура *UDP* датаграммы. Заголовок *UDP* датаграммы добавляет только восемь байт к заголовку *IP* датаграммы. *UDP* заголовок включает номера портов источника и назначения, длину всего того, что следует за *IP* заголовком, и необязательную контрольную сумму. Поскольку номера портов задаются как двухбайтные целые числа без знака, для каждого хоста доступно 65 536 различных возможных портов *UDP*. Они отличаются от 65 536 различных портов *TCP* на том же хосте. Поскольку длина (данных и *UDP* заголовка) также является двухбайтным целым числом без знака, размер данных в датаграмме ограничен 65 536 минус восемь байтов для заголовка. Это значение превышает возможный размер исходя заголовка *IP*, где вводится предельный размер датаграммы в диапазоне между 65 467 и 65 507 байтами. (Точное значение зависит от размера *IP* заголовка). Поле контрольной суммы является необязательным и обычно не используется. Если не совпала контрольная сумма для данных, то встроенное нативное сетевое программное обеспечение тихо отбрасывает датаграмму; ни отправитель, ни получатель датаграммы при этом не уведомляются.



Хотя, согласно теории, максимальный объем данных в *UDP* датаграмме составляет **65 507** байт, на практике их размер намного меньше. На многих платформах реальный предел составлял **8192** байта (8 КБ). На практике также следует учитывать, что если длина *IPv4* пакета с *UDP* будет превышать *MTU* (англ. *maximum transmission unit* - максимальная единица передачи - максимальный размер полезного блока данных одного пакета, который может быть передан протоколом без фрагментации, для *Ethernet* по умолчанию **1500** байт), то отправка такого пакета может вызвать его фрагментацию, что может привести к тому, что он вообще не сможет быть доставлен, если промежуточные маршрутизаторы или конечный хост не будут поддерживать фрагментированные *IP* пакеты. Также в *RFC 791* указывается минимальная длина *IP* пакета **576** байт, которую должны поддерживать все участники *IPv4*, и рекомендуется отправлять *IP* пакеты большего размера только в том случае если вы уверены, что принимающая сторона может принять пакеты такого размера. Следовательно, чтобы избежать фрагментации *UDP* пакетов (и возможной их потери), размер данных в *UDP* не должен превышать: *MTU* — (*Max IP Header Size*) — (*UDP Header Size*) = $1500 - 60 - 8 = 1432$ байт. Для того чтобы быть уверенным, что пакет будет принят любым хостом, размер данных в *UDP* не должен превышать: (*минимальная длина IP пакета*) — (*Max IP Header Size*) — (*UDP Header Size*) = $576 - 60 - 8 = 508$ байт.

В современных сетях допускаются *Jumbogram IPv6 UDP* пакеты, которые могут иметь больший размер. Максимальное значение составляет **4 294 967 295** байт ($2^{32} - 1$), из которых **8** байт соответствуют заголовку, а остальные **4 294 967 287** байт — данным. Следует заметить, что большинство современных сетевых устройств отправляют и принимают пакеты *IPv4* длиной до **10000** байт без их разделения на отдельные пакеты. Неофициально такие пакеты называют «*Jumbo-пакетами*», хотя понятие *Jumbo* официально

относится к *IPv6*. Тем не менее, «*Jumbo-пакеты*» поддерживают не все устройства и перед организацией связи с помощью *UDP IPv4* пакетов с длиной, превышающей 1500 байт, нужно проверять возможность такой связи опытным путем на конкретном оборудовании. В общем, нужно быть крайне осторожны с любой программой, которая зависит от отправки или получения пакетов *UDP* с более чем 8 КБ данных.

В *Java* датаграмма *UDP* представлена экземпляром класса `DatagramPacket`:

```
public final class DatagramPacket extends Object
```

Этот класс предоставляет методы для получения и установки адреса источника или назначения для заголовка *IP*, методы для получения и установки порта источника или назначения, методы для получения и установки данных, а также методы для получения и установки длины данных. Остальные поля заголовка датаграммы недоступны из *Java* кода.

Сейчас рассмотрим основные возможности, предоставляемые классом; о всех возможностях можно узнать из документации на сайте *Oracle*: <https://docs.oracle.com/javase/8/docs/api/java/net/DatagramPacket.html>

Конструкторы

При создании объектов класса `DatagramPacket` используются различные конструкторы в зависимости от того, каким образом будет использоваться созданный пакет: для отправки или получения данных. В классе есть шесть конструкторов, и все они принимают в качестве параметров байтовый массив (`byte[]`), в котором хранятся данные датаграммы и количество байт в этом массиве, используемое для данных датаграммы. В случае если создаваемый объект должен использоваться для получения датаграммы, то это все, что нужно передать конструктору. При этом, массив должен быть пустым. Когда создается объект для получения датаграммы из сети, он сохраняет данные пакета в массиве -буфере объекта `DatagramPacket` вплоть до указанной вами длины.

Второй набор конструкторов `DatagramPacket` используется для создания датаграмм, которые будут отправлены по сети. Как и в случае первого набора, этим конструкторам также нужен массив для буфера и длина, но в этом случае нужно также указать `InetAddress` и порт, на который следует отправить пакет. В этом случае конструктору передается байтовый массив, содержащий предназначенные для отправки данные, а также адрес и порт назначения, куда должен быть отправлен пакет. Затем объект

DatagramSocket считывает адрес и порт назначения из переданного на отправку пакета.

Конструкторы для получения датаграмм

Для создания объектов DatagramPacket, предназначенных для получения данных из сети, существует два конструктора:

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

Когда сокет получил датаграмму, он сохраняет часть данных датаграммы в буфере, начиная с `buffer[0]` и продолжая, пока пакет не будет полностью заполнен или пока `length` байт не будут записаны в буфер. Если используется второй конструктор, сохранение данных датаграммы начинается с `buffer[offset]`. В остальном эти два конструктора одинаковы. Параметр `length` должен быть меньше или равен разности `buffer.length - offset`. Если осуществляется попытка создать объект `DatagramPacket` с таким значением параметра `length`, при котором произойдет переполнение буфера, конструктор выдает исключение `IllegalArgumentException`. Это исключение относится к `RuntimeException`, поэтому его не обязательно обрабатывать. При этом разрешается создавать `DatagramPacket` с таким значением параметра `length` который меньше, чем `buffer.length - offset`. В этом случае при получении датаграммы будет заполнено не более, чем первые `length` байт буфера. Для примера, приведет фрагмент кода, который создает новый `DatagramPacket` для получения датаграммы длиной до 8 192 байт:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

Конструктор не ограничивает размер буфера и может создать объект `DatagramPacket` инкапсулирующий мегабайты данных. Однако, базовое нативное программное обеспечение сети и большинство нативных реализаций *UDP* не поддерживают более 8192 байт данных на каждую датаграмму. Следовательно, не рекомендуется создавать объекты `DatagramPacket` с более чем 8192 байтами данных.

Конструкторы для отправки датаграмм

Существуют четыре конструктора, которые создают новые объекты DatagramPacket, предназначенные для отправки данных по сети на другой хост:

```
public DatagramPacket(byte[] data, int length,  
                      InetAddress destination, int port)  
public DatagramPacket(byte[] data, int offset, int length,  
                      InetAddress destination, int port)  
public DatagramPacket(byte[] data, int length,  
                      SocketAddress destination, int port)  
public DatagramPacket(byte[] data, int offset, int length,  
                      SocketAddress destination, int port)
```

Пакет заполняется length байтами массива данных, начиная со смещения offset или с 0, если параметр offset не используется. Если будет попытка создать DatagramPacket с параметром length, превышающим data.length, конструктор выдает исключение IllegalArgumentException. Разрешается создавать объект DatagramPacket с такими значениями параметров length и offset, которые оставят дополнительное неиспользуемое пространство в конце массива данных. В этом случае по сети будут отправлены только байты данных. Объекты InetAddress или SocketAddress определяют хост, на который нужно доставить пакет; целочисленный параметр port определяет номер порта назначения на этом хосте.

Перед созданием объекта DatagramPacket данные, которые нужно отправить преобразовываются в байтовый массив и передаются как параметр data конструктору. Вообще-то данные не копируются во внутренний буфер и изменение массива data после построения датаграммы и до ее отправки изменяет данные в датаграмме. Иногда это бывает удобно. Например, можно сделать датаграмму на основании массива данных, который постоянно меняется, и регулярно, в заданные моменты времени отправлять датаграмму (с самыми последними данными). Однако, гораздо чаще бывает необходимо удостовериться, что данные не изменяются, если вы этого не хотите. Это особенно важно, если программа многопоточная, и разные потоки могут записывать информацию в буфер данных. В этом случае нужно синхронизировать переменную data или сделать копию переменной data во временный буфер, прежде чем создавать DatagramPacket.

Приведем, в качестве примера, фрагмент кода, который создает новый пакет `DatagramPacket`, заполненный данными «This is a test» в кодировке *ASCII*. Пакет направляется на порт 7 (*эхо-порт*) хоста *www.ibiblio.org*:

```
String s = "This is a test";
byte[] data = s.getBytes("ASCII");
try {
    InetAddress ia = InetAddress.getByName("www.ibiblio.org");
    int port = 7;
    DatagramPacket dp =
        new DatagramPacket(data, data.length, ia, port);
    // send the packet...
} catch (IOException ex) {
}
```

В большинстве случаев самой сложной частью создания нового `DatagramPacket` является перевод данных в байтовый массив. В приведенном примере используется метод `java.lang.String.getBytes()` (*ASCII* строка в массив байт). Кроме того, для подготовки данных для включения в датаграммы может быть очень полезен класс `java.io.ByteArrayOutputStream`.

Методы доступа: геттеры

Класс `DatagramPacket` содержит шесть методов, которые извлекают различные части датаграммы: фактические данные датаграммы, а также некоторые поля из ее заголовка. Эти методы в основном используются для датаграмм, полученных из сети.

public InetAddress getAddress()

Метод `getAddress()` возвращает объект `InetAddress`, который представляет адрес удаленного хоста. Если датаграмма была получена из *Интернет*, адрес, который будет возвращен методом, является адресом компьютера, который ее отправил (адрес источника). Если же датаграмма была создана локально для отправки на удаленный компьютер, этот метод возвращает адрес хоста, которому адресована датаграмма (адрес назначения). Этот метод чаще всего используется для определения адреса хоста, который отправил *UDP*-датаграмму, чтобы получатель мог ответить отправителю.

public int getPort()

Метод `getPort()` возвращает целое число, определяющее удаленный порт. Если датаграмма была получена из *Интернет*, то это порт хоста, который отправил пакет. Если датаграмма была создана локально для отправки на удаленный хост, это порт, к которому адресован пакет на удаленном компьютере.

public SocketAddress getSocketAddress()

Метод `getSocketAddress()` возвращает объект `SocketAddress`, содержащий *IP*-адрес и порт удаленного хоста. Как и в случае с методом `getInetAddress()`, если датаграмма была получена из *Интернет*, возвращаемым адресом является адрес компьютера, который ее отправил (адрес источника). Если же датаграмма была создана локально для отправки на удаленный компьютер, этот метод возвращает адрес хоста, которому адресована датаграмма (адрес назначения). Обычно этот метод используется для определения адреса и порта хоста, который отправил *UDP*-датаграмму. Обычно результат вызова этого метода не отличается от вызовов `getAddress()` и `getPort()`, но только это один вызов метода. При использовании неблокирующего ввода / вывода, класс `DatagramChannel` принимает `SocketAddress`, а не `InetAddress` и `port`.

public byte[] getData()

Метод `getData()` возвращает байтовый массив `byte[]`, содержащий данные из датаграммы. Обычно, после получения такого массива, требуется преобразовать байты в какую-либо другую форму данных. Один из возможных способов сделать это - преобразовать этот байтовый массив в строку с помощью конструктора строки:

public String(byte[] buffer, String encoding)

Первый аргумент, байтовый массив `buffer`, содержит данные из датаграммы. Второй аргумент содержит имя кодировки, используемой для этой строки, например `ASCII` или `ISO-8859-1`. Таким образом, для переменной `DatagramPacket dp`, полученной из сети, можно преобразовать полученные данные в строку:

```
String s = new String(dp.getData( ), "ASCII");
```

Если датаграмма содержит не текст, а данные какого-либо другого вида, то для преобразования полученных данных в данные *Java* нужно выполнить больше действий. Один из подходов заключается в преобразовании байтового

массива, возвращаемого методом `getData()`, в объект `ByteArrayInputStream` с помощью конструктора:

```
public ByteArrayInputStream(byte[] buffer, int offset, int length)
```

Здесь `buffer` - это байтовый массив, который будет использоваться в качестве `InputStream`. При этом важно указать ту часть буфера, которую нужно использовать в качестве `InputStream`; это можно сделать с помощью аргументов `offset` и `length`. В ходе преобразовании данных, перенесенных датаграммой в объекты `InputStream` смещение либо равно 0, либо задается методом `getOffset()` объекта `DatagramPacket`, а длина задается методом `getLength()` объекта `DatagramPacket`. Например:

```
InputStream in = new ByteArrayInputStream(packet.getData(),  
                                         packet.getOffset(), packet.getLength());
```

При создании объекта `ByteArrayInputStream` из данных датаграммы рекомендуется указывать смещение и длину. При этом не рекомендуется использовать конструктор `ByteArrayInputStream()`, который принимает в качестве аргумента только массив. Массив, возвращаемый `packet.getData()`, может содержать дополнительное пустое место, которое не было заполнено данными из сети. Это пространство будет содержать любые случайные значения, которые были у этих компонентов массива при создании `DatagramPacket`.

После создания объект `ByteArrayInputStream` может быть обернут `DataInputStream`:

```
DataInputStream din = new DataInputStream(in);
```

После этого данные можно прочесть, используя обычные методы `DataInputStream`, например `readInt()`, `readLong()`, `readChar()` и другие. Эта процедура неявно предполагает, что отправитель датаграммы использует такой же формат данных, что и *Java*; обычно, это бывает тогда, когда отправитель написан на *Java*. Однако, большинство современных компьютеров используют тот же формат с плавающей запятой, что и *Java*.

```
public int getLength( )
```

Метод `getLength()` возвращает количество байт данных, реально содержащихся в датаграмме. Это не обязательно совпадает с длиной массива,

возвращаемого `getData()`, то есть `getData().length`. Целочисленное значение, возвращаемое `getLength()`, может быть меньше длины массива, возвращаемого `getData()`.

public int getOffset()

Этот метод возвращает индекс массива, возвращаемого методом `getData()`, где начинаются данные из датаграммы.

Рассмотрим простой искусственный пример работы с объектом `DatagramPacket`. Программа создает объект `DatagramPacket`, и все о нем знает, но эти действия будут использоваться для обработки `DatagramPacket`, который получен из сети.

```
import java.net.*;
public class DatagramExample {
    public static void main(String[] args) {
        String s = "This is a test.";
        byte[] data = s.getBytes( );
        try {
            InetAddress ia = InetAddress.getByName("www.ibiblio.org");
            int port = 7;
            DatagramPacket dp
                = new DatagramPacket(data, data.length, ia, port);
            System.out.println("This packet is addressed to "
                + dp.getAddress() + " on port " + dp.getPort( ));
            System.out.println("There are " + dp.getLength( )
                + " bytes of data in the packet");
            System.out.println(
                new String(dp.getData(), dp.getOffset(), dp.getLength( )));
        }
        catch (UnknownHostException e) {
            System.err.println(e);
        }
    }
}
```

Результат работы примера:

```
This packet is addressed to www.ibiblio.org/152.19.134.40 on port 7
There are 15 bytes of data in the packet
This is a test.
```

Методы доступа: сеттеры

В большинстве случаев шести конструкторов достаточно для работы с датаграммами. Однако *Java* предоставляет методы для изменения, как данных, так и удаленного адреса и удаленного порта уже после создания датаграммы. Эти методы могут использоваться в ситуации, когда время создания новых объектов `DatagramPacket` и время сбора мусора сильно снижает производительность приложения. В некоторых ситуациях повторное

использование объектов может быть более удобным, чем создание новых, особенно тогда, когда генерируется большое количество датаграмм.

public void setData(byte[] data)

Метод setData() изменяет данные, переносимые датаграммой *UDP*. Этот метод можно использовать в ситуации, когда на удаленный хост отправляется большой файл (больше, чем может удобно поместиться в одну датаграмму): можете повторно отправлять один и тот же объект DatagramPacket, каждый раз меняя его данные.

public void setData(byte[] data, int offset, int length)

Этот перегруженный вариант метода setData() представляющий альтернативный подход к отправке большого количества данных. Вместо того, чтобы отправлять большое количество новых массивов, можно поместить все данные в один массив и отправлять их по частям за раз. Например, этот цикл отправляет большой массив из 512-байтовых блоков:

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;
while (bytesSent < bigarray.length) {
    socket.send(dp);
    bytesSent += dp.getLength( );
    int bytesToSend = bigarray.length - bytesSent;
    int size = (bytesToSend > 512) ? 512 : bytesToSend;
    dp.setData(bigarray, bytesSent, 512);
}
```

Однако, это тоже не идеальная стратегия: нужна гарантия того, что все отправленные данные действительно поступят на удаленный хост, либо, наоборот, понимать, что не все пакеты когут добраться и игнорировать последствия их отсутствия. При таком подходе относительно сложно как-то назначить порядковые номера или другие теги к отдельным пакетам.

public void setAddress(InetAddress remote)

Метод setAddress() меняет адрес, на который отправляются датаграммы. Этот метод поможет отправить одну и ту же датаграмму разным получателям. Например:

```
String s = "Really Important Message";
byte[] data = s.getBytes("ASCII");
DatagramPacket dp = new DatagramPacket(data, data.length);
dp.setPort(2000);
int network = "128.238.5.";
for (int host = 1; host < 255; host++) {
    try {
        InetAddress remote = InetAddress.getByName(network + host);
        dp.setAddress(remote);
        socket.send(dp);
    } catch (IOException ex) {
        // slip it; continue with the next host
    }
}
```

Но только от применения зависит, удобен ли этот поход. Если нужно отправить сообщение на все станции в заданном сегменте сети, то, скорее всего, будет лучше использовать локальный широковещательный адрес. Локальный широковещательный адрес определяется путем установки всех битов *IP*-адреса после идентификаторов сети и подсети равными 1. Например, сетевой если адрес 128.238.0.0, то его широковещательный адрес будет 128.238.255.255. Отправка датаграммы по адресу 128.238.255.255 копирует ее на каждый хост в этой сети (хотя иногда маршрутизаторы и брандмауэры могут блокировать ее).

Для этого, бывает лучше использовать многоадресную рассылку, о которой мы поговорим на следующем занятии.

public void setPort(int port)

Метод `setPort()` изменяет порт, на который адресована датаграмма. Этот метод можно использовать в приложении сканера портов, которое пытается найти открытые порты, на которых работают определенные службы, работающие на основе *UDP*. Другой возможностью использования может быть сервер конференц-связи, на котором клиенты, которым требуется получать одинаковую информацию, работают на разных хостах и на разных портах. В этом случае метод `setPort()` может использоваться совместно с методом `setAddress()`.

public void setAddress(SocketAddress remote)

Метод `setSocketAddress()` сразу изменяет адрес и порт, на который отправляется пакет датаграммы. Рассмотрим пример, в котором после получения датаграммы и формируется ответ на тот же адрес:

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);
SocketAddress address = input.getSocketAddress( );
DatagramPacket output = new
    DatagramPacket("Hello there".getBytes("ASCII"), 11);
output.setAddress(address);
socket.send(output);
```

В ранних версиях *Java* нужно было вместо `SocketAddress` использовать порты и `InetAddress`:

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);
InetAddress address = input.getAddress( );
int port = input.getPort( );
DatagramPacket output = new
    DatagramPacket("Hello there".getBytes("ASCII"), 11);
output.setAddress(address);
output.setPort(port);
socket.send(output);
```

public void setLength(int length)

Метод `setLength()` изменяет количество байт данных во внутреннем буфере датаграммы, которые считаются частью данных, а не просто незаполненным пространством. Этот метод обычно используется при получении датаграмм. Когда получен пакет, длина данных устанавливается равной длине входящих данных. Поэтому, если нужно получить другой пакет в тот же объект `DatagramPacket`, длина будет ограничена количеством байт в первом пакете. Этот метод позволяет вам переустановить длину буфера, чтобы последующие датаграммы не были усечены.

Класс DatagramSocket

Для отправки и получения объектов `DatagramPacket`, нужно создать датаграммный сокет. В *Java* датаграммный сокет представлен классом `DatagramSocket`:

```
public class DatagramSocket extends Object
```

Все датаграммные сокеты привязаны (*bound*) к локальному порту, на котором они ожидают входящие данные и отправляют исходящие. Если создается клиент, то совершенно не важно, с каким локальным портом он работает, поэтому в этом случае можно использовать конструктор, который позволяет системе автоматически назначать какой-либо неиспользуемый порт (*анонимный порт*). Этот, автоматически выбранный номер порта, помещается в любые исходящие датаграммы и будет использоваться сервером для адресации любых ответных сообщений. Если создается сервер, то клиенты должны знать, на каком порту сервер ожидает входящие датаграммы; следовательно, когда сервер создает объект `DatagramSocket`, нужно указать локальный порт, который будет прослушиваться. В остальном сокеты, используемые и клиентами, и серверами одинаковы: основное отличие только тем, используют ли они анонимный (автоматически назначенный системой) или общеизвестный порт. В отличие от работы *TCP* протоколу, в случае использования протокола *UDP* нет различий между клиентскими и серверными сокетами.

Сейчас рассмотрим основные возможности, предоставляемые классом; о всех возможностях можно узнать из документации на сайте *Oracle*:

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

Конструкторы

Так же, как и конструкторы `DatagramPacket`, конструкторы `DatagramSocket` используются в разных ситуациях. Один из конструкторов создает датаграммный сокет на анонимном локальном порту. Другой конструктор открывает датаграммный сокет на указанном локальном порту, который прослушивает все локальные сетевые интерфейсы. Еще один конструктор создает датаграммный сокет на указанном локальном порту определенного сетевого интерфейса. В следующих версиях *Java* к ним был добавлен конструктор, который позволяет указывать сетевой интерфейс и порт с помощью объекта `SocketAddress`. В еще более свежей версии *Java* добавлен защищенный конструктор, который позволяет изменять класс реализации. Все пять конструкторов имеют дело только с адресом и портом локального хоста. Адрес и порт удаленного хоста хранятся в `DatagramPacket`, а не в

DatagramSocket. Один объект DatagramSocket может отправлять датаграммы на и получать датаграммы от нескольких удаленных хостов и портов.

public DatagramSocket() throws SocketException

Этот конструктор создает сокет, связанный с анонимным локальным портом. Например:

```
try {
    DatagramSocket client = new DatagramSocket( );
    // send packets...
} catch (SocketException ex) {
    System.err.println(ex);
}
```

Это конструктор предназначен для использования в клиенте, который инициирует взаимодействие с сервером. В этом случае не важно, к какому порту подключен сокет, потому что сервер отправит свой ответ на порт, с которого пришла датаграмма. Здесь важно, что система сама найдет подходящий порт, а если нужно узнать номер локального порта, вы можете узнать об этом с помощью метода `getLocalPort()`.

Один и тот же сокет может не только отправлять, но и принимать датаграммы. Если сокет не может быть создан, то выбрасывается исключение `SocketException`. Но, это исключительно редкая ситуация; трудно представить себе ситуации, в которых сокет не может быть открыт, поскольку система сама, автоматически выбирает локальный порт.

public DatagramSocket(int port) throws SocketException

Этот конструктор создает сокет, который прослушивает входящие датаграммы на указанном в качестве аргумента порту `port`. Обычно, этот конструктор используется для создания сервера, который прослушивает заданный порт; если серверы прослушивают анонимные порты, у клиентов не будет возможности с ними связаться. Если сокет не может быть создан, то выбрасывается исключение `SocketException`. Обычно существует две причины сбоя работы конструктора: указанный порт уже занят, или была предпринята попытка подключения к порту ниже *1024*, при недостатке прав доступа.

Следует отметить, что *TCP*-порты и *UDP*-порты друг с другом никак не связаны. Два разных сервера или клиента могут использовать один и тот же

номер порта, если один использует протокол *UDP*, а другой использует протокол *TCP*. Рассмотрим простой пример - сканер портов, который ищет *UDP*-порты, используемые на локальном хосте. Сканер принимает решение, что порт используется, если конструктор `DatagramSocket` выбрасывает исключение.

```
import java.net.*;
public class UDPPortScanner {
    public static void main(String[] args) {
        for (int port = 1024; port <= 65535; port++) {
            try {
// the next line will fail and drop into the catch block if
// there is already a server running on port i
                DatagramSocket server = new DatagramSocket(port);
                server.close( );
            } catch (SocketException ex) {
                System.out.println("There is a server on port " + port + ".");
            } // end try
        } // end for
    }
}
```

Скорость, с которой работает `UDPPortScanner`, сильно зависит от скорости компьютера и реализации *UDP*. Попробуйте работу программы на Вашем компьютере.

Создать сканер портов для удаленного хоста с помощью протокола *UDP* гораздо сложнее, чем с помощью протокола *TCP*. Дело в том, что *UDP* протокол не дает гарантий доставки сообщений.

**public DatagramSocket(int port, InetAddress interface)
throws SocketException**

Этот конструктор чаще всего используется на хостах с более чем одним сетевым интерфейсом. Он создает сокет `port`, который прослушивает указанный порт и сетевой интерфейс `interface`. Если сокет не может быть создан, то выбрасывается исключение `SocketException`. Существует три распространенные причины сбоя работы этого конструктора: указанный порт уже занят, вы пытаетесь подключиться к порту ниже 1024, и вы не являетесь пользователем `root` в системе `Unix`, или была предпринята попытка

подключения к порту ниже *1024*, при недостатке прав доступа, или адрес `InetAddress` не является адресом какого-либо системного интерфейса.

public DatagramSocket(SocketAddress interface) throws SocketException

Этот конструктор похож на предыдущий, за исключением того, что адрес и порт сетевого интерфейса берутся из объекта `SocketAddress`. Например, этот фрагмент кода создает сокет, который прослушивает только локальный адрес обратной связи (*local loopback address*):

```
SocketAddress address = new InetSocketAddress("127.0.0.1", 9999);
DatagramSocket socket = new DatagramSocket(address);
```

protected DatagramSocket(DatagramSocketImpl impl) throws SocketException

Этот конструктор позволяет предоставить собственную реализацию протокола *UDP*, а не пользоваться реализацией по умолчанию. В отличие от сокетов, созданных другими четырьмя конструкторами, этот сокет изначально не привязан к порту. Перед использованием сокета необходимо связать его с `SocketAddress` с помощью метода `bind()`.

```
public void bind(SocketAddress addr) throws SocketException
```

Этому методу можно передать `null`, привязав сокет к какому-либо доступному адресу и порту.

Отправка и получение датаграм

Основная задача объектов класса `DatagramSocket` – отправлять и получать *UDP* датаграммы. Один сокет может и отправлять, и получать их.

public void send(DatagramPacket dp) throws IOException

После создания `DatagramPacket` и создания `DatagramSocket` можно отправить пакет, передав его в метод `send()` сокета. Например, если `theSocket` является объектом `DatagramSocket`, а `theOutput` является объектом `DatagramPacket`, отправьте `theOutput` с помощью `theSocket`, как показано ниже:

```
theSocket.send(theOutput);
```

Если при отправке данных возникнет проблема, будет выброшено исключение `IOException`. Однако при использовании `DatagramSocket`, это происходит не часто, поскольку *UDP* является ненадежным протоколом, исключение не может быть выброшено потому, что пакет не прибыл в пункт назначения. Исключение `IOException` можно получить, если имела место попытка отправить более крупную датаграмму, чем поддерживает нативное сетевое программное обеспечение хоста. Этот метод также может вызвать исключение `SecurityException`, если `SecurityManager` не позволит вам связаться с хостом, которому адресован пакет. Это в первую очередь проблема для апплетов и другого удаленно загруженного кода.

Рассмотрим простой пример на основе *UDP*. В примере пользователь вводит строки, отправляет на сервер, который просто отбрасывает все данные.

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

public class UDPDiscardClient {
    public final static int DEFAULT_PORT = 9;
    public static void main(String[] args) {
        String hostname;
        int port = DEFAULT_PORT;
        if (args.length > 0) {
            hostname = args[0];
            try {
                port = Integer.parseInt(args[1]);
            }
            catch (Exception ex) {
                // use default port
            }
        }
        else {
            hostname = "localhost";
        }
        try {
            InetAddress server = InetAddress.getByName(hostname);
            BufferedReader userInput
                = new BufferedReader(new InputStreamReader(System.in));
            DatagramSocket theSocket = new DatagramSocket( );
            while (true) {
                String theLine = userInput.readLine( );
                if (theLine.equals(".")) break;
                byte[] data = theLine.getBytes( "UTF-8");
                DatagramPacket theOutput
                    = new DatagramPacket(data, data.length, server, port);
                theSocket.send(theOutput);
            } // end while
        } // end try
        catch (UnknownHostException uhex) {
            System.err.println(uhex);
        }
        catch (SocketException sex) {
            System.err.println(sex);
        }
        catch (IOException ioex) {
            System.err.println(ioex);
        }
    }
}
```

```

    } // end main
}

```

Класс `UDPDiscardClient` имеет достаточно простую структуру. В классе содержится одно статическое поле, `DEFAULT_PORT`, для которого задан стандартный порт для протокола сброса (*порт 9*), и единственный метод `main()`. Метод `main()` считывает имя хоста из командной строки и преобразует это имя хоста в объект `InetAddress`, соответствующий серверу. Создается объект `BufferedReader`, связанный с `System.in` для чтения ввода пользователя с клавиатуры. Затем создается объект `DatagramSocket` с именем `theSocket`. После создания сокета программа входит в бесконечный цикл `while`, который построчно читает вводимые пользователем данные, используя `readLine()`. Поскольку протокол сброса имеет дело только с необработанными байтами, мы можем игнорировать проблемы кодировки символов.

В цикле `while` каждая строка преобразуется в байтовый массив с помощью метода `getBytes()`, а байты помещаются в новый `DatagramPacket`, `theOutput`. Затем, объект `theOutput` отправляется через сокет, и цикл продолжается. Если в какой-то момент пользователь вводит точку в строке ввода, то программа завершается. Конструктор `DatagramSocket` может выбрасывать исключение `SocketException`, поэтому его необходимо обработать. Так как это клиент сброса (*discard client*), то о данных, возвращаемых с сервера, беспокоиться не нужно.

public void receive(DatagramPacket dp) throws IOException

С помощью этого метода можно получить одну *UDP* датаграмму из сети и сохранить ее в существующем объекте `DatagramPacket dp`. Так же, как и метод `accept()` класса `ServerSocket`, этот метод блокирует вызывающий поток исполнения до прибытия датаграммы. Если программа делает что-то еще, кроме ожидания датаграмм, то рекомендуется вызвать `receive()` в отдельном потоке исполнения.

Буфер датаграммы должен быть достаточно большим для хранения полученных данных. Если объема буфера недостаточно, то метод `receive()` помещает в буфер столько данных, сколько буфер может получить; при этом остальная информация будет потеряна. Мы уже обсуждали, что максимальный размер данных *UDP* датаграммы составляет *65 507* байт. (Это максимальный размер *65 536* байт для *IP* датаграммы за исключением *20*-байтового заголовка *IP* и *8*-байтового заголовка *UDP*). Некоторые протоколы приложений, использующие *UDP*, дополнительно ограничивают максимальное количество

байтов в пакете; например, *NFS* использует максимальный размер пакета 8192 байта.

Если при получении данных возникла проблема, то будет выброшено исключение `IOException`. Но на практике эта ситуация возникает редко. В отличие от метода `send()`, этот метод не генерирует исключение `SecurityException`, если апплет получает датаграмму не от хоста апплета. Он просто будет отбрасывать все такие пакеты.

Рассмотрим еще один простой пример сервер сброса с использованием протокола *UDP* (*UDP discard server*). Для контроля будем записывать полученные данные из каждой датаграммы в `System.out`.

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

public class UDPDiscardClient {
    public final static int DEFAULT_PORT = 9;
    public static void main(String[] args) {
        String hostname;
        int port = DEFAULT_PORT;
        if (args.length > 0) {
            hostname = args[0];
            try {
                port = Integer.parseInt(args[1]);
            }
            catch (Exception ex) {
                // use default port
            }
        }
        else {
            hostname = "localhost";
        }
        try {
            InetAddress server = InetAddress.getByName(hostname);
            BufferedReader userInput
                = new BufferedReader(new InputStreamReader(System.in));
            DatagramSocket theSocket = new DatagramSocket( );
            while (true) {
                String theLine = userInput.readLine( );
                if (theLine.equals(".")) break;
                byte[] data = theLine.getBytes( "UTF-8" );
                DatagramPacket theOutput
                    = new DatagramPacket(data, data.length, server, port);
                theSocket.send(theOutput);
            } // end while
        } // end try
        catch (UnknownHostException uhex) {
            System.err.println(uhex);
        }
        catch (SocketException sex) {
            System.err.println(sex);
        }
        catch (IOException ioex) {
            System.err.println(ioex);
        }
    } // end main
}
```

Это простой класс с единственным методом `main()`. Метод анализирует командную строку и считывает оттуда номер порта, который будет прослушивать сервер. Если в командной строке порт не указан, то будет прослушиваться *порт 9*. Затем создается объект `DatagramSocket` на этом порту и создает `DatagramPacket` с буфером размером *65 507* байт - достаточно большим, чтобы принять любой возможный пакет. Затем сервер входит в бесконечный цикл, который принимает пакеты и выводит в консоль и их содержимое, и информацию о хосте-получателе. Обычно сервер сброса этого не делает. При получении каждой датаграммы длина пакета устанавливается равной длине данных в этой датаграмме. Поэтому, последним шагом цикла длина пакета восстанавливается до максимально возможного значения. Если этого не делать, то входящие пакеты будут ограничены минимальным размером всех предыдущих пакетов. Вы можете запустить клиент сброса на одном компьютере и подключиться к серверу сброса на другом компьютере, чтобы убедиться, что сеть работает.

Запустите программу на Вашем компьютере.

Результат работы программы на моем компьютере приведен ниже.

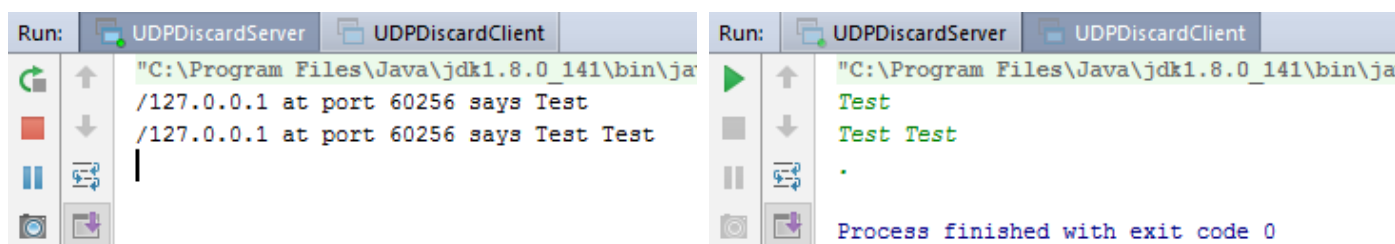


Рисунок слева – снимок экрана работы сервера, а справа – клиента.

public void close()

Вызов метода `close()` объекта `DatagramSocket` освобождает порт, занятый этим сокетом. Например:

```
try {
    DatagramSocket server = new DatagramSocket( );
    server.close( );
} catch (SocketException ex) {
    System.err.println(ex);
}
```

Закрытие сокетов `DatagramSocket`, когда с ним закончена работа, является хорошей и рекомендуемой практикой. Особенно это важно, если

программа продолжает работать в течение длительного периода времени. Например, метод `close()` был необходим в примере `UDPPortScanner`: если бы эта программа не закрывала открытые сокеты, то работала бы гораздо дольше. Если программа заканчивается, как только закончилась работа с `DatagramSocket`, то явно закрывать сокет не обязательно; сокет автоматически закрывается при сборке мусора. Однако виртуальная машина *Java* не будет запускать сборщик мусора только потому, что закончились порты или сокеты; сборщик будет запущен только в случае нехватки памяти, и очень повезет, если это произойдет одновременно.

public int getLocalPort()

Метод `getLocalPort()` объекта `DatagramSocket` возвращает целое число, представляющее локальный порт, с которым связан сокет. Обычно, этот метод используется, если `DatagramSocket` создан с анонимным портом и нужно узнать, с каким конкретно портом связан сокет. Например:

```
try {
    DatagramSocket ds = new DatagramSocket( );
    System.out.println("The socket is using port " + ds.getLocalPort( ));
} catch (SocketException ex) {
    ex.printStackTrace( );
}
```

public InetAddress getLocalAddress()

Метод `getLocalAddress()` объекта `DatagramSocket` возвращает объект `InetAddress`, представляющий локальный адрес, с которым связан сокет. Эта возможность редко требуется на практике. Обычно, это либо известно, либо это не важно.

public SocketAddress getLocalSocketAddress()

Метод `getLocalSocketAddress()` возвращает объект `SocketAddress`, который инкапсулирует локальный интерфейс и порт, к которому привязан сокет. Этот метод используется крайне редко и, возможно, в основном существует для симметрии с `setLocalSocketAddress()`.

Управление соединениями

В отличие от *TCP* сокетов, *UDP* сокеты датаграмм не работают с соединениями. По умолчанию они могут принимать датаграммы от любых хостов. Часто это не совсем то, что нужно. В ранних версиях *Java* программы

должны были вручную проверять адреса и порты хостов-источников, отправляющих им данные, чтобы убедиться, что они те, кто нужно. Однако в последующих версиях *Java* были добавлены четыре метода, которые позволяют выбрать, с какого хоста можете отправлять датаграммы и получать датаграммы, отклоняя при этом все остальные пакеты.

public void connect(InetAddress host, int port)

Метод `connect()` на самом деле не устанавливает соединение в смысле протокола *TCP*. Он просто указывает, что данный `DatagramSocket` будет отправлять пакеты и получать пакеты только от указанного удаленного хоста на указанном удаленном порту. Попытки отправить пакеты на другой хост или порт приведут к исключению `IllegalArgumentException`. Пакеты, полученные от другого хоста или другого порта, будут отброшены без выброса исключения или какого-либо другого уведомления.

При вызове метода `connect()` выполняется проверка безопасности. Если виртуальной машине не разрешено отправлять данные на этот хост и порт, то выбрасывается исключение `SecurityException`. Но после установления соединения вызов методов `send()` и `receive()` на этом `DatagramSocket` не будут выполнять проверку безопасности, которую они обычно выполняют.

public void disconnect()

Метод `disconnect()` разрывает «установленное соединение» для соответствующего `DatagramSocket`, чтобы он мог снова отправлять и получать пакеты с любого хоста и порта.

public int getPort()

Если `DatagramSocket` соединен с удаленным хостом, то метод `getPort()` возвращает номер этого удаленного порта; в противном случае возвращается -1.

public InetAddress getInetAddress()

Если `DatagramSocket` соединен с удаленным хостом, то метод `getInetAddress()` возвращает адрес этого удаленного хоста; в противном случае возвращается `null`.

public SocketAddress getRemoteSocketAddress()

Если `DatagramSocket` соединен с удаленным хостом, то метод `getRemoteSocketAddress()` возвращает адрес этого удаленного хоста; в В противном случае возвращается `null`.

Опции сокета

В современной *Java* данный вид сокет поддерживает небольшое количество параметров (напр., `SO_TIMEOUT`, `SO_SNDBUF`, `SO_RCVBUF`, `SO_REUSEADDR`, `SO_BROADCAST`). Мы пока познакомимся с параметром `SO_TIMEOUT`.

Параметр `SO_TIMEOUT`

Параметр `SO_TIMEOUT` определяет интервал времени (в миллисекундах), в течение которого метод `receive()` ожидает входящую датаграмму, прежде чем выбросить исключение типа `InterruptedIOException` (подкласс `IOException`). Значение этого параметра сокета должно быть неотрицательным числом. Если `SO_TIMEOUT` равен 0, то метод `receive()` никогда не будет прерываться исключением. Значение этого параметра можно изменить с помощью метода `setSoTimeout()`, а проверить текущее значение – с помощью метода `getSoTimeout()`:

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

По умолчанию предполагается, что параметр `SO_TIMEOUT` имеет значение 0, и ожидание поступление датаграммы может продолжаться неограниченное время. Однако, есть случаи, когда требуется установить ненулевое значение этому параметру: есть требования, чтобы ответы приходили в течение фиксированного периода времени; кроме того, иногда нужно понять, а работает ли тот хост, куда отправляются пакеты (если нет ответа в течение заданного периода времени, то, скорее всего, хост не работает).

Метод `setSoTimeout()` устанавливает поле `SO_TIMEOUT` для датаграммного сокета. По истечении времени ожидания выбрасывается исключение. Тип исключения зависит от версии *Java*: в ранних версиях выбрасывалось исключение `InterruptedIOException`, а в более поздних версиях вместо этого выбрасывается `SocketTimeoutException`, подкласс `InterruptedIOException`. Этот параметр не обходимо устанавливать перед вызовом метода `receive()`. Параметр `SO_TIMEOUT` нельзя изменить во время того, как `receive()` ожидает датаграммы. Если аргумент метода

setSoTimeout() меньше нуля, то выбрасывается исключение SocketException. Например:

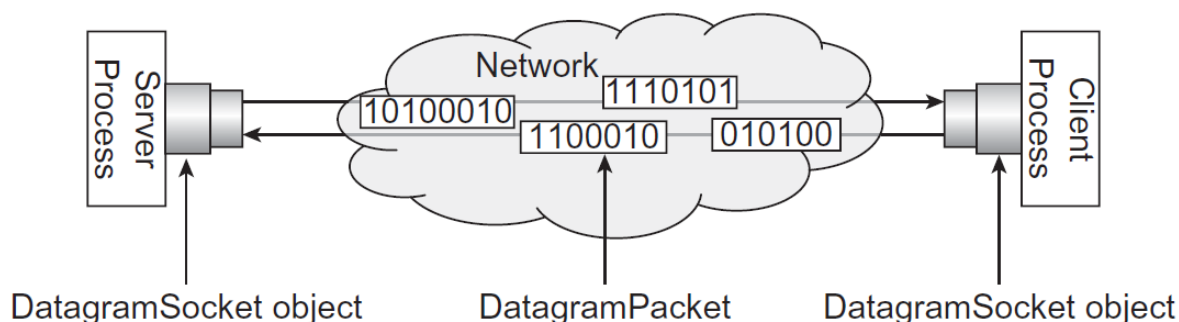
```
try {
    buffer = new byte[2056];
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
    DatagramSocket ds = new DatagramSocket(2048);
    ds.setSoTimeout(30000); // block for no more than 30 seconds
    try {
        ds.receive(dp);
        // process the packet...
    } catch (InterruptedException ex) {
        ds.close( );
        System.err.println("No connection within 30 seconds");
    } catch (SocketException ex) {
        System.err.println(ex);
    } catch (IOException ex) {
        System.err.println("Unexpected IOException: " + ex);
    }
}
```

Метод getSoTimeout() возвращает текущее значение поля SO_TIMEOUT у заданного объекта DatagramSocket. Например:

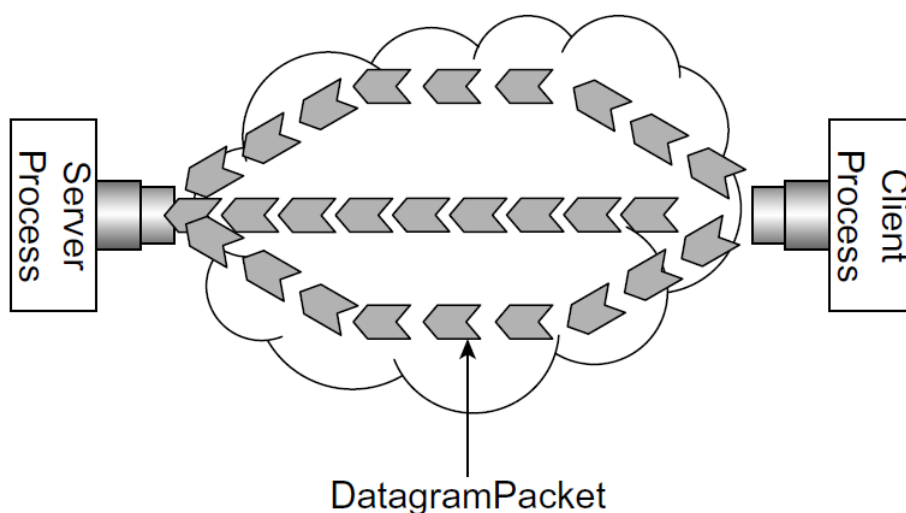
```
public void printSoTimeout(DatagramSocket ds) {
    int timeout = ds.getSoTimeout( );
    if (timeout > 0) {
        System.out.println(ds + " will time out after " + timeout +
            "milliseconds.");
    } else if (timeout == 0) {
        System.out.println(ds + " will never time out.");
    } else {
        System.out.println("Something is seriously wrong with " + ds);
    }
}
```

Основные этапы создания клиент / серверного приложения на UDP сокетах

Как уже было сказано ранее, с помощью *UDP* сокетов можно передавать данные через сеть немного по другому, чем спомощью сокетов *TCP*. Вместо установления соединения создаются два объекта *DatagramSocket*, а данные отправляются и принимаются с использованием контейнеров, называемых датаграммами.



Программа-получатель (*receiver*) сначала создает датаграмный сокет и ожидает входящие датаграммы. В свою очередь, программа-отправитель (*sender*) создает датаграммы, указывает *IP*-адрес и номер порта получателя и отправляет его в сеть. В результате датаграммный пакет по какому-то либо пути, возможно, через промежуточные хосты, направляется получателю. Причем, разные датаграммы на один и тот же хост могут направляться по разным путям.



При организации взаимодействия следует иметь в виду, что датаграмма может быть потеряна или может иметь место какая-либо ошибка в данных пакета. Как мы говорили, связь по протоколу *UDP* не гарантирует, что датаграмма будет доставлена получателю, и то, что в сети будет реально существовать адресат сообщения. Кроме того, датаграммы могут разными

путями достигать адресата, и нет гарантии, что они придут в том порядке, в котором были отправлены.

Рассмотрим основные шаги, которые нужно предпринять для создания сетевых приложений на основе *UDP* сокетов:

Получатель (Receiver)	Отправитель (Sender)
1. Создается объект <code>DatagramSocket</code> с явным указанием номера порта	2. Создается объект <code>DatagramSocket</code> и привязывается (<code>bind</code>) к какому-то локальному порту и сетевому интерфейсу
2. Создается буфер (байтовый массив) для хранения получаемых данных	2. Создается буфер (байтовый массив) для хранения отправляемых данных
3. Создается объект <code>DatagramPacket</code> на основе этого буфера	3. Создается объект <code>DatagramPacket</code> , на основе уже созданного буфера с данными для отправки, с указанием адреса хоста и номера порта объекта <code>DatagramSocket</code> , созданного получателем
4. Вызывается метод <code>receive()</code> на созданном объекте <code>DatagramSocket</code> с созданным объектом <code>DatagramPacket</code> в качестве аргумента	4. Вызывается метод <code>receive()</code> на созданном объекте <code>DatagramSocket</code> с созданным объектом <code>DatagramPacket</code> в качестве аргумента

Простой пример

Рассмотрим пример простого демонстрационного приложения, реализующего эту схему. Сервер создает сокет и ожидает получения датаграммы от 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. Если клиент отправил команду *stop*, то сервер заканчивает работу.

Серверная часть приложения:

```
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        //1.
        DatagramSocket serverSocket = new DatagramSocket(9876);
        //2.
        byte[] receiveData = new byte[1024];
        byte[] sendData = null;
        System.out.println("Server is ready for requests");
        while (true) {
            //3.
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
                receiveData.length);
```

```

//4.
serverSocket.receive(receivePacket);
String sentence = new
String(receivePacket.getData(),receivePacket.getOffset(),receivePacket.getLength());
System.out.print("\tRECEIVED: " + sentence);
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
System.out.println(" From: " + IPAddress + " port: " + port);
receivePacket.setLength(1024);
String capitalizedSentence = sentence.toUpperCase();
System.out.println("\tSEND: " + capitalizedSentence);
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
if (sentence.trim().equalsIgnoreCase("stop")) {
    break;
}
}
System.out.println("Server stopped");
}
}

```

Клиентская часть приложения:

```

package lec_test2;

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

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

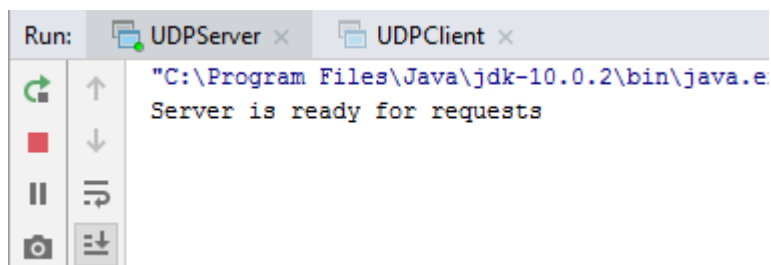
        //1.
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("localhost");
        byte[] sendData = null;
        byte[] receiveData = new byte[1024];
        System.out.print("> ");
        String sentence = inFromUser.readLine();
        System.out.println("Data: " + sentence + " from " +
clientSocket.getLocalAddress() +
            " port: " + clientSocket.getLocalPort());

        //2.
        sendData = sentence.getBytes();
        //3.
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
IPAddress, 9876);
        //4.
        clientSocket.send(sendPacket);
        DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}

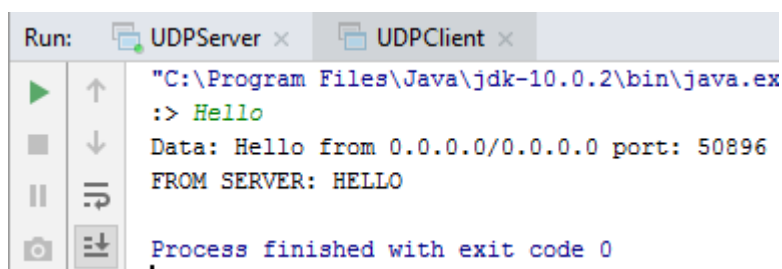
```

Рассмотрим процесс запуска такого простейшего клиент / серверного приложения.

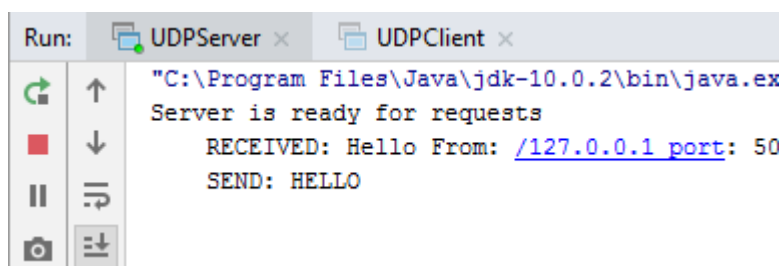
Сначала запускаем серверную часть. Создается сокет и приступает к ожиданию запросов:



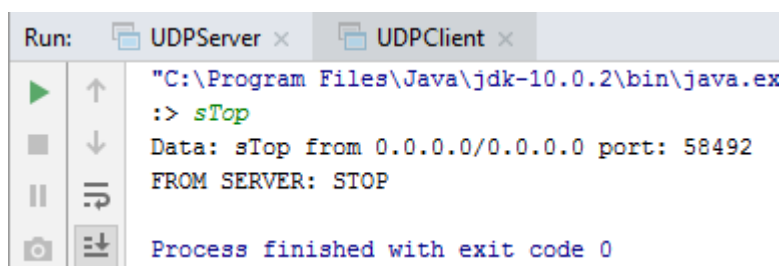
Запускаем клиентскую часть приложения. Вводим данные и получаем ответ от сервера:



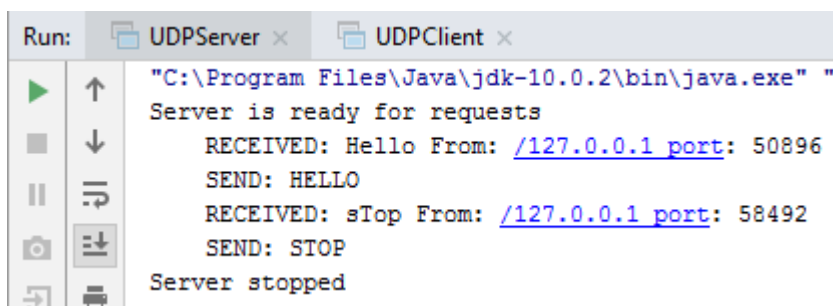
Приведем информацию, выведенную на экран на серверной стороне:



Запустим клиента, который даст команду на завершение работы сервера:



Серверная часть закончит работу.



Примеры приложений

Рассмотрим примеры простых демонстрационных сетевых приложений, реализованных с помощью *UDP* 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.net.*;
import java.io.*;

public class UDPFactClient {
    public static void main(String args[]) throws Exception {
        byte[] rbuf = new byte[1024], sbuf = new byte[1024];
        BufferedReader fromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket socket = new DatagramSocket();
        //InetAddress addr = InetAddress.getByName(args[0]);
        InetAddress addr = InetAddress.getByName("localhost");
        //get an integer from user
        System.out.print("Enter an integer: ");
        String data = fromUser.readLine();
        sbuf = data.getBytes();
        DatagramPacket spkt = new DatagramPacket(sbuf, sbuf.length, addr, 5000);
        //send it to server
        socket.send(spkt);
        System.out.println("Sent to server: " + data);
        DatagramPacket rpkt = new DatagramPacket(rbuf, rbuf.length);
        //retrieve result
        socket.receive(rpkt);
        data = new String(rpkt.getData(), 0, rpkt.getLength());
        System.out.println("Received from server: " + data);
        //close the socket
        socket.close();
    }
}
```

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

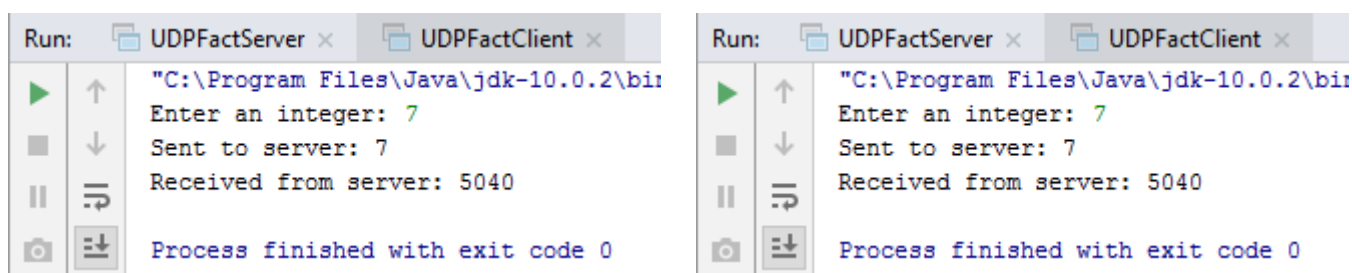
public class UDPFactServer {
    public static void main(String args[]) throws Exception {
        byte[] rbuf = new byte[10], sbuf = new byte[10];
```

```

//create a server socket at port 5000
DatagramSocket socket = new DatagramSocket(5000);
System.out.println("Server ready");
DatagramPacket rpkt = new DatagramPacket(rbuf, rbuf.length);
//receive a packet from client
socket.receive(rpkt);
//extract data and client information from this packet
String data = new String(rpkt.getData(), 0, rpkt.getLength());
InetAddress addr = rpkt.getAddress();
int port = rpkt.getPort();
int fact = 1, n = Integer.parseInt(data);
System.out.println("Received: " + n + " from " + addr + ":" + port);
for (int i = 2; i <= n; i++)
    fact *= i;
sbuf = String.valueOf(fact).getBytes();
DatagramPacket spkt = new DatagramPacket(sbuf, sbuf.length, addr, port);
//send result to the client
socket.send(spkt);
System.out.println("Sent: " + fact);
    }
}

```

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



Будем понемногу усложнять приложение. Спроектируем серверную часть приложения так, чтобы сервер мог обрабатывать несколько запросов либо от одного, либо от многих клиентов. Вариант последовательной обработки рассматривать не будем – он аналогичен первому рассмотренному примеру. Рассмотрим вариант усложнения приложения для обработки нескольких запросов, используя параллельный подход.

Схема работы приложения аналогична многопоточковому *UDP* серверу. Единственное существенное отличие – в новый поток выполнения передается не сокет, а датаграмма, которая в нем обрабатывается.

Приведем пример кода серверной части приложения:

```

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

class Handler implements Runnable {
    DatagramSocket socket;
    DatagramPacket pkt;

    Handler(DatagramPacket pkt, DatagramSocket socket) {
        this.pkt = pkt;
    }
}

```

```

        this.socket = socket;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("\tNew thread created: " +
Thread.currentThread().getName());
        try {
            byte[] sbuf = new byte[10];
            //extract data and client information from this packet
            String data = new String(pkt.getData(), 0, pkt.getLength());
            InetAddress addr = pkt.getAddress();
            int port = pkt.getPort();
            int fact = 1, n = Integer.parseInt(data);
            System.out.println("Received: " + n + " from " + addr + ":" + port);
            for (int i = 2; i <= n; i++)
                fact *= i;
            sbuf = String.valueOf(fact).getBytes();
            DatagramPacket spkt = new DatagramPacket(sbuf, sbuf.length,
                addr, port);
            //send result to the client
            socket.send(spkt);
            System.out.println("Sent: " + fact);
        } catch (IOException e) {
        }
        System.out.println("\tThread finished: " + Thread.currentThread().getName());
    }
}

```

```

import java.net.*;

public class UDPMTFactServer {
    public static void main(String args[]) throws Exception {
        //create a server socket at port 5000
        DatagramSocket socket = new DatagramSocket(5000);
        System.out.println("Server ready");
        while (true) {
            byte[] rbuf = new byte[10];
            DatagramPacket rpkt = new DatagramPacket(rbuf, rbuf.length);
            //receive a packet from client
            socket.receive(rpkt);
            System.out.println("Receiver a packet");
            //hand over this packet to Handler
            new Handler(rpkt, socket);
        }
    }
}

```

Далее приведем немного измененный код клиентской части приложения:

```

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

public class UDPMTFactClient {
    public static void main(String args[]) throws Exception {
        byte[] rbuf = new byte[1024], sbuf = new byte[1024];
        BufferedReader fromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket socket = new DatagramSocket();
    }
}

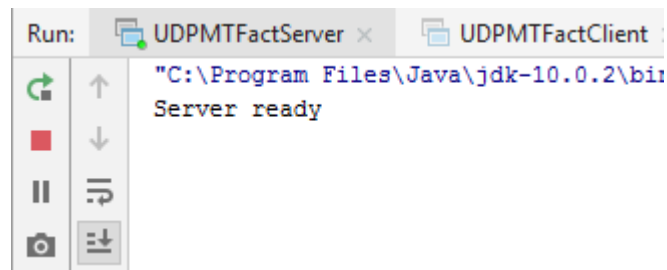
```

```

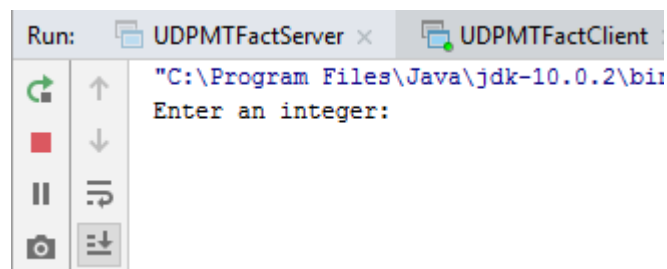
//InetAddress addr = InetAddress.getByName(args[0]);
InetAddress addr = InetAddress.getByName("localhost");
while (true) {
    //get an integer from user
    System.out.print("Enter an integer: ");
    String data = fromUser.readLine();
    if (data.equals("-1")) break;
    sbuf = data.getBytes();
    DatagramPacket spkt = new DatagramPacket(sbuf, sbuf.length,
        addr, 5000);
    //send it to server
    socket.send(spkt);
    System.out.println("Sent to server: " + data);
    DatagramPacket rpkt = new DatagramPacket(rbuf, rbuf.length);
    //retrieve result
    socket.receive(rpkt);
    data = new String(rpkt.getData(), 0, rpkt.getLength());
    System.out.println("Received from server: " + data);
}
//close the socket
socket.close();
}
}

```

Рассмотрим работу приложения. Сначала запустим серверную часть приложения. Будет создан сокет, прослушивающий заданный при его создании порт. После вызова метода `receive()` на созданном сокете будет выполнено блокирующее ожидание поступающей от клиента датаграммы.



На следующем этапе запускаем клиентские части приложения. После запуска первого клиента создается *UDP* сокет для обмена датаграммами, после чего пользователь должен ввести целое число для вычисления.



В отличие от аналогичного примера с *TCP* сокетами сейчас не установлено никакого соединения с сервером. Только после ввода информации будет создана датаграмма и отправлена на сервер.

```

Run: UDPMTFactServer x UDPMTFactClient
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Enter an integer: 3
Sent to server: 3
Received from server: 6
Enter an integer:

```

```

Run: UDPMTFactServer x UDPMTFactClient
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Server ready
Receiver a packet
    New thread created: Thread-0
Received: 3 from /127.0.0.1:59310
Sent: 6
    Thread finished: Thread-0

```

Аналогичным образом обрабатываются пакеты от другого хоста. Следует обратить внимание на отсутствие постоянного соединения между сервером и клиентом – как только датаграмма пришла, неважно откуда, будет создан новый, отдельный поток исполнения для ее обработки и отправки результата клиенту.

```

"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Enter an integer: 3
Sent to server: 3
Received from server: 6
Enter an integer: 5
Sent to server: 5
Received from server: 120
Enter an integer: -1
Process finished with exit code 0

```

```

"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Enter an integer: 3
Sent to server: 3
Received from server: 6
Enter an integer: -1
Process finished with exit code 0

```

Сервер так работает со всеми клиентами.

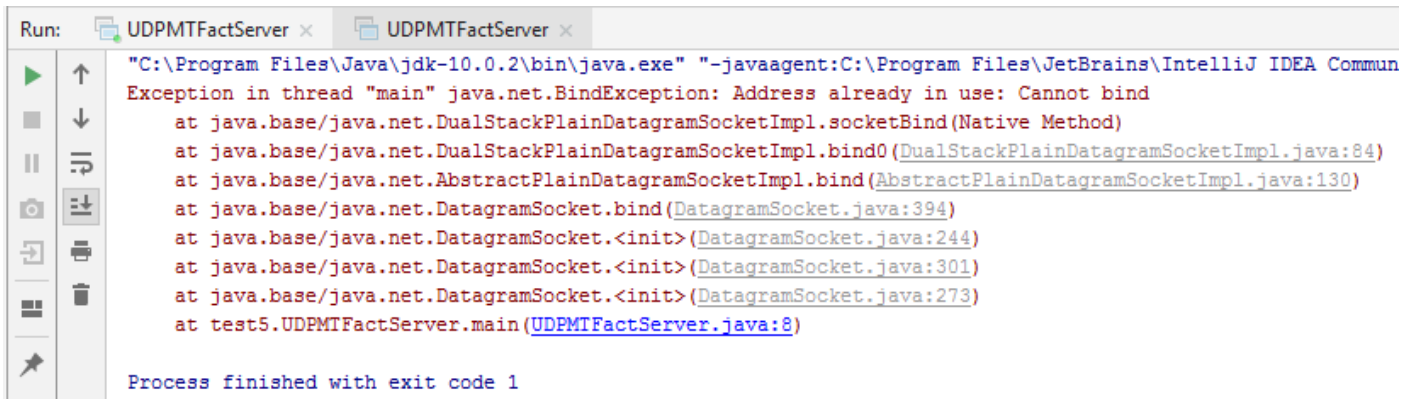
```

Server ready
Receiver a packet
    New thread created: Thread-0
Received: 3 from /127.0.0.1:59310
Sent: 6
    Thread finished: Thread-0
Receiver a packet
    New thread created: Thread-1
Received: 3 from /127.0.0.1:54735
Sent: 6
    Thread finished: Thread-1
Receiver a packet
    New thread created: Thread-2
Received: 5 from /127.0.0.1:54735
Sent: 120
    Thread finished: Thread-2

```

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

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



```
Run: UDPMTFactServer x UDPMTFactServer x
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Commun
Exception in thread "main" java.net.BindException: Address already in use: Cannot bind
    at java.base/java.net.DualStackPlainDatagramSocketImpl.socketBind(Native Method)
    at java.base/java.net.DualStackPlainDatagramSocketImpl.bind0(DualStackPlainDatagramSocketImpl.java:84)
    at java.base/java.net.AbstractPlainDatagramSocketImpl.bind(AbstractPlainDatagramSocketImpl.java:130)
    at java.base/java.net.DatagramSocket.bind(DatagramSocket.java:394)
    at java.base/java.net.DatagramSocket.<init>(DatagramSocket.java:244)
    at java.base/java.net.DatagramSocket.<init>(DatagramSocket.java:301)
    at java.base/java.net.DatagramSocket.<init>(DatagramSocket.java:273)
    at test5.UDPMTFactServer.main(UDPMTFactServer.java:8)
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.

Если время обработки одной датаграммы мало (меньше чем время, затрачиваемое на создание отдельного потока исполнения), то можно работать и с последовательным решением.

Отправка и прием объектов по сети

На прошлой лекции мы уже обсудили, что отправлять / получать данные в виде объектов гораздо удобнее, чем работать с ними в виде «необработанного потока байт». Было разработано простое приложение, в котором объект типа `Message` отправляется одной программой и принимается другой программой с использованием *TCP* сокетов. Сегодня обсудим, как можно создать аналогичное приложение с использованием *UDP* сокетов.

При использовании протокола *UDP* для передачи объектов, в качестве контейнера данных для обмена используется датаграмма. Таким образом, все данные, которые нужно отправить, сначала сохраняются в байтовом массиве, который затем инкапсулируется в датаграмму, которая отправляется через сеть. На стороне получателя байты данных извлекаются из датаграммы и преобразуются в желаемый формат. Для отправки и получения объекта могут быть выполнены такие шаги.

Отправляемый объект сначала сериализуется, и байтовый образ объекта помещается в байтовый массив. Этот байтовый массив используется для создания датаграммы, которая, сразу после создания, отправляется по сети. Получив датаграмму, получатель извлекает из нее массив байт, который затем используется для восстановления объекта с использованием процедуры десериализации. Рассмотрим как же, можно создать датаграмму на базе заданного объекта и получить объект обратно из датаграммы.

Класс, определяющий объекты, которыми будут обмениваться стороны взаимодействия.

```
class Message implements java.io.Serializable {
    String subject, text;

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

    String getSubject() {
        return subject;
    }

    String getText() {
        return text;
    }
}
```

Класс – приемник сообщения.

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

public class UDPObjReceiver {
    public static void main(String[] args) {
        try {
            //Create a DatagramSocket and bind it to port 8379
            DatagramSocket socket = new DatagramSocket(8379);
            //Construct a DatagramPacket to receive packet
            byte[] in = new byte[256];
            DatagramPacket packet = new DatagramPacket(in, in.length);
            System.out.println("Waiting to receive a Message object...");
            //Receive the packet now and display
            socket.receive(packet);
            ByteArrayInputStream bais = new ByteArrayInputStream(in);
            ObjectInputStream ois = new ObjectInputStream(bais);
            Message msg = (Message) ois.readObject();
            System.out.println("Received a message:");
            System.out.println("subject : " + msg.getSubject() + "\nbody : "
+msg.getText());
        } catch (Exception ioe) {
            System.out.println(ioe);
        }
    }
}
```

Клиентская часть – отправка сообщения.

```
package test5;

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

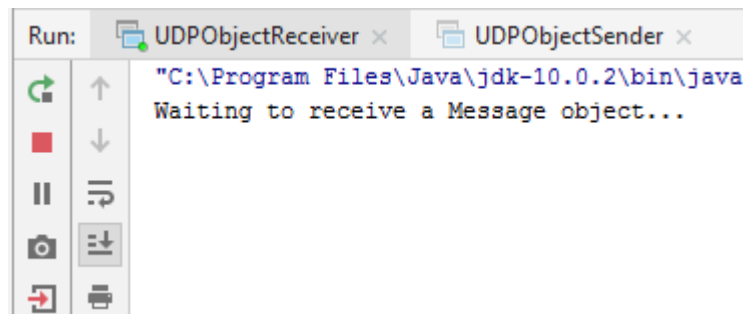
public class UDPObjSender {
    public static void main(String[] args) {
        try {
            //Create a DatagramSocket
```

```

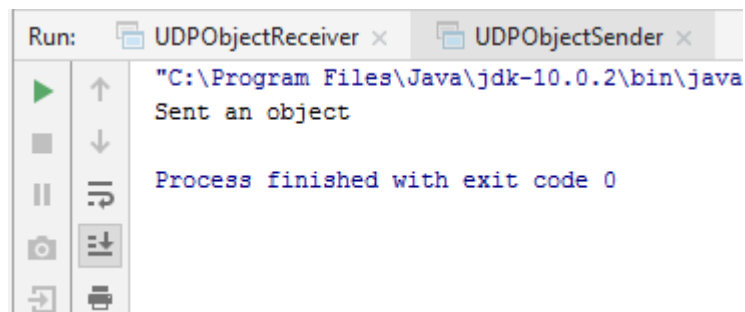
DatagramSocket socket = new DatagramSocket();
//Create a Message object to be sent
Message msg = new Message("Remainder", "Return my book on Monday");
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(msg);
oos.writeObject(msg);
byte[] out = baos.toByteArray();
//Multicast group where packet has to sent
//InetAddress group = InetAddress.getBy_name(args[0]);
InetAddress group = InetAddress.getBy_name("localhost");
//Port the receiver listens on
int port = 8379;
//Create a DatagramPacket with buffer, address and port
DatagramPacket packet = new DatagramPacket(out, out.length, group, port);
//Send the packet now
socket.send(packet);
System.out.println("Sent an object");
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

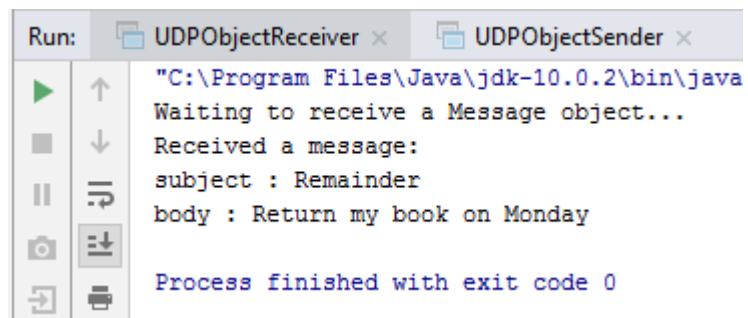
Запускаем приложение. Сначала запускается серверная часть – приемник сообщений и ожидается подключение клиентов.



Создаем клиента, отправляем сообщение и отправляем его по сети:



Сервер получил сериализованный объект и восстановил его.



The screenshot shows a Java IDE's Run console. At the top, there are two tabs: "UDPObjectReceiver" and "UDPObjectSender". The console output is as follows:

```
Run: UDPObjectReceiver x UDPObjectSender x
"C:\Program Files\Java\jdk-10.0.2\bin\java
Waiting to receive a Message object...
Received a message:
subject : Remainder
body : Return my book on Monday
Process finished with exit code 0
```