

Тема: Remote Method Invocation (Part 2)

План занятия:

1. [Служба реестра](#)
2. [Использование *RMI URL*](#)
3. [Организация обратного вызова](#)
 - [Простое удаленное приложение](#)
 - [Создание удаленных интерфейсов](#)
 - [Реализация интерфейсов](#)
 - [Создание сервера](#)
 - [Создание клиента](#)
 - [Запуск приложения](#)
 - [Другое приложение демонстрирующее обратный вызов](#)
4. [Динамическая загрузка классов](#)
 - [Пример](#)
 - [Создание RMI сервера](#)
 - [Создание интерфейса](#)
 - [Реализация интерфейсов](#)
 - [Реализация сервера](#)
 - [Создание клиента](#)
 - [Развертывание, компиляция и запуск приложения](#)
 - [Компиляция и распространение интерфейсов](#)
 - [Компиляция серверной части](#)
 - [Компиляция клиентской части](#)
 - [Запуск распределенного приложения](#)

Литература

1. Кей Хорстманн, Гари Корнелл «*Java. Библиотека профессионала. Том 2*»
2. Harold E. R. *Java Network Programming*: - O'Reilly, 2005 – 735 p.
3. Trail: RMI: <https://docs.oracle.com/javase/tutorial/rmi/index.html>
4. Java™ Remote Method Invocation API (Java RMI):
<https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/>
5. Java Remote Method Invocation:
<https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html>
6. Getting Started Using Java™ RMI:
<https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html>

7. Remote Method Invocation: <https://www.oracle.com/technetwork/java/rmi-141556.html>
8. Jan Graba *An Introduction to Network Programming with Java*, 2013
9. Роберт Орфали, Дан Харки. *Java и CORBA в приложениях клиент-сервер*. Издательство: Лори, 2000 - 734 с.
10. Патрик Нимейер, Дэниэл Леук. *Программирование на Java* (Исчерпывающее руководство для профессионалов). М.: Эксмо, 2014
11. William Grosso. *Java RMI*. O'Reily, 2001, p. 752
12. Esmond Pitt, Kathleen McNiff. *java(TM).rmi: The Remote Method Invocation Guide*. Addison-Wesley, 2001, p. 320

Служба реестра

Давайте немного подробнее обсудим службу *RMI* реестра. По своему смыслу реестр – это приложение, с помощью которого серверная часть приложения регистрирует предоставляемые сервисы, и то место, где клиенты могут запросить и получить эти сервисы. Давайте разберемся, как может работать такое приложение.

Когда служба реестра запускается или с помощью команды `rmiregistry`, или с помощью метода `Registry.createRegistry()`, то создается объект типа `ServerSocket`, который ожидает входящих соединений. Когда клиентская программа вызывает метод `lookup()`, то при этом устанавливается соединение с помощью сокета с удаленным приложением – реестром *RMI*, и после установки соединения отправляет через него имя объекта, указанное в качестве аргумента. Удаленный реестр находит заглушку с указанным именем и отправляет этот объект по сети обратно клиенту при помощи механизма сериализации. После получения сериализованных данных метод `lookup()` реконструирует объект и возвращает его вызывающей стороне. Возвращаемый объект является клиентским *прокси* серверного объекта и знает, как взаимодействовать с объектом через серверный *прокси*. Таким образом, если каким-либо образом создать экземпляр заглушки на стороне клиента, то становится возможным организовать вызов метода для удаленного объекта.

В том же пакете самостоятельно создадим собственную простую службу реестра, которая ведет себя так, как описано выше.

```
package all_in_one;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.PrintWriter;
import java.net.ServerSocket;
```

```
import java.net.Socket;
import java.rmi.Remote;
import java.util.Hashtable;

public class SimpleRegistry implements Runnable {

    int port;
    Hashtable objects = new Hashtable();

    public SimpleRegistry(int prt) {
        this.port = prt;
        new Thread(this).start();
    }

    public SimpleRegistry() {
        this(6789);
    }

    public void rebind(Remote o, String name) {
        objects.put(name, o);
    }

    public static Object lookup(String host, int port, String name) throws
        IOException, ClassNotFoundException {
        Socket clientEnd = new Socket(host, port);
        PrintWriter toServer = new
            PrintWriter(clientEnd.getOutputStream(), true);
        toServer.println(name);
        ObjectInputStream in = new
            ObjectInputStream(clientEnd.getInputStream());
        return in.readObject();
    }

    public void run() {
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            while (true) {
                Socket serverEnd = serverSocket.accept();
                BufferedReader fromClient = new BufferedReader(new
                    InputStreamReader(serverEnd.getInputStream()));
                String name = fromClient.readLine();
                Remote o = (Remote) objects.get(name);
                ObjectOutputStream oos = new
```

```

        ObjectOutputStream(serverEnd.getOutputStream());
        oos.writeObject(o);
    }
} catch (Exception e) {
}
}
}

```

Для работы с этой службой немного изменим код сервера:

```

package all_in_one;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class GServer implements Greeting {

    private int numberOfVisitors;

    public GServer() {
        this.numberOfVisitors = 0;
    }

    public int getNumberOfVisitors() {
        return this.numberOfVisitors;
    }

    @Override
    public String greet(String name) {
        return "Hello, dear " + name + ". You are the " +
            (++this.numberOfVisitors);
    }

    public static void main(String args[]) {
        int port = 1099;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        }
        System.out.println("port: " + port);
        GServer server = new GServer();
        try {

```

```

        Greeting stub = (Greeting)
            UnicastRemoteObject.exportObject(server, 0);
        //Registry registry = LocateRegistry.getRegistry();
        //Registry registry = LocateRegistry.getRegistry(port);
        String name = "Greet";
        //registry.rebind(name, stub);
        SimpleRegistry reg = new SimpleRegistry();
        reg.rebind(stub, name);
        System.out.println("The server object is ready for clients");
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
}

```

Особое внимание обратим на следующие строчки кода серверной стороны (закомментировано старое, указано новое):

```

//Registry registry = LocateRegistry.getRegistry();
//registry.rebind(name, stub);
SimpleRegistry reg = new SimpleRegistry();
reg.rebind(stub, name);

```

Кроме того, придется немного модифицировать клиентскую часть приложения

```

package all_in_one;

import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Scanner;

public class GClient {

    public static void main(String[] args) {
        String host = "localhost";
        int port = 1099;
        if (args.length > 0) {
            host = args[0];
            port = Integer.parseInt(args[1]);

```

```

}
System.out.println("Host: " + host + "\tPort: " + port);
Registry registry;
try {
    //registry = LocateRegistry.getRegistry("localhost");
    //registry = LocateRegistry.getRegistry(host, port);
    //Greeting stub = (Greeting) registry.lookup("Greet");
    Greeting stub = (Greeting) SimpleRegistry.lookup(host, port,
                                                    "Greet");

    Scanner in = new Scanner(System.in);
    System.out.print("Your name: ");
    String name = in.nextLine();
    System.out.println(stub.greet(name));
} //catch (RemoteException | NotBoundException | IOException e) {
catch (RemoteException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
}
}

```

Как и в коде серверной стороны, исходные строки в клиенте закомментированы и заменены на новые:

```

//registry = LocateRegistry.getRegistry(host, port);
//Greeting stub = (Greeting) registry.lookup("Greet");
Greeting stub = (Greeting) SimpleRegistry.lookup(host, port, "Greet");

```

Код удаленного интерфейса остался без изменения. Запустим приложение. Так как служба реестра у нас нестандартная, то и запуск этого распределенного приложения будет отличаться от стандартного.

Служба реестра у нас теперь своя собственная. Реестр создается программно, а не запускается как отдельная программа, и класс-файл реестра теперь должен и на серверном компьютере, и на клиентском. (В принципе, при использовании встроенной службы реестра соответствующие класс-файлы также есть на серверной и на клиентской стороне).

Распределим файлы по каталогам: в разных каталогах разместим серверную и клиентскую части:

different

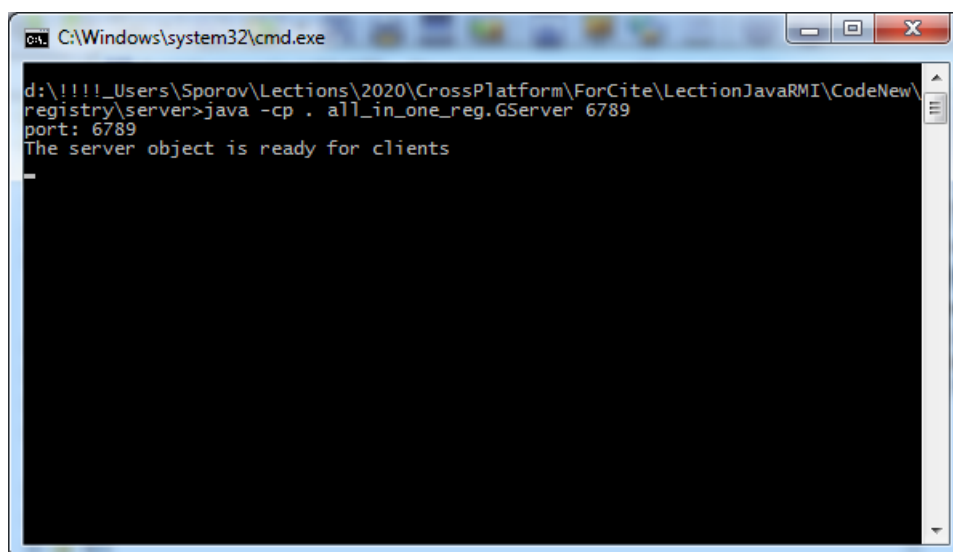
```
server
    all_in_one
        Greeting.java
        GServer.java
        SimpleRegistry.java
client
    all_in_one
        Greeting.java
        GClient.java
        SimpleRegistry.java
```

Сначала стандартным способом откомпилируем классы (см. команду компиляции в предыдущей лекции, либо это можно сделать в любой среде разработки (даже в *DrJava*)).

Находясь в каталоге `server`, выполним команду запуска серверной части приложения:

```
java -cp . all_in_one_reg.GServer 6789
```

Будет запущена серверная часть приложения и на экран будет выведено терминальное окно:



После этого переходим в каталог `client` и даем команду для запуска клиента:

```
java -cp . all_in_one.GClient localhost 6789
```

Выполним эту команду несколько раз и увидим, что приложение ведет себя так же, как и предыдущая версия приложения со встроенной службой *RMI* реестра:

```

Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
registry\client>2.bat

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
registry\client>java -cp . all_in_one_reg.GClient localhost 6789
Host: localhost Port: 6789
Your name: User 1
Hello, dear User 1. You are the 1

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
registry\client>2.bat

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
registry\client>java -cp . all_in_one_reg.GClient localhost 6789
Host: localhost Port: 6789
Your name: User 2
Hello, dear User 2. You are the 2

d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
registry\client>_

```

Использование *RMI URL*

В нашем прошлом примере «вежливого» распределенного приложения серверная часть явно экспортирует, а затем и регистрирует в службе *RMI* реестра удаленный объект с помощью довольно длительной процедуры, состоящей из нескольких последовательных шагов. Клиентская часть распределенного приложения, в свою очередь, получает ссылку на удаленный объект с также помощью нескольких последовательных шагов. В некоторых случаях технология *Java RMI* позволяет использовать простой механизм адресации *RMI URL*, который может использоваться и сервером и клиентом для решения указанных задач.

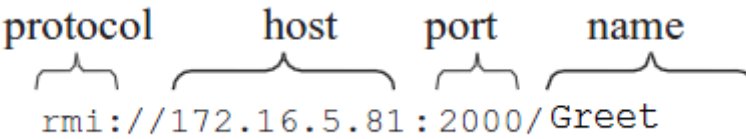
При использовании этого способа адресации, серверное приложение создает удаленный объект и регистрирует его в *RMI* реестре с помощью статического метода `getbind()` класса `java.rmi.Naming` (смю документацию по классу <https://docs.oracle.com/javase/8/docs/api/java/rmi/Naming.html>). Серверное приложение должно указать номер порта, на котором работает приложение *RMI* реестра, а также или *IP*-адрес, или *имя компьютера*, где запущена эта служба. Кроме этого, как и раньше, серверная часть также должна указать логическое имя, под которым объект будет известен клиентам. Клиенты будут использовать это имя для того, чтобы получить удаленную ссылку на этот объект.

При таком способе адресации вся эта информация может быть записана в виде *RMI URL*. Этот вид адреса очень похож на *HTTP URL*-адрес; единственное существенное отличие состоит в том, что в качестве имени протокола используется «*rmi*». Такой адрес выглядит так:

`rmi://host:[port]/objectName`

В этой записи `host` - это или *IP*-адрес или полное доменное имя (*Fully Qualified Domain Name, FQDN*) компьютера, на котором работает *RMI* реестр. Необязательный `port` — это номер порта, на котором работает служба *RMI* реестра. Как и раньше, номер порта по умолчанию - 1099. `objectName` — это логическое имя объекта. Например, если *RMI* реестр работает на компьютере с

IP-адресом 172.16.5.81 на порте 2000, а имя объекта – «Приветствуйте» (*Greet*), то *RMI URL*-адрес этого объекта будет таким:



В случае, когда *RMI* реестр работает на порте по умолчанию (1099), приведенный выше *URL*-адрес сокращается до

```
rmi://172.16.5.81/Greet
```

Если *RMI* реестр и серверная часть распределенного *RMI* приложения работают на одном компьютере, то приведенный выше *URL*-адрес сокращается еще сильнее до

Greet

А поскольку архитектура *Java RMI* поддерживает запуск службы реестра (в стандартной реализации) и *RMI* сервера только на одном компьютере, то для серверной части *RMI* приложения *URL* нашего удаленного объекта будет просто его логическое имя *Greet*.

Рассмотрим фрагмент кода, в котором создается удаленный объект и регистрируется с помощью метода `Naming.rebind()` (см. справочную статью <https://docs.oracle.com/javase/8/docs/api/java/rmi/Naming.html#rebind-java.lang.String-java.rmi.Remote->) в службе *RMI* реестра:

```
String url = "Greet";
GServer server = new GServer();
Naming.rebind(url, server);
```

Метод `java.rmi.Naming.rebind()`, по смыслу работы, парсит (*parse*) *URL*-адрес, получает ссылку на *RMI* реестр с помощью метода `LocateRegistry.getRegistry()` и, наконец, вызывает метод `rebind()` для этого реестра.

Рассмотрим фрагмент кода, демонстрирующий возможную реализацию метода `java.rmi.Naming.rebind()`:

```
package java.rmi;
import java.rmi.registry.*;
//...
public final class Naming {
//...
public static void rebind(String name, Remote obj)
```

```

throws RemoteException, java.net.MalformedURLException {
    ParsedNamingURL parsed = parseURL(name);
    Registry registry = getRegistry(parsed);
    if (obj == null)
        throw new
            NullPointerException("cannot bind to null");
    registry.rebind(parsed.name, obj);
}
//...
}

```

Следует обратить внимание на то, что метод `java.rmi.Naming.rebind()` ожидает, что его второй аргумент будет обязательно объектом типа `java.rmi.Remote`. Таким образом, чтобы воспользоваться таким простым методом адресации, следует изменить определение класса `GServer`, сделав его наследником класса `UnicastRemoteObject`, например так:

```

package all_in_one;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

public class GServer extends UnicastRemoteObject
                                implements Greeting {

    private int numberOfVisitors;

    public GServer() throws RemoteException {
        super();
        this.numberOfVisitors = 0;
    }

    public int getNumberOfVisitors() {
        return this.numberOfVisitors;
    }

    @Override
    public String greet(String name) {
        return "Hello, dear " + name + ". You are the " +

```

```

        ( ++this.numberofVisitors);
    }

    public static void main(String args[]) {
        int port = 1099;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        }
        System.out.println("port: " + port);
        String name = "Greet";
        String url = "rmi://localhost:" + port + "/" + name;
        System.out.println("URL: " + url);
        try {
            GServer server = new GServer();
            Naming.rebind(url, server);
            System.out.println("The server object is ready for clients");
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}

```

В приведенном выше коде жирным шрифтом выделены строки, отличающие этот вариант приложения от предыдущего.

Метод `Naming.rebind()` регистрирует удаленный объект с именем «Greet» в реестре, работающем на той же машине на указанном порту. При этом процедура запуска `rmiregistry` и *RMI* сервера такая же, как и раньше.

Клиентская часть распределенного *RMI* приложения также может использовать *RMI URL* для получения удаленной ссылки. Если предположить, что *RMI* реестр будет работать на порте по умолчанию, то *URL* удаленного объекта для клиента будет выглядеть так:

```
String url = "rmi://" + args[0] + "/Greet";
```

Формат строки адреса аналогична рассмотренной ранее: логическое имя удаленного объекта - «Greet»; первый аргумент командной строки `args[0]` - *IP*-адрес или полное доменное имя (*Fully Qualified Domain Name, FQDN*) компьютера, на котором работает серверная часть удаленного приложения.

```
Greeting stub = (Greeting)Naming.lookup(url);
```

Приведенный выше код связывается с *RMI* реестром, работающим на том компьютере, адрес которого указан в аргументе командной строки, и запрашивает заглушку для объекта, зарегистрированного под именем Greet. Остальной клиентский код остается таким же, как и в прошлом примере. Рассмотрим полный код клиента:

```
package all_in_one;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.Scanner;

public class GClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 1099;
        if (args.length > 0) {
            host = args[0];
            port = Integer.parseInt(args[1]);
        }
        System.out.println("Host: " + host + "\tPort: " + port);
        String name = "Greet";
        String url = "rmi://localhost:" + port + "/" + name;
        System.out.println("URL: " + url);
        try {
            Greeting stub = (Greeting) Naming.lookup(url);
            Scanner in = new Scanner(System.in);
            System.out.print("Your name: ");
            String userName = in.nextLine();
            System.out.println(stub.greet(userName));
        } catch (RemoteException | NotBoundException |
            MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

Как и для случая серверной части приложения, новые фрагменты выделены полужирным шрифтом. Код удаленного интерфейса остается таким же, как и раньше:

```

package all_in_one;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Greeting extends Remote {
    public String greet(String name) throws RemoteException;
}

```

Процедура запуска приложения аналогичная той, что была использована ранее. Как и раньше предположим, что наше удаленное приложение разделено на серверную и клиентскую части, которые размещены в разных папках компьютера (может даже и на разных хостах):

```

naming
  server
    all_in_one
      Greeting.java
      GServer.java
    test
    Web
      RMI
        all_in_one
          Greeting.class

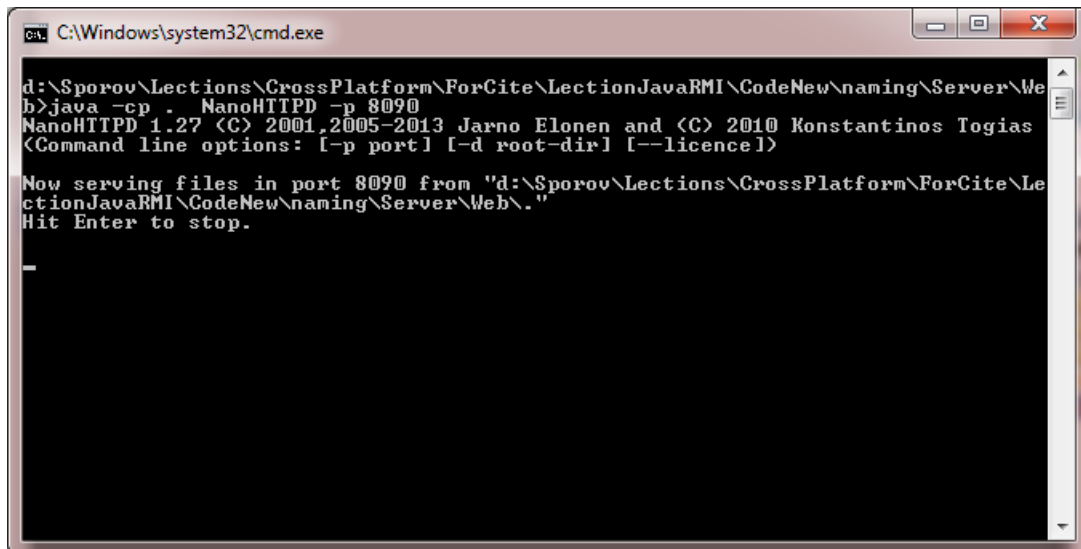
  client
    all_in_one
      Greeting.java
      GClient.java

```

В каталоге **Web** размещен файл **NanoHTTPD.java** с исходным кодом простого **Web** – сервера. Как и раньше, сначала откомпилируем файлы с исходным кодом нашего приложения. Далее, первым действием из каталога **Web** запустим *HTTP* – сервер:

```
java -cp . NanoHTTPD -p 8090
```

На экран будет выведено терминальное окно, в которое выводятся служебные сообщения *HTTP* – сервера:



```

C:\Windows\system32\cmd.exe
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Server\Web>java -cp . NanoHTTPD -p 8090
NanoHTTPD 1.27 (C) 2001, 2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togias
<Command line options: [-p port] [-d root-dir] [--licence]>

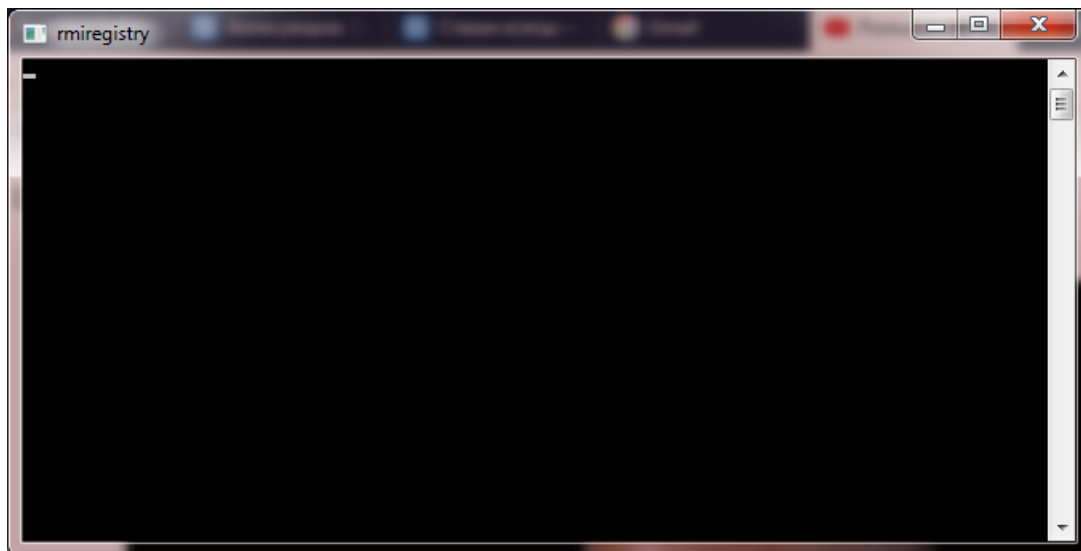
Now serving files in port 8090 from "d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Server\Web\."
Hit Enter to stop.

```

После этого из каталога test запускаем службе *RMI* реестра:

```
start "rmiregistry" rmiregistry 6789
-J-Djava.rmi.server.useCodebaseOnly=false
```

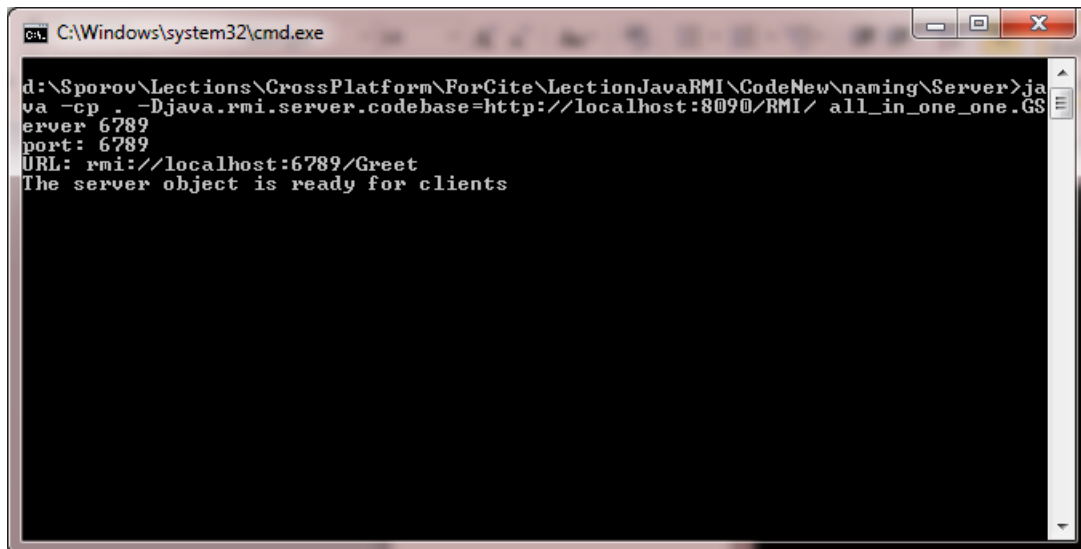
При таком варианте запуска кодовая база реестру передается от серверной части приложения по *RMI* каналу. Самостоятельно можно попробовать другой способ запуска приложения.



Затем из каталога server запускаем серверную часть приложения. Для этого в этом каталоге в выполним команду:

```
java -cp . -Djava.rmi.server.codebase=http://localhost:8090/RMI/
all_in_one.GServer 6789
```

На экран будет выведено терминальное окно, содержащее служебные сообщения серверной части приложения:



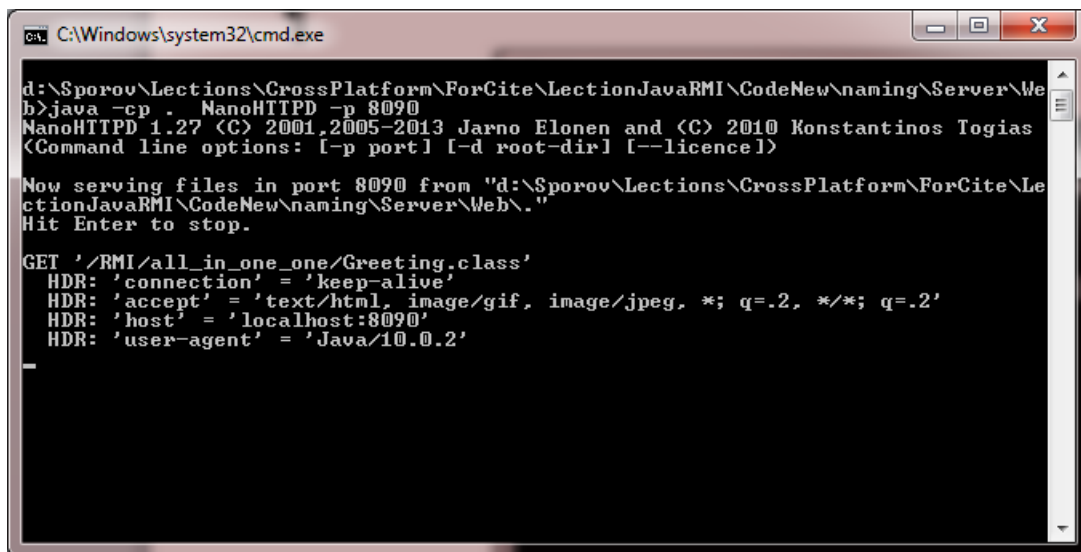
```

C:\Windows\system32\cmd.exe

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Server>java -cp . -Djava.rmi.server.codebase=http://localhost:8090/RMI/ all_in_one_one.GS
erver 6789
port: 6789
URL: rmi://localhost:6789/Greet
The server object is ready for clients

```

С помощью нового способа удаленный объект будет автоматически экспортирован и зарегистрирован в службе реестра. При этом, как и в прошлом случае, класс-файл удаленного интерфейса будет доставлен службе *RMI* реестра при помощи *HTTP*-сервера.



```

C:\Windows\system32\cmd.exe

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Server\Web>java -cp . NanoHTTPD -p 8090
NanoHTTPD 1.27 <C> 2001,2005-2013 Jarno Elonen and <C> 2010 Konstantinos Trogias
<Command line options: [-p port] [-d root-dir] [--licence]>

Now serving files in port 8090 from "d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Server\Web\"
Hit Enter to stop.

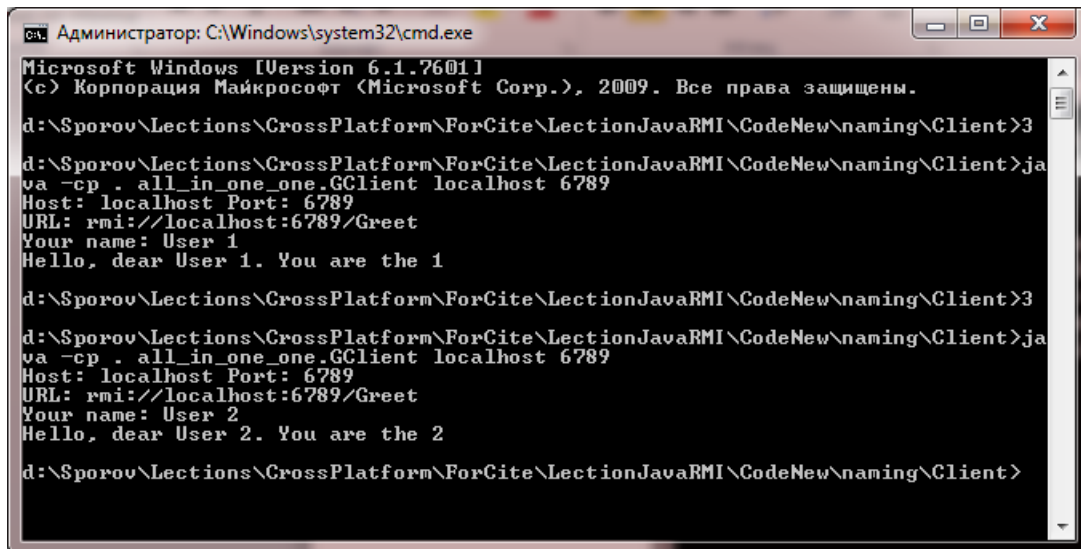
GET '/RMI/all_in_one_one/Greeting.class'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/1.0.0.2'

```

После этого, перейдет в папку client и запустим клиентскую часть приложения при помощи команды:

```
java -cp . all_in_one.GClient localhost 6789
```

На экран будет выведено терминальное окно, в котором работает клиентская часть распределенного приложения. Запустим клиента несколько раз, и убедимся, что приложение работает так же, как и аналогичное на прошлом занятии.



```

Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Client>3
d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Client>ja
va -cp . all_in_one_one.GClient localhost 6789
Host: localhost Port: 6789
URL: rmi://localhost:6789/Greet
Your name: User 1
Hello, dear User 1. You are the 1

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Client>3
d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Client>ja
va -cp . all_in_one_one.GClient localhost 6789
Host: localhost Port: 6789
URL: rmi://localhost:6789/Greet
Your name: User 2
Hello, dear User 2. You are the 2

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\naming\Client>

```

Можно самостоятельно попробовать различные способы запуска такого приложения.

Организация обратного вызова

В типичном *RMI* приложении клиенты вызывают методы на удаленных объектах, созданных сервером. Механизм обратного вызова позволяет серверу вызывать методы на удаленных объектах, созданных и переданных серверу клиентами. Эта схема взаимодействия клиента и сервера бывает необходима во многих случаях. Рассмотрим несколько типичных ситуаций применения обратного вызова.

Рассмотрим случай, когда клиент вызывает метод на удаленном объекте, и работа метода занимает достаточно много времени. Таким образом, клиент должен долго ждать завершения работы метода и возвращения результата. При помощи указанного подхода клиент может сразу после совершения вызова вернуться к своей дальнейшей работе. После завершения работы метода, сервер асинхронно отправит ответ клиенту, вызывая метод на удаленном объекте, который был создан и передан клиентом удаленному объекту, который был создан сервером.

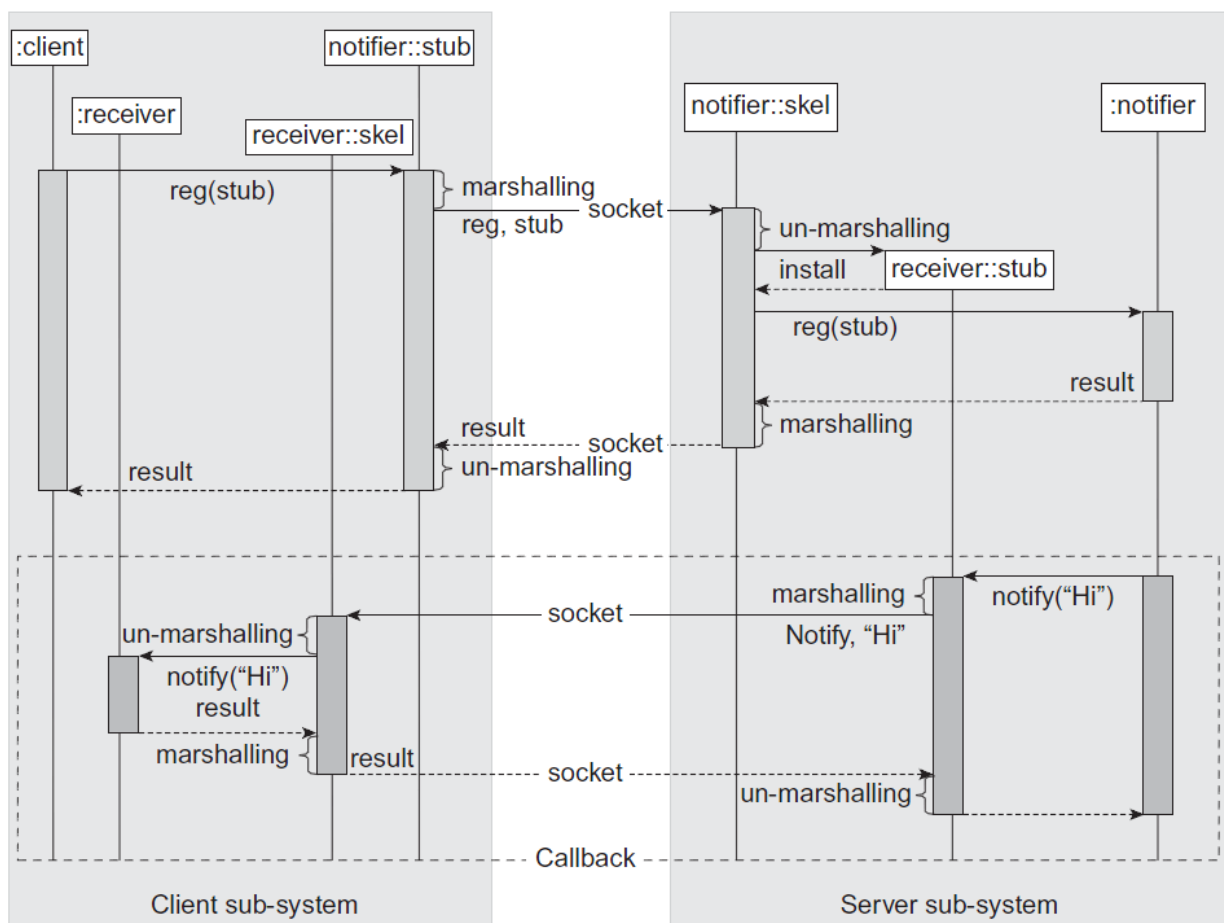
Возможна другая ситуация. Клиент хочет отобразить получить некоторые данные (например, счет игры), как только они поступят на сервер. Поскольку клиент не знает точно, когда новые данные поступят на сервер, он может периодически запрашивать данные с сервера. Это может привести или к простоя, или к нежелательной многократной повторной загрузке одних и тех же данных. Чтобы этого избежать, можно сделать так, чтобы не клиент был должен получать данные от сервера, наоборот, сервер был должен отправлять данные клиенту сразу, как только они поступят.

Реализация механизма обратного вызова формально очень проста. Как уже было сказано на прошлой лекции, система *Java RMI* позволяет передавать не только локальный, но и удаленный объект (класс которого реализует интерфейс *Remote*) в качестве параметра в удаленный метод. Но, для того, чтобы разобраться с особенностями работы механизма обратного вызова,

необходимо понять, что же происходит, когда в рамках технологии *Java RMI* передаются удаленные объекты.

Мы уже обсуждали раньше, что когда выполняется экспорт удаленного объекта, создается экземпляр его скелетона (*skeleton*). Этот скелетон (*skeleton*) «оборачивается» вокруг удаленного объекта и содержит методы, аналогичные методам удаленного объекта. Поэтому, когда клиент сначала создает, затем экспортирует и передает удаленный объект в качестве параметра в удаленный метод, система *Java RMI* выполняет следующие действия:

- создается экземпляр прокси (заглушка, *stub*) для этого удаленного объекта;
- эта заглушка знает всю необходимую информацию (номер порта и IP-адрес) для взаимодействия со скелетоном (*skeleton*) объекта – аргумента метода;
- затем, при помощи механизма сериализации, эта заглушка преобразуется в байтовый массив;
- эти сериализованные данные отправляются получателю (удаленному методу) по сети посредством сокетов;
- на получателе (удаленный метод) восстанавливается и затем устанавливается (*installed*) экземпляр заглушки;
- формальный параметр удаленного метода ссылается на этот восстановленный и установленный экземпляр заглушки.



Таким образом, теперь формальный параметр удаленного метода ссылается на прокси для фактического объекта, который был передан методу клиентом. Поэтому вызов метода через формальный параметр приводит к аналогичному вызову метода в локальном прокси-объекте, который, в свою очередь, перенаправляет информацию о вызове метода реальному объекту. После этого, сервер может асинхронно вызывать удаленные методы на удаленных объектах клиента таким же самым образом, как клиент асинхронно вызывает методы на удаленных объектах сервера.

Простое удаленное приложение

Для отработки такого подхода, давайте разработаем простое удаленное приложение, в котором клиент сначала создает объект-получатель (*receiver*) и экспортирует его; затем он получает ссылку на удаленный объект-уведомитель (*notifier*). Клиент, для того, чтобы зарегистрироваться, вызывает метод `registerMe()` на объекте-уведомителе (*notifier*), передавая этот объект-получатель (*receiver*). Внутри метода `registerMe()` сервер вызывает метод `notify()` на объекте-получателе (*receiver*), при этом передавая строку.

Создание удаленных интерфейсов

Поскольку клиент и сервер отдельно создают свои удаленные объекты, приложению будут нужны два интерфейса.

Рассмотрим интерфейс удаленного объекта, который будет создан клиентом. В интерфейсе будет объявлен один метод `notify()`, который принимает строку.

```
import java.rmi.*;

public interface Receiver extends Remote {
    public void notify(String s) throws RemoteException;
}
```

Интерфейс объекта на стороне сервера выглядит следующим образом:

```
import java.rmi.*;

public interface Notifier extends Remote {
    public void registerMe(Receiver r) throws RemoteException;
}
```

В этом интерфейсе определен единственный метод `registerMe()` для регистрации объекта `Receiver`.

Реализация интерфейсов

Рассмотрим простой класс реализующий интерфейс `Notifier`:

```
import java.rmi.*;

public class SimpleNotifier implements Notifier {
    public void registerMe(Receiver r) {
        try {
            System.out.println("registered the receiver : "+r);
            String msg = "A message from SimpleNotifier";
            r.notify(msg); //callback
            System.out.println("Sent : "+msg);
        } catch (RemoteException e) {}
    }
}
```

Метод registerMe() просто вызывает метод notify() на указанном объекте Receiver.

На стороне клиента, на котором будет реализована служба обратного вызова, нужно создать удаленный объект и экспортировать его. Клиент может сделать это двумя способами:

- создать класс, реализующий удаленный интерфейс и расширяющий или класс `java.rmi.server.UnicastRemoteObject`, или класс `javax.rmi.PortableRemoteObject`. В этом случае объект автоматически экспортируется во время создания.
- создать класс, который реализует удаленный интерфейс, но не расширяет или класс `java.rmi.server.UnicastRemoteObject`, или класс `javax.rmi.PortableRemoteObject`. В этом случае объект экспортируется явно, используя статический метод `exportObject()` или класса `java.rmi.server.UnicastRemoteObject`, или класса `javax.rmi.PortableRemoteObject`.

В нашем простом демонстрационном приложении воспользуемся первым методом. Таким образом, простой класс, реализующий интерфейс Receiver может быть таким:

```
import java.rmi.*;
import java.rmi.server.*;

public class SimpleReceiver extends UnicastRemoteObject
    implements Receiver, java.io.Serializable {
    SimpleReceiver() throws RemoteException {}
    public void notify(String msg) {
        try {
            System.out.println("received : " + msg);
        } catch (Exception e) {e.printStackTrace();}
```

```

    }
}

```

Следует обратить внимание, что этот класс должен имплементировать интерфейс `Serializable`, поскольку *Java RMI* использует механизм сериализации для отправки и получения объектов. Кроме того, класс также расширяет `UnicastRemoteObject`. Таким образом, его экземпляр экспортируется автоматически при создании. Ну а метод `notify()` просто отображает полученную строку.

Создание сервера

Серверная часть нашего распределенного приложения по смыслу аналогична серверной части предыдущего приложения. Рассмотрим исходный код:

```

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class CallbackServer {
    public static void main(String args[]) {
        try {
            String name = "notifier";
            SimpleNotifier notifier = new SimpleNotifier();
            Notifier stub =
                (Notifier)UnicastRemoteObject.exportObject(notifier, 0);
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind(name, stub);
            System.out.println("Notifier ready...");
        } catch (Exception e) { e.printStackTrace(); }
    }
}

```

Создание клиента

Рассмотрим пример простого клиента для нашего распределенного приложения:

```

import java.rmi.*;

public class CallbackClient {
    public static void main(String args[]) {
        try {
            String url = "rmi://" + args[0] + "/notifier";
            Notifier notifier = (Notifier)Naming.lookup(url);

```

```

        SimpleReceiver recv = new SimpleReceiver();
        notifier.registerMe(recv);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Обратите внимание, что экземпляр `recv` не экспортируется явно, поскольку его класс расширяет `UnicastRemoteObject`, он автоматически экспортируется при его создании.

Запуск приложения

Для моделирования ситуации, когда распределенное приложение запускается с двух разных компьютеров, разместим классы по двум разным каталогам, представляющим компьютеры сервера и клиента.

Каталог `server`:

```

Notifier.java
Receiver.java
SimpleNotifier.java
CallbackServer.java

```

Каталог `client`:

```

Notifier.java
Receiver.java
SimpleReceiver.java
CallbackClient.java

```

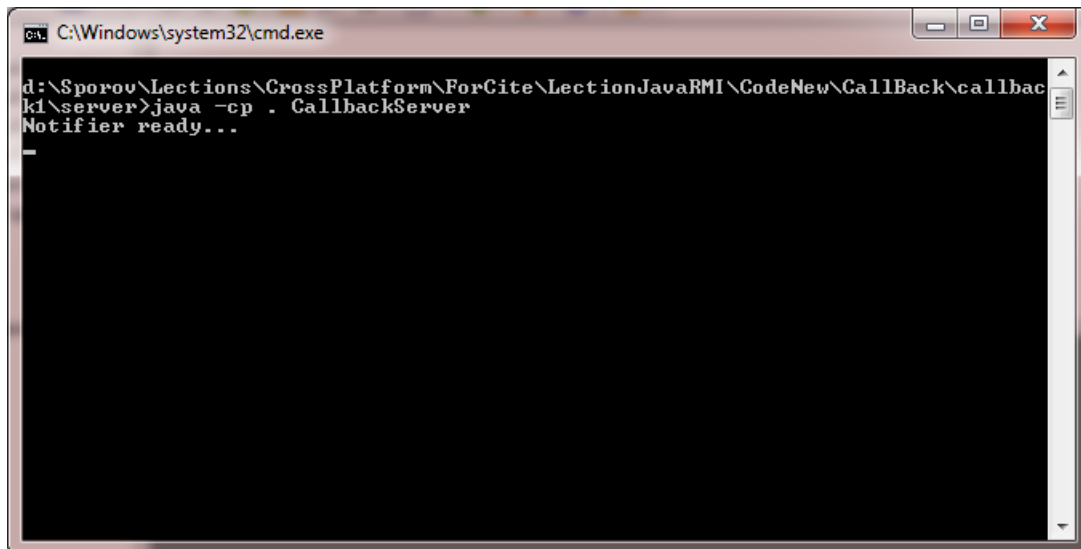
Находясь в каждом из каталогов, откомпилируем находящиеся в них классы. Для этого в каждом каталоге выполним команду:

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

После успешной компиляции клиентской и серверной частей приступим к запуску распределенного приложения. Перейдем в каталог `server` и выполним команду запуска серверной части приложения:

```
java -cp . CallbackServer
```

На экран будет выведено терминальное окно с информационным сообщением о готовности сервера к работе:

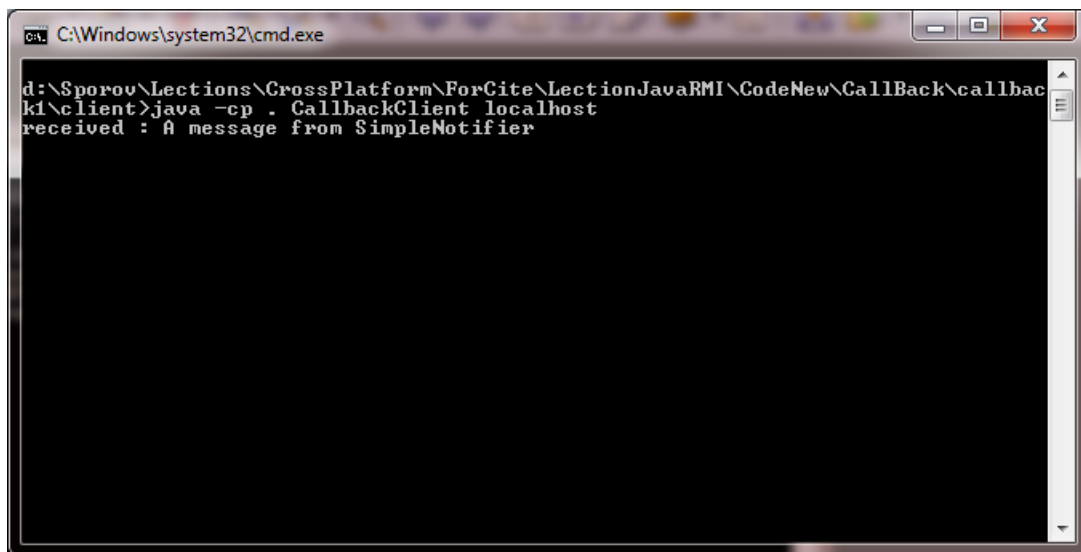


```
C:\Windows\system32\cmd.exe

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\CallBack\callbac
ki\server>java -cp . CallbackServer
Notifier ready...
```

Затем перейдем в каталог client и выполним запуск клиентской части при помощи коданды:

```
java -cp . CallbackClient localhost
```



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

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\CallBack\callbac
ki\client>java -cp . CallbackClient localhost
received : A message from SimpleNotifier
```

Как и планировалось, будет получена удаленная ссылка на объект относящийся к типу Notifier, будет создан объект, реализующий интерфейс Receiver и автоматически экспортирован при создании и зарегистрирован на сервере. Во время регистрации клиенту будет отправлено текстовое сообщение, которое и будет выведено в терминальном окне.

Кроме того, при регистрации на сервере, будет выведена ссылка на объект-получатель и отосланное сообщение:

```

C:\Windows\system32\cmd.exe
d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\CallBack\callbac
ki\server>java -cp . CallbackServer
Notifier ready...
registered the receiver : Proxy[Receiver,RemoteObjectInvocationHandler[UnicastRe
f [liveRef: [endpoint:[192.168.205.53:51367]<remote>,objID:[a3a5c79:171b1a63ec2:
-7ffe, 78198880755429712131111]
Sent : A message from SimpleNotifier

```

Приложение завершает работу внешним образом (требуется закрыть все терминальные окна).

Другое приложение демонстрирующее обратный вызов

Рассмотрим создание более сложного *RMI* распределенного приложения, демонстрирующего применение обратного вызова.

Приложение будет состоять из сервера и одного или нескольких клиентов. Сервер будет передавать всем зарегистрированным на нем клиентам счет некоторого спортивного матча (похожее по смыслу приложение мы рассматривали, когда изучали многоадресные сокеты). Клиенты могут регистрироваться на сервере, чтобы получить счет игры, и могут отменять свою регистрацию. Для реализации этой идеи воспользуемся новыми версиями интерфейсов *Notifier* и *Receiver*:

```
package callback;
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface Receiver extends Remote {
    public void notify(String s) throws RemoteException;
}
```

```
package callback;
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface Notifier extends Remote {
    public void register(Receiver r) throws RemoteException;
```

```

    public void cancel(Receiver r) throws RemoteException;
}

```

Рассмотрим серверную сторону задачи. Класс, реализующий интерфейс Notifier, может иметь такой вид:

```

package callback;

import java.rmi.RemoteException;
import java.util.Random;
import java.util.concurrent.CopyOnWriteArrayList;

public class ScoreNotifier implements
    Notifier, Runnable {

    private CopyOnWriteArrayList<Receiver> list;

    public ScoreNotifier() {
        list = new CopyOnWriteArrayList<Receiver>();
        new Thread(this).start();
    }

    @Override
    public void register(Receiver r) throws RemoteException {
        System.out.println("Attempt to register");
        if (!list.contains(r)) {
            list.add(r);
        }
        System.out.println("Registered the receiver");
    }

    @Override
    public void cancel(Receiver r) throws RemoteException {
        System.out.println("Try to cancel the registration");
        if (list.contains(r)) {
            list.remove(r);
        }
        System.out.println("Cancelled the registration");
    }

    @Override
    public void run() {
        Random rand = new Random();
    }
}

```



```

int score = 0, add;
while (true) {
    do {
        try {
            Thread.sleep(1000 + rand.nextInt(1000));
        } catch (Exception e) {
            e.printStackTrace();
        }
    } while ((add = rand.nextInt(9)) == 0);
    score += add;
    System.out.println("Current score: " + score);
    for (Receiver r : list) {
        new Sender(r, score).start();
    }
}
}
}

```

Рассмотрим вспомогательный класс Sender. В нашем примере он будет обычным классом, но, вообще-то, этот класс лучше сделать закрытым внутренним классом предыдущего класса (ScoreNotifier).

```
package callback;
```

```

public class Sender extends Thread {
    Receiver receiver;
    int score;

    Sender(Receiver r, int s) {
        this.receiver = r;
        this.score = s;
    }

    public void run() {
        try {
            receiver.notify(String.valueOf(score));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Рассмотренные два первых класса нашего приложения генерируют искусственный счет и уведомляют об этом всех зарегистрированных клиентов. Серверная часть приложения имеет типичную структуру:

```
package callback;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ScoreNotifierServer {
    public static void main(String[] args) {
        String name = "notifier";
        int port = 6789;
        ScoreNotifier notifier = new ScoreNotifier();
        try {
            Notifier stub =
                (Notifier) UnicastRemoteObject.exportObject(notifier, 0);
            Registry registry = LocateRegistry.createRegistry(port);
            registry.rebind(name, stub);
            System.out.println("Notifier ready...");
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Теперь реализуем клиентскую часть приложения. Приведем возможный исходный код класса, который имплементирует интерфейс Receiver:

```
package callback;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ScoreReceiver implements
    Receiver, java.io.Serializable {

    public Receiver export() throws RemoteException {
        return (Receiver) UnicastRemoteObject.exportObject(this, 0);
    }

    @Override
```

```

    public void notify(String s) throws RemoteException {
        System.out.println("Received: " + s);
    }
}

```

Теперь приведем исходный код класса, представляющего клиента нашей распределенной системы:

```

package callback;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class ScoreReceiverClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 6789;
        String url = "rmi://" + host + ":" + port + "/notifier";
        try {
            Notifier notifier = (Notifier) Naming.lookup(url);
            ScoreReceiver receiver =
                new ScoreReceiver();
            Receiver stub = receiver.export();
            notifier.register(stub);
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Try to unregister...");
            notifier.cancel(stub);
            System.exit(0);
        } catch (NotBoundException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

Данное распределенное приложение может быть запущено прямо из среды разработки (при этом реестр удаленных объектов программно создается при выполнении кода сервера). Но, для моделирования ситуации, когда отдельная часть приложения (серверная, клиентская) запускаются на отдельном компьютере, разнесем класс-файлы приложения по каталогам:

Каталог server:

```
callback
    Notifier.class
    Receiver.class
    ScoreNotifier.class
    Sender.class
    ScoreNotifierServer.class
```

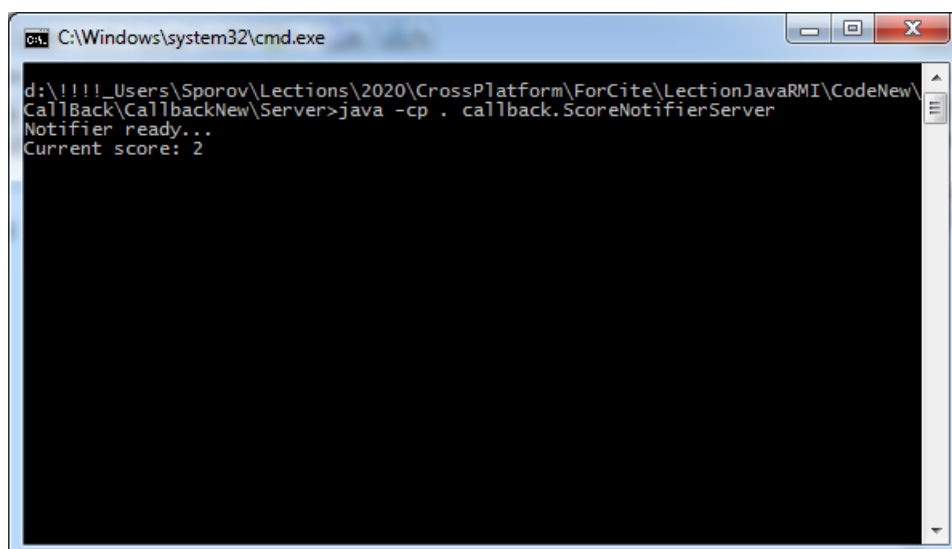
Каталог client:

```
callback
    Notifier.class
    Receiver.class
    ScoreReceiver.class
    ScoreReceiverClient.class
```

Для запуска серверной части приложения перейдем в каталог server и оттуда в терминальном окне выполним команду:

```
java -cp . callback.ScoreNotifierServer
```

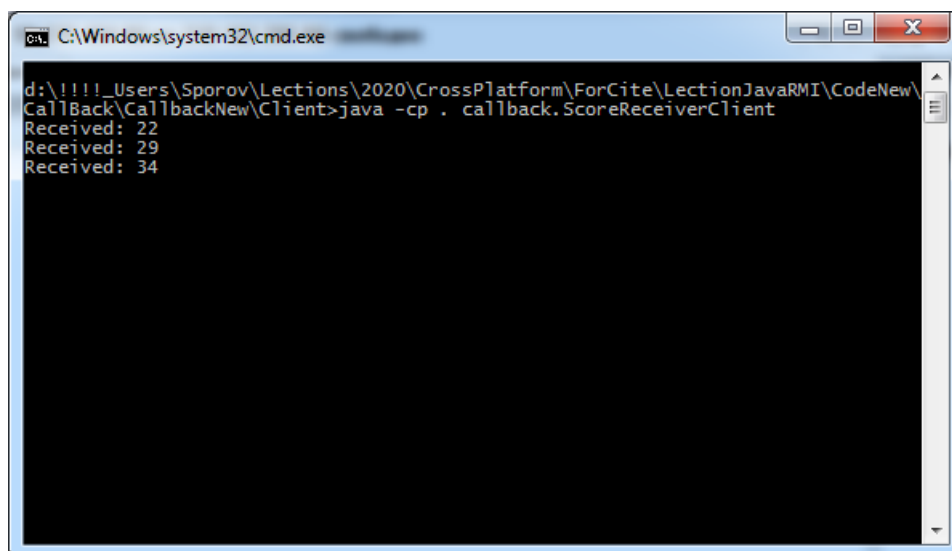
В результате будет запущена серверная часть приложения. На экран будем выведено терминальное окно с информационными сообщениями:



Затем перейдем в каталог client и оттуда в терминальном окне выполним команду:

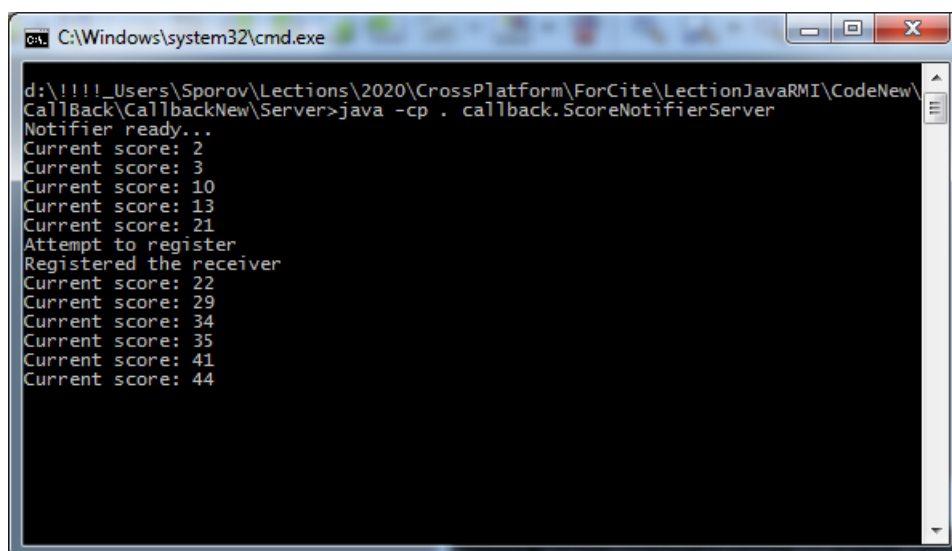
```
java -cp . callback.ScoreReceiverClient
```

Будет загружено терминальное окно клиента, в котором клиент, после регистрации на сервере, будет выводить принятый от серверной стороны счет игра:



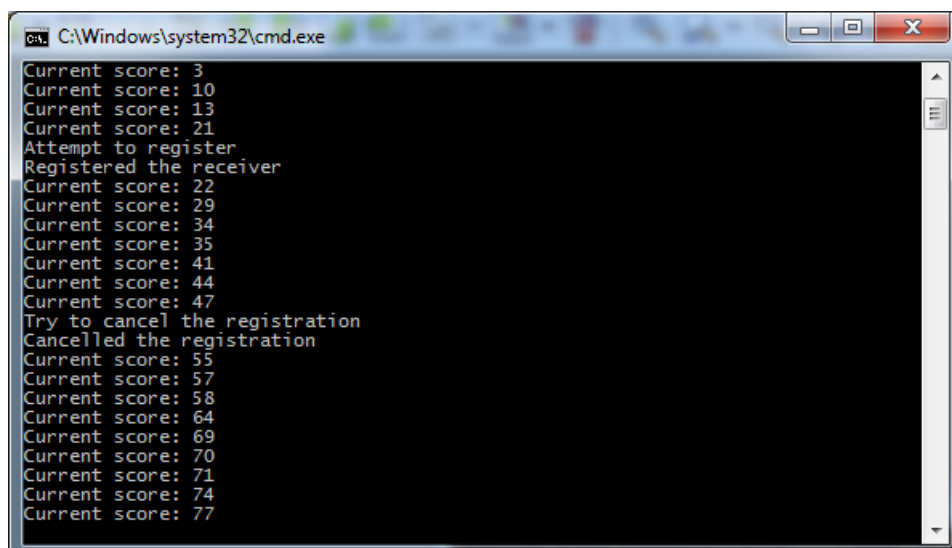
```
C:\Windows\system32\cmd.exe
d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
CallBack\CallbackNew\Client>java -cp . callback.ScoreReceiverClient
Received: 22
Received: 29
Received: 34
```

В терминальном окне на серверной стороне будет отражен факт регистрации клиента:



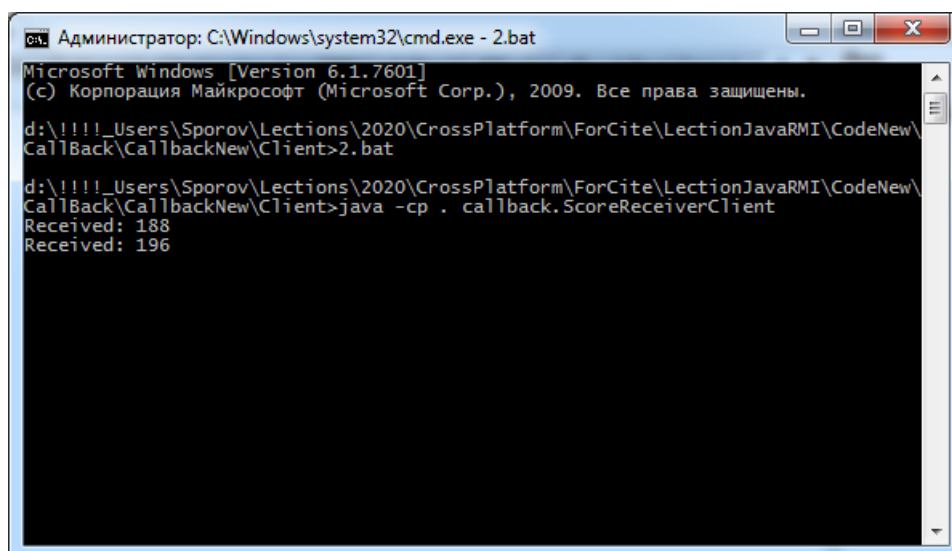
```
C:\Windows\system32\cmd.exe
d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
CallBack\CallbackNew\Server>java -cp . callback.ScoreNotifierServer
Notifier ready...
Current score: 2
Current score: 3
Current score: 10
Current score: 13
Current score: 21
Attempt to register
Registered the receiver
Current score: 22
Current score: 29
Current score: 34
Current score: 35
Current score: 41
Current score: 44
```

После завершения работы клиента (после 10 сек работы клиент отсоединиться от сервера и завершит работу). Информация об этом будет выведена в информационное терминальное окно сервера:



```
C:\Windows\system32\cmd.exe
Current score: 3
Current score: 10
Current score: 13
Current score: 21
Attempt to register
Registered the receiver
Current score: 22
Current score: 29
Current score: 34
Current score: 35
Current score: 41
Current score: 44
Current score: 47
Try to cancel the registration
Cancelled the registration
Current score: 55
Current score: 57
Current score: 58
Current score: 64
Current score: 69
Current score: 70
Current score: 71
Current score: 74
Current score: 77
```

Запустим еще одного клиента (его можно запустить во время работы первого клиента):

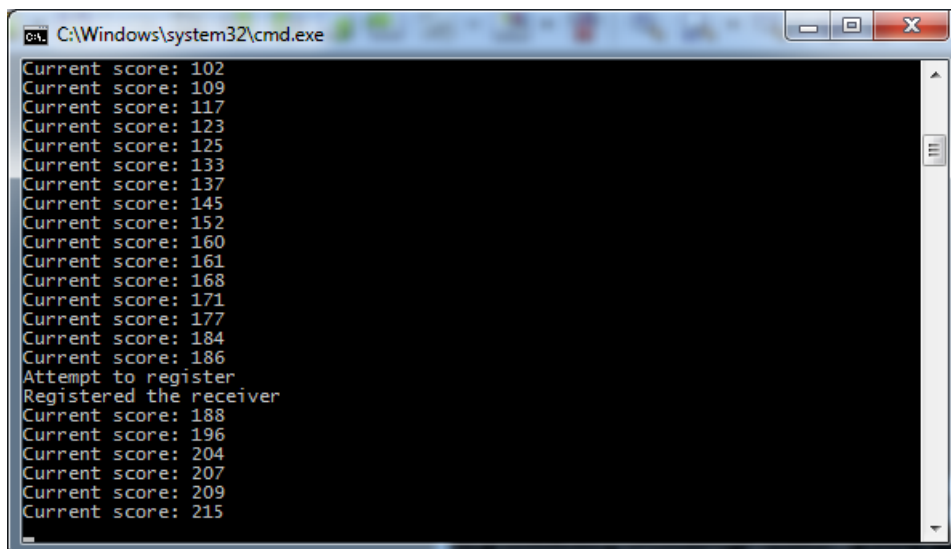


```
Администратор: C:\Windows\system32\cmd.exe - 2.bat
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectureJavaRMI\CodeNew\
Callback\CallbackNew\Client>2.bat

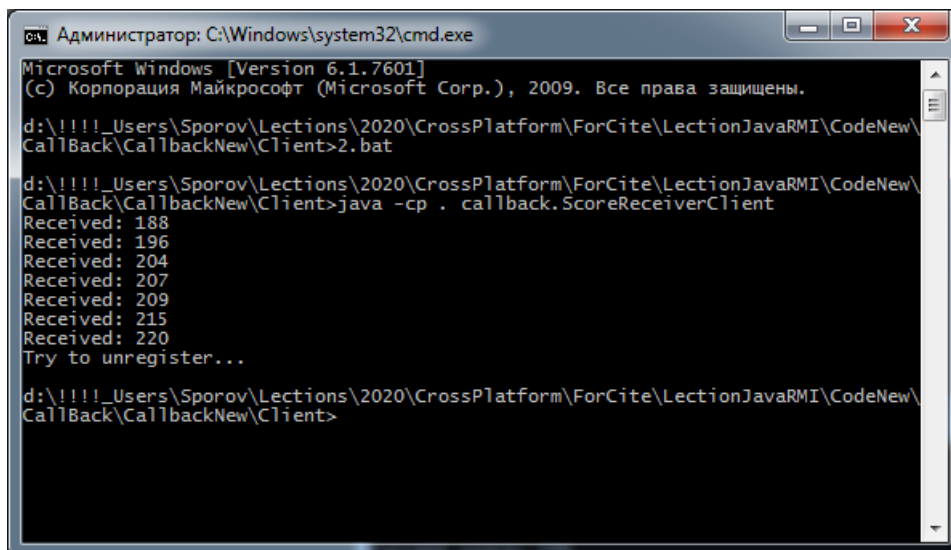
d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectureJavaRMI\CodeNew\
Callback\CallbackNew\Client>java -cp . callback.ScoreReceiverClient
Received: 188
Received: 196
```

Опять, информация о регистрации клиента отображается в терминальном окне сервера:



```
C:\Windows\system32\cmd.exe
Current score: 102
Current score: 109
Current score: 117
Current score: 123
Current score: 125
Current score: 133
Current score: 137
Current score: 145
Current score: 152
Current score: 160
Current score: 161
Current score: 168
Current score: 171
Current score: 177
Current score: 184
Current score: 186
Attempt to register
Registered the receiver
Current score: 188
Current score: 196
Current score: 204
Current score: 207
Current score: 209
Current score: 215
```

После заданного времени работы клиент также отменяет регистрацию на сервере и завершается:



```
Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
Ca1lBack\Ca1lbackNew\Client>2.bat

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
Ca1lBack\Ca1lbackNew\Client>java -cp . callback.ScoreReceiverClient
Received: 188
Received: 196
Received: 204
Received: 207
Received: 209
Received: 215
Received: 220
Try to unregister...

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
Ca1lBack\Ca1lbackNew\Client>
```

Информация об этом также отображается в терминальном окне серверной стороны:

```

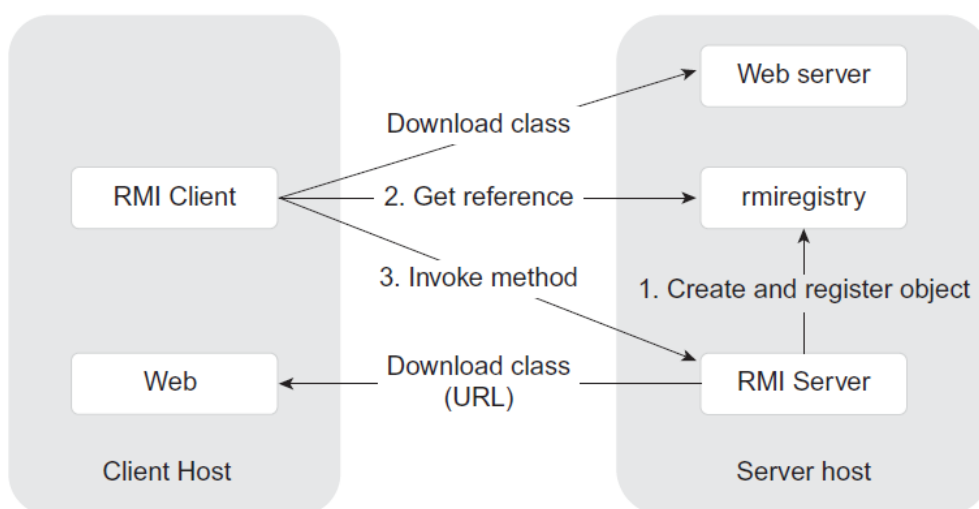
C:\Windows\system32\cmd.exe
Current score: 160
Current score: 161
Current score: 168
Current score: 171
Current score: 177
Current score: 184
Current score: 186
Attempt to register
Registered the receiver
Current score: 188
Current score: 196
Current score: 204
Current score: 207
Current score: 209
Current score: 215
Current score: 220
Try to cancel the registration
Cancelled the registration
Current score: 222
Current score: 224
Current score: 228
Current score: 229
Current score: 234
Current score: 238

```

Пока работу серверной части приложения завершаем внешним образом – просто закрывая терминальное окно (или средствами интегрированной среды разработки, если приложение запускается в ней). Чуть позже мы рассмотрим, как можно корректно завершить работу *RMI* приложения

Динамическая загрузка классов

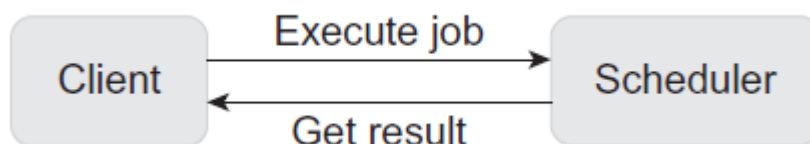
Одной из важных особенностей технологии *Java RMI* является возможность динамически загружать определение класса объекта, если оно не доступно в *JVM* получателя. Рассмотрим рисунок, на котором показано, как классы загружаются с клиента на сервер и с сервера на клиент с использованием *URL* протокола обмена данными.



Таким образом, можно передать определение класса объекта на другую, возможно удаленную, виртуальную машину *Java*, и технология *RMI*, при необходимости, может передать объекты совместно с их фактическими классами. Эта возможность позволяет передавать новые типы и новые функциональные возможности в удаленную *JVM* во время работы распределенного приложения, таким образом динамически изменяя поведение *RMI* приложения.

Пример

Для демонстрации возможности динамической загрузки классов разработаем достаточно сложное распределенное приложение, использующее технологию *RMI*. Более простой аналог такого приложения мы рассматривали при изучении потоковых сокетов.



В этом приложении серверная часть создаст удаленный объект, который назовем *scheduler* (планировщик). Этот объект будет принимать задания от клиентов, локально выполнять эти задания, вычислять время их выполнения и возвращать клиенту, предоставившему это задание, результат, а также значение расчетного времени. Таким образом, клиенты смогут выполнять свои задания удаленно или на более мощном компьютере, или на компьютере, имеющем специализированное оборудование, или на компьютере, имеющем специальное разрешение.

Следует обратить внимание на то, что *scheduler* (планировщик) заранее не должен знать те задания, которые он будет выполнять. По мере необходимости клиенты могут создавать свои собственные задания и отправлять их объекту *scheduler* (планировщику) на выполнение. Понятно, что в таком случае будет необходимо наложить единственное ограничение на задание, которое заключается в том, что класс задания должен реализовывать интерфейс, определенный планировщиком. Класс-файл определения задания можно загрузить с использованием технологии *RMI* во время выполнения распределенного приложения с клиента, который отправляет задание на сервер. Как только класс-файл задания станет доступен, *scheduler* (планировщик) сможет локально выполнить задание.

Есть еще одна проблема: а как же планировщик сможет вернуть результат клиенту? Для разных задач могут потребоваться разные классы для создания объекта результата. Если класс результата является встроенным, то у клиента определение класса уже есть и его можно легко использовать. Если же тип результата является пользовательским типом, определенным планировщиком, то его нужно как-то передать клиенту. Клиент точно так же, используя возможности динамической загрузки классов, может загрузить определение класса результата с сервера и получить объект-результат. И опять будет единственное требование к объекту результата - его класс должен имплементировать интерфейс, известный клиенту.

Таким образом, вследствие реализации такой схемы серверный объект - планировщик сможет выполнять произвольные задания предварительного наличия определения класса задания. Клиенты также смогут получать объект-результат без предварительного наличия определения класса результата. Среда выполнения *Java RMI* будет по мере необходимости загружать

необходимые файлы классов из определенного при запуске приложения места. Таким образом, с помощью технологии *Java RMI* можно изменять поведение объекта и динамически устанавливать его на удаленной машине.

Создание *RMI* сервера

В серверной части приложения можно выделить три части: интерфейс планировщика, реализация этого интерфейса и часть, которая создает объект планировщика *scheduler* и подготавливает его к работе.

Создание интерфейса

Создадим интерфейс *Scheduler*, который задает правила взаимодействия между сервером и клиентом, а также определяет то, как клиент видит удаленный объект.

```
package intf;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Scheduler extends Remote {
    Result execute(Task task) throws RemoteException;
}
```

В интерфейсе определен единственный метод *execute()*, который будет использоваться клиентами для отправки своих заданий на сервер. В этом интерфейсе используются два других интерфейса: интерфейс *Task* и интерфейс *Result*. Интерфейс *Task* является локальным (не удаленным) интерфейсом и определяет структуру задания, которое будет отправляться клиентами на сервер. Приведем возможный исходный код интерфейса:

```
package intf;

public interface Task {
    Object solve();
}
```

В интерфейсе *Task* определен единственный метод *solve()*, который фактически выполняет задание и возвращает результат в виде объекта типа *Object*. Это значение впоследствии будет возвращено клиенту, который отправил задание на сервер, а этот клиент знает, ответ какого типа должен быть возвращен и может выполнить корректное преобразование типа. Класс каждого конкретного задания, которое будет отправлено клиентом на сервер, должен имплементировать этот интерфейс и определить метод *solve()*. Ну а серверный удаленный объект-планировщик (с типом *Scheduler*) должен

загрузить это определение класса, чтобы иметь возможность выполнить задание локально.

Интерфейс Result также является локальным интерфейсом и определяет структуру результата, который должен быть возвращен серверным удаленным объектом - планировщиком (типа Scheduler) тому клиенту, который отправил задание. Приведем возможный исходный код интерфейса Result.

```
package intf;

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

В интерфейсе определено два метода, output() и computeTime(), которые возвращают результат и время выполнения задания, соответственно. Класс для объекта, представляющего результат выполнения задания, должен имплементировать этот интерфейс, ну а клиент, отправивший задание на сервер, должен загрузить определение класса для этого конкретного объекта Result, чтобы иметь возможность получить результат.

Мы уже обсуждали, что для передачи объектов в рамках технологии *Java RMI* используется механизм бинарной сериализации (*object serialization*), поэтому классы, реализующие интерфейсы Task и Result, должны также имплементировать и интерфейс java.io.Serializable.

Реализация интерфейсов

На серверной стороне нашего распределенного приложения должно быть определено два класса: SchedulerImpl и ResultImpl, которые реализуют удаленный интерфейс Scheduler и локальный интерфейс Result, соответственно. Приведем возможный исходный код для класса ResultImpl.

```
package impl;

import java.io.Serializable;
import intf.Result;

public class ResultImpl implements Result, Serializable {
    Object output;
    double computeTime;

    public ResultImpl(Object o, double c) {
        output = o;
        computeTime = c;
    }
}
```

```

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

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

```

Объект этого класса представляет результат выполнения задания, включая в себя, собственно, результат вычисления и время, которое было потрачено на выполнение этого задания. Объект-результат этого класса будет создан на серверной стороне удаленным объектом типа `SchedulerImpl` после выполнения задания и будет передан на сторону клиента, который его отправил на сервер. Таким образом, этот объект нужно передать на другую сторону взаимодействия при помощи механизма сериализации, и поэтому его класс реализует интерфейс `Serializable`. Определение класса `ResultImpl` должно быть загружено по сети на клиентскую сторону для обеспечения возможности принять и восстановить переданный с сервера объект-результат.

Приведем возможный исходный код для класса `SchedulerImpl`.

```

package impl;

import java.rmi.RemoteException;
import intf.*;

public class SchedulerImpl implements Scheduler {
    public SchedulerImpl() {
        super();
    }

    @Override
    public Result execute(Task task) throws RemoteException {
        double startTime = System.nanoTime();
        Object output = task.solve();
        double endTime = System.nanoTime();
        return new ResultImpl(output, endTime-startTime);
    }
}

```

При создании класса `SchedulerImpl` следует реализовать метод `execute()`, в котором собственно и выполняется переданное от клиента задание. При этом конкретный класс для объекта задания определяется на стороне клиента. В результате объект-планировщик не имеет никакого представления о смысле задания; он просто для объекта задания вызывает метод `solve()`. Понятно, что сервер должен загрузить определение класса для объекта задания с клиентской стороны перед вызовом метода `solve()` (даже перед восстановлением объекта). Затем вычисляется расчетное время, создается объект результата и возвращается обратно клиенту. Следует обратить внимание на наличие в классе конструктора. При таком определении класса конструктор необязателен, но если бы класс был объявлен наследником класса `UnicastRemoteObject`, то конструктор, даже пустой, выбрасывающий исключение `RemoteException`, был бы необходим.

Реализация сервера

Сервер построен по стандартной структуре. Вначале обычным образом создается объект класса `SchedulerImpl`:

```
SchedulerImpl scheduler = new SchedulerImpl();
```

Затем, с помощью приведенного ниже фрагмента кода, этот объект экспортируется в среду выполнения *RMI* (*RMI runtime*), чтобы он мог принимать удаленные вызовы метода:

```
Scheduler stub =  
    (Scheduler) UnicastRemoteObject.exportObject(scheduler, 0);
```

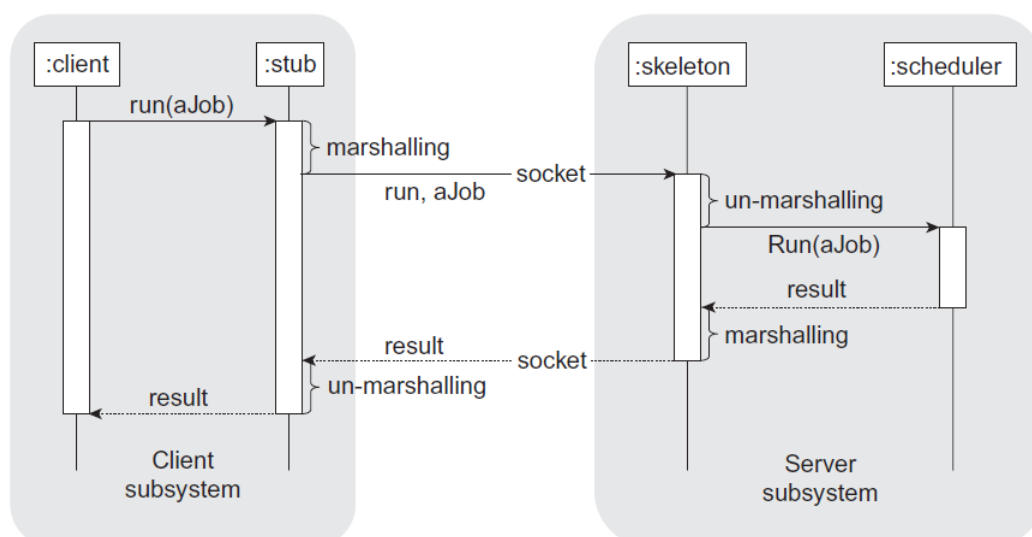
Затем, для того, чтобы зарегистрировать эту заглушку (`stub`) в службе реестра *RMI*, нужно получить ссылку на работающий реестр объектов.

```
Registry registry = LocateRegistry.getRegistry();
```

Ну и затем можно стандартным образом зарегистрировать эту заглушку (`stub`) в службе реестра *RMI*:

```
String name = "Scheduler";  
registry.rebind(name, stub);
```

Вся эта последовательность выполнения показана на рисунке, а весь приведенный кусок кода должен быть размещен в блок `try-catch`.



Поскольку файлы классов заданий, реализующих интерфейс `Task`, будут во время работы удаленного объекта типа `SchedulerImpl` загружены из другой *JVM*, то необходимо установить менеджер безопасности (*security manager*), который будет защищать и регулировать доступ к системным ресурсам для этого загруженного кода. Если код, переданный в данную *JVM* в результате загрузки по сети, выполняет какую-либо небезопасную операцию, то менеджер безопасности проверит, имеет ли загруженный код право выполнять эту операцию, и предпримет необходимые действия. Если не устанавливать менеджер безопасности, то возможность динамической загрузки кода по каналам *RMI* не будет включена. Приведем пример кода для установки менеджера безопасности:

```

if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}

```

Если установлен менеджер безопасности, то необходимо указать политику безопасности (*security policy file* см. <https://docs.oracle.com/javase/tutorial/security/tour1/index.html>, https://docs.oracle.com/cd/E12839_01/core.111/e10043/introjps.htm#JISEC1801), которая будет использоваться менеджером безопасности. Мы определим файл политики безопасности при запуске распределенного приложения. Исходный код для серверной части нашего распределенного приложения может быть таким:

```

package impl;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import intf.*;

```

```

import impl.*;

public class Server {
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        try {
            SchedulerImpl scheduler = new SchedulerImpl();
            Scheduler stub = (Scheduler)
                UnicastRemoteObject.exportObject(scheduler, 0);
            Registry registry = LocateRegistry.getRegistry();
            String name = "Scheduler";
            registry.rebind(name, stub);
            System.out.println("SchedulerImpl object is bound and ready to
work...");
        } catch (Exception e) {
            System.err.println("SchedulerImpl exception:");
            e.printStackTrace();
        }
    }
}

```

Создание клиента

В клиентской части распределенного приложения нужно получить удаленную ссылку на серверный объект Scheduler, сформировать объект задания, отправить его на выполнение на сервер и получить результат вычислений. При этом, именно на стороне клиента нужно определить класс заданий, которые будут выполнены на сервере. В нашем примере определим только одно задания - вычисление факториала числа. Определение класса Factorial (класс задания) может быть таким:

```

package impl;

import intf.*;
import java.io.Serializable;

public class Factorial implements Task, Serializable {
    int n;

    public Factorial(int n) {
        this.n = n;
    }
}

```

```

}

@Override
public Object solve() {
    int result = 1;
    for(int i = 2; i <= n; i++)
        result *= i;
    return Integer.valueOf(result);
}
}

```

Во-первых, класс Factorial должен имплементировать интерфейс Task. Для этого определяем метод solve(), который вычисляет и возвращает факториал указанного целого числа. Когда клиент передает объект задания этого типа удаленному объекту-планировщику, то это выполняется с использованием механизма *сериализации Java*. Таким образом, класс Factorial еще должен имплементировать интерфейс Serializable (или Externalizable).

Для того, чтобы было можно в *JVM* на серверной стороне получить и восстановить объект задания Factorial, необходимо как-то передать на серверную сторону определение класса для объекта задания. Система времени выполнения *RMI* от имени объекта Scheduler загружает определение класса Factorial на серверную сторону приложения. При этом клиентская сторона соединения должна предоставить *RMI* системе место, где она может найти определение класса этого объекта задания. После этого на стороне сервера можно будет без проблем получить объект - задание, вызывать метод execute() объекта SchedulerImpl, который, в свою очередь, вызывает метод solve() объекта Factorial.

Кроме этого, клиенты могут отправлять и другие задания удаленному объекту - планировщик на серверной стороне. Планировщик выполняет эти задания, используя описанную выше процедуру. Планировщику нужно только знать, что каждое полученное задание реализует интерфейс Task и содержит метод solve(), а также знать, откуда система времени выполнения *RMI* предоставить класс-файл задания.

Разберем основные моменты кода клиента. Сначала клиент должен получить ссылку на удаленный реестр объектов с помощью метода LocateRegistry.getRegistry():

```
Registry registry = LocateRegistry.getRegistry(args[0]);
```

Метод getRegistry() в качестве аргумента получает первый параметр args[0], указанный в командной строке при запуске приложения, который представляет собой *имя* или *IP-адрес* компьютера, на котором работает реестр на порте по умолчанию (1099). Если реестр работает на порте, отличном от

1099, то нужно указать этот номер порта в качестве второго аргумента метода `getRegistry()`.

Затем с помощью метода `lookup()`, вызванном на этой найденной ссылке на реестр, клиент получает ссылку на удаленный объект с именем `Scheduler` следующим образом:

```
String name = "Scheduler";
Scheduler scheduler = (Scheduler) registry.lookup(name);
```

Далее, создаем объект-задачу, создавая экземпляр объекта `Factorial`.

```
Factorial task = new Factorial(Integer.parseInt(args[1]));
```

Второй аргумент, указанный в командной строке при вызове приложения, `args[1]` после преобразования его в целое число передается конструктору `Factorial`. Этот аргумент определяет целое число, факториал которого должен быть вычислен. После этого клиент может отправить задание на удаленный объект для выполнения.

```
Result res = scheduler.execute(task);
```

Результат выполнения задания сохраняется в объекте класса, реализующего интерфейс `Result`. Этот класс определен на серверной стороне приложения, и у клиента перед работой нет определения класса для объекта `Result`. Система *RMI* загрузит требуемое определение класса из того места, которое указано сервером. Таким образом, получив определение класса, клиент может работать с объектом `Result`, чтобы получить вычисленный результат и время, затраченное на выполнение.

```
System.out.println(args[1] + "! = " + (Integer)res.output());
System.out.println("Execution time = " + res.computeTime() +
    " microsec(s)");
```

Указанные строки кода должны быть записаны в блоке `try-catch` для обработки тех ошибок, которые могут возникать во время выполнения. Для защиты системы должен быть установлен менеджер безопасности, поскольку определение класса для объекта `Result` загружается в *JVM* клиента.

Приведем возможный код клиентской части приложения:

```
package impl;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import intf.*;
```

```

import impl.*;

public class Client {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        try {
            Registry registry = LocateRegistry.getRegistry(args[0]);
            String name = "Scheduler";
            Scheduler scheduler = (Scheduler) registry.lookup(name);
            Factorial task = new Factorial(Integer.parseInt(args[1]));
            Result res = scheduler.execute(task);
            System.out.println(args[1] + "! = " + (Integer)res.output());
            System.out.println("Execution time = " + res.computeTime() +
                               " microsec(s)");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Развертывание, компиляция и запуск приложения

На практике нужно развернуть удаленное приложение, предоставив серверной и клиентской сторонам необходимые классы и интерфейсы. Давайте предположим следующий сценарий развертывания приложения. Разместим класс-файлы по разным папкам, моделируя расположение классов на разных компьютерах. Если в локальной подсети есть несколько машин (до 5 шт.), то разные части можно запустить на разных компьютерах.

Компиляция и распространение интерфейсов

Для работы и клиентской, и серверной частей приложения необходимы класс-файлы интерфейсов. Предположим, что представители разработчиков сервера и клиентов собрались на конференцию по согласованию интерфейсов. В результате в подкаталоге `InterfHost` каталога `MeetingHosts` размещен каталог пакета `intf`, содержащий исходные файлы интерфейсов.

MeetingHosts

InterfHost

intf

Result.java

Scheduler.java

Task.java

Перейдем в каталог `InterfHost` и выполним команду компиляции исходных файлов:

```
javac -cp . intf\*.java
```

При этом следует убедиться, что каталог, содержащий компилятор *Java*, включен в переменную операционной системы `PATH`. После успешной компиляции исходных файлов интерфейсов нужно объединить класс-файлы в *Jar* – архив. Для этого из этого же каталога `InterfHost` выполнить команду объединения **`*.class`** файлов в *Jar* – архив.

```
jar cvf schedulerIntf.jar intf\*.class
```

На экран будут выведены диагностические сообщения:

```
added manifest
adding: intf/Result.class(in = 166) (out= 137)(deflated 17%)
adding: intf/Scheduler.class(in = 227) (out= 171)(deflated 24%)
adding: intf/Task.class(in = 133) (out= 109)(deflated 18%)
```

и в каталоге `InterfHost` будет расположен сформированный из **`*.class`** файлов интерфейсов *Jar* – архив: `schedulerIntf.jar`.

Для создания серверной и клиентских частей приложения будет нужен этот *Jar* – архив. Для того, чтобы к нему могли получить доступ все заинтересованные стороны, его нужно разместить в доступном для получения по сети месте. Поэтому, в каталоге `MeetingHosts` создадим каталог `WebHost`, в котором поместим исходный код нашего маленького *Web*-сервера `NanoHTTPD.java`. Как и на прошлом занятии откомпилируем этот исходный файл, в этом каталоге создадим подкаталог `RMI`, а в него поместим файл с *Jar* – архивом `schedulerIntf.jar`.

MeetingHosts

WebHost

RMI

schedulerIntf.jar

NanoHTTPD.java

***.class**

После этого из каталога `WebHost` выполним команду запуска нашего *Web*-сервера:

```
java -cp . NanoHTTPD -P 8090
```

В результате *Web*-сервер будет работать на порту 8090. Если разные части нашего распределенного приложения запускаются с различных компьютеров, то на каждом отдельном компьютере *Web*-сервер может работать на порту по умолчанию.

```

Администратор: C:\Windows\system32\cmd.exe - start_Web.bat

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
DistribAppl2020\MeetingHosts\WebHost>start_Web.bat

d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
DistribAppl2020\MeetingHosts\WebHost>java -cp . NanoHTTPD -P 8090
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togiias
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8090 from "d:\!!!!_Users\Sporov\Lections\2020\CrossPla
tform\ForCite\LectonJavaRMI\CodeNew\DistribAppl2020\MeetingHosts\WebHost\".
Hit Enter to stop.

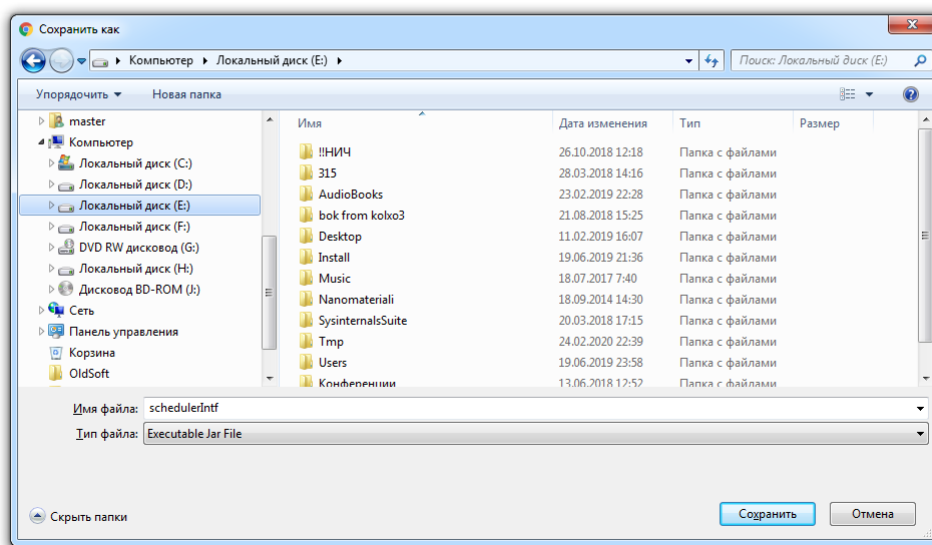
```

Теперь каждая заинтересованная сторона, обратясь по указанному адресу, может получить *Jar* – архив с класс-файлами интерфейсов.



Directory /RMI/

[schedulerIntf.jar](#) (1.11 KB)



Компиляция серверной части

Для создания серверной части приложения в каталоге, содержащем каталог MeetingHosts создадим каталог ServerHosts, представляющий компьютеры серверной стороны. В этом каталоге создадим подкаталог ServerHost, в нем разместим скачанный по сети с сайта конференции по

интерфейсам файл `schedulerIntf.jar`, а также создадим каталог пакета `impl`, содержащий исходные файлы, представляющие серверную сторону приложения. Пакет `impl` содержит три исходных файла: `SchedulerImpl.java`, `ResultImpl.java` и `Server.java`. Напомним, что в файле `SchedulerImpl.java` содержится реализация интерфейса `Scheduler`, в файле `ResultImpl.java` реализация интерфейса `Result`, а файлы `Server.java` содержит серверную программу, где создается и экспортируется объект `Scheduler`.

ServerHosts

ServerHost

```
schedulerIntf.jar
impl
    ResultImpl.java
    SchedulerImpl.java
    Server.java
```

Находясь в каталоге `ServerHost` выполним команду компиляции исходных файлов:

```
javac -cp .;schedulerIntf.jar impl/*.java
```

После успешной компиляции в каталоге пакета `impl` будут содержаться результирующие класс-файлы.

Заглушка и скелетон для `SchedulerImpl` реализуют интерфейс `Scheduler`, который ссылается на интерфейсы `Task` и `Result`. Поэтому для службы *RMI* реестра объектов *Java*, которая будет запущена на этом компьютере, будут нужны определения этих интерфейсов. Они будут получены от работающего *Web*-сервера конференции по интерфейсам. Кроме того, клиентам понадобится определение класса `ResultImpl`, имплементирующего интерфейс `Result`. Поэтому соответствующий класс-файл нужно поместить в доступном для получения по сети месте.

Поэтому, в каталоге `ServerHosts` создадим каталог `WebHost`, в котором поместим исходный код нашего маленького *Web*-сервера `NanoHTTPD.java`. Как и на прошлом занятии откомпилируем этот исходный файл, в этом каталоге создадим подкаталог `RMI`, а в него каталог пакета `impl` с класс-файлом результата `ResultImpl.class`.

ServerHosts

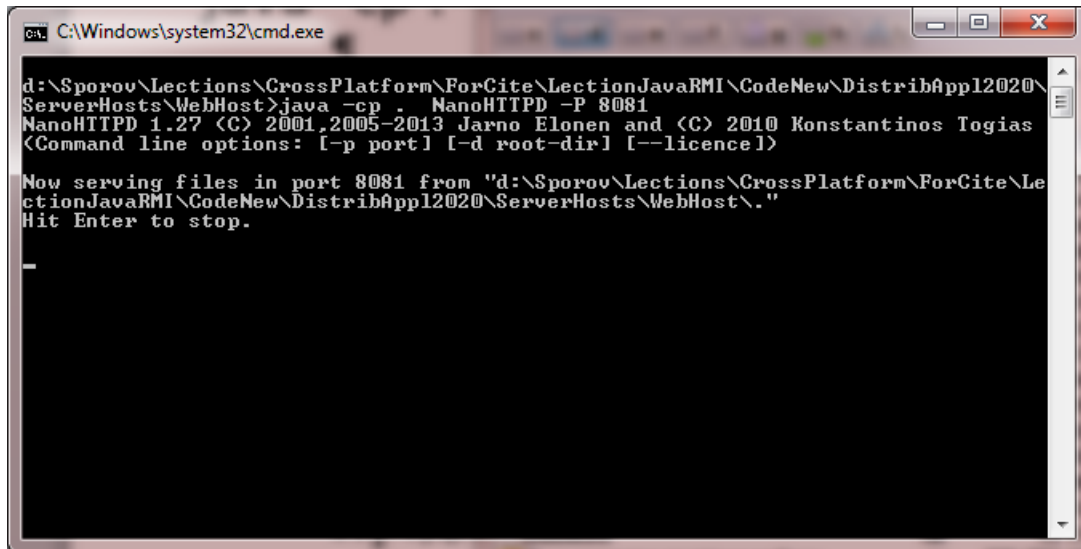
WebHost

```
RMI
    impl
        ResultImpl.class
    NanoHTTPD.java
    *.class
```

После этого, во время запуска приложения, из каталога WebHost будет выполнена команда запуска нашего *Web*-сервера:

```
java -cp . NanoHTTPD -P 8081
```

В результате потом *Web*-сервер будет работать на порту 8081 и по запросу клиента сможет предоставить ему класс-файл необходимый для работы с объектом результата, полученного от сервера.



Компиляция клиентской части

Предположим, что клиентская часть приложения создается на другом компьютере, который в нашей модели развертывания представлен каталогом ClientHosts. Для создания клиентской части приложения в каталоге, содержащем каталог MeetingHosts создадим каталог ClientHosts, представляющий компьютеры серверной стороны. В этом каталоге создадим подкаталог ClientHost, в нем разместим скачанный по сети с сайта конференции по интерфейсам файл schedulerIntf.jar, а также создадим каталог пакета impl, содержащий исходные файлы, представляющие клиентскую сторону приложения. Пакет impl на клиентской стороне содержит два исходных файла: Factorial.java и Client.java. Напомним, что в файле Factorial.java содержится реализация задания, которое будет передано на вычисление на сервер, а файл Client.java содержит клиентскую программу, в которой сначала нужно получить удаленную ссылку на объект Scheduler и затем уже можно с помощью этой ссылки вызвать метод execute().

ClientHosts

ClientHost

```
schedulerIntf.jar
impl
    Client.java
    Factorial.java
```

Находясь в каталоге `ClientHost` выполним команду компиляции исходных файлов:

```
javac -cp .;schedulerIntf.jar impl/*.java
```

Следует напомнить, что серверной части приложения для выполнения своей работы необходим конкретный класс, реализующий интерфейс `Task` (в рассматриваемом примере это класс `Factorial`). В правильно развернутом приложении этот класс должен быть загружен на сервер системой времени выполнения *Java RMI*. Для этого клиентская часть приложения должна разместить этот класс-файл в доступном для получения по сети месте. На клиентской стороне приложения поять будем использовать *Web*-сервер `NanoHttpD`.

Поэтому, в каталоге `ClientHosts` создадим каталог `WebHost`, в котором поместим исходный код нашего маленького *Web*-сервера `NanoHTTPD.java`. Как и на прошлом занятии откомпилируем этот исходный файл, в этом каталоге создадим подкаталог `RMI`, а в него каталог пакета `impl` с класс-файлом задачи `Factorial.class`.

```
ClientHosts
    WebHost
        RMI
            impl
                Factorial.class
            NanoHTTPD.java
            *.class
```

После этого, во время запуска приложения, из каталога `WebHost` будет выполнена команда запуска нашего *Web*-сервера:

```
java -cp . NanoHTTPD -p 8082
```

```

C:\Windows\system32\cmd.exe

d:\Sporov\Lections\CrossPlatform\ForCite\LectioJavaRMI\CodeNew\DistribApp12020\
ClientHosts\WebHost>java -cp . NanoHTTPD -p 8082
NanoHTTPD 1.27 <C> 2001, 2005-2013 Jarno Elonen and <C> 2010 Konstantinos Togias
<Command line options: [-p port] [-d root-dir] [--licence]>

Now serving files in port 8082 from "d:\Sporov\Lections\CrossPlatform\ForCite\Le
ctioJavaRMI\CodeNew\DistribApp12020\ClientHosts\WebHost\."
Hit Enter to stop.

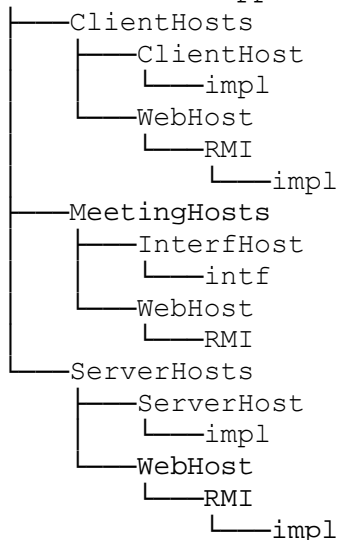
```

Запуск распределенного приложения

Перед запуском еще раз посмотрим на возможную структуру каталогов нашего распределенного приложения:

Directory Structure

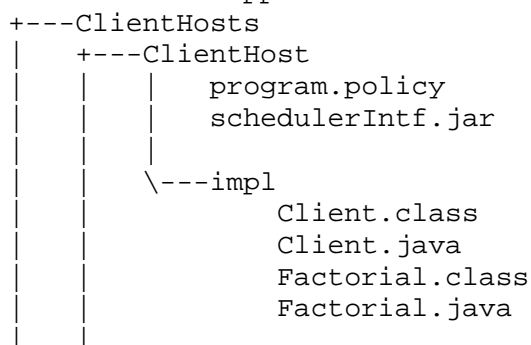
Distributive Application



Кроме того, приведем более полную версию данной схемы, указав не только каталоги, но и расположение самых важных файлов.

File Tree

Distributive Application




```

\---WebHost
|   NanoHTTPD$1.class
|   NanoHTTPD$2.class
|   NanoHTTPD$HTTPSession.class
|   NanoHTTPD$Response.class
|   NanoHTTPD.class
|   NanoHTTPD.java
|
\---RMI
|   \---impl
|       Factorial.class
|
+---MeetingHosts
|   +---InterfHost
|       schedulerIntf.jar
|       \---intf
|           Task.class
|           Task.java
|           Result.class
|           Result.java
|           Scheduler.class
|           Scheduler.java
|       \---WebHost
|           NanoHTTPD$1.class
|           NanoHTTPD$2.class
|           NanoHTTPD$HTTPSession.class
|           NanoHTTPD$Response.class
|           NanoHTTPD.class
|           NanoHTTPD.java
|       \---RMI
|           schedulerIntf.jar
|
\---ServerHosts
|   +---ServerHost
|       program.policy
|       schedulerIntf.jar
|       \---impl
|           ResultImpl.class
|           ResultImpl.java
|           SchedulerImpl.class
|           SchedulerImpl.java
|           Server.class
|           Server.java
|       \---WebHost
|           NanoHTTPD$1.class
|           NanoHTTPD$2.class
|           NanoHTTPD$HTTPSession.class
|           NanoHTTPD$Response.class
|           NanoHTTPD.class
|           NanoHTTPD.java
|       \---RMI
|           \---impl
|               ResultImpl.class

```

При создании и серверной, и клиентской частей приложения (файлы `Client.java` и `Server.java`) на каждой части был установлен менеджер безопасности. Для того, чтобы разрешить динамическую загрузку классов нужно следует определить для них политику безопасности. Более подробно о безопасности можно прочитать в документации (см., например, https://docs.oracle.com/cd/E12839_01/core.1111/e10043/introjps.htm#JISEC1801, <https://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-specTOC.fm.html>), а мы все определим в упрощенном виде. В нашем способе запуска приложения определим политику безопасности в виде файла политики (*policy file*), который укажем как аргумент командной строки при запуске приложения. С правилами создания файлов политики можно познакомиться в документации (см., например, <https://docs.oracle.com/javase/tutorial/security/tour1/index.html>). Опять же, мы запишем файл политики в упрощенном виде – все будет разрешено всем. Для нашего примера этот файл подойдет, но в реальных приложениях такой файл использовать не следует. В каталогах `ServerHost` и `ClientHost` создадим файл с именем `program.policy` и таким содержимым:

```
grant {
    permission java.security.AllPermission;
};
```

Перед началом запуска приложения нужно убедиться, что все *Web*-сервера (*Web*-сервер конференции интерфейсов, *Web*-сервер серверной и *Web*-сервер клиентской стороны) запущены и работают. Если это не так, то нужно перейти в соответствующий каталог и при помощи соответствующей команды запустить соответствующий сервер.

Следующий шаг – запуск службы *RMI* реестра. Для этого нужно перейти в каталог `ServerHosts` и выполнить команду запуска службы реестра:

```
start "rmiregistry" rmiregistry
-J-Djava.rmi.server.useCodebaseOnly=false
```

При таком способе запуска кодовая база будет передана реестру по *RMI* каналу со стороны сервера.

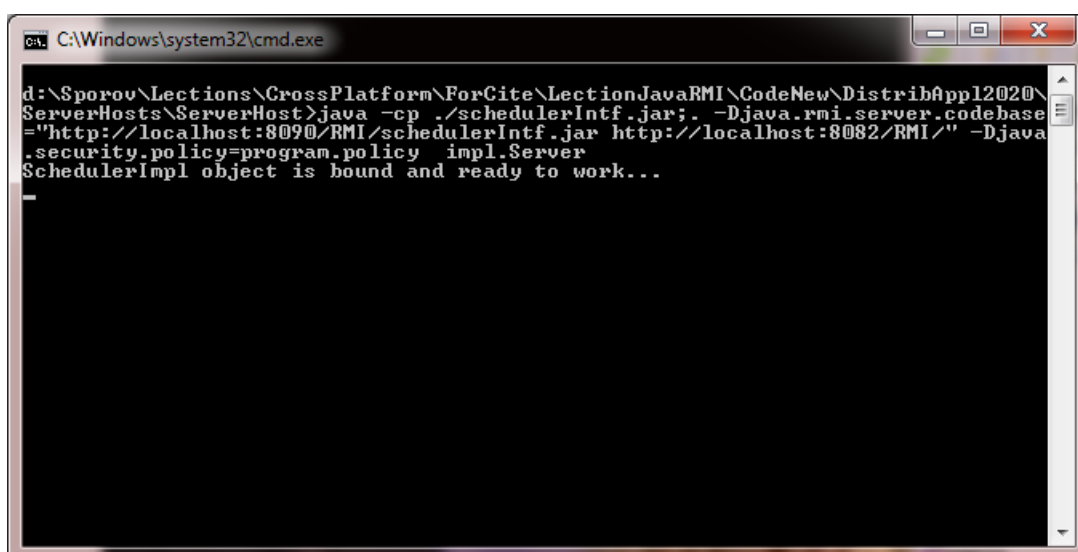


После запуска службы *RMI* реестра можно приступить к запуску сервера. Для этого нужно перейти в каталог `ServerHost` и выполнить команду запуска сервера:

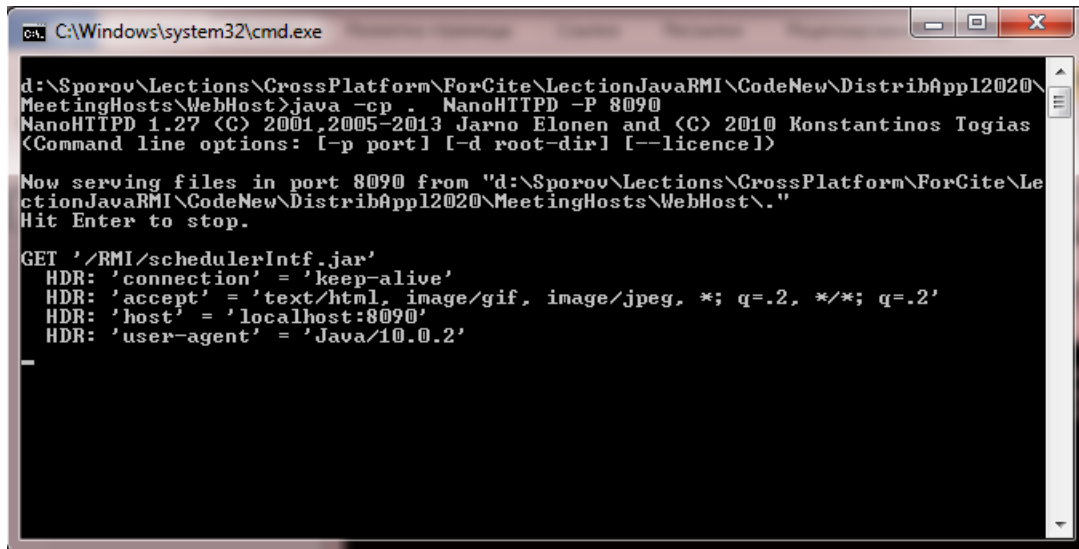
```
java -cp ./schedulerIntf.jar; .
-Djava.rmi.server.codebase="http://localhost:8090/RMI/schedulerIntf.jar
http://localhost:8082/RMI/" -Djava.security.policy=program.policy
impl.Server
```

Следует обратить внимание:

- на указание параметра `CLASSPATH` – это и текущий каталог и *jar*-файл,
- на указание кодовой базы – это доступные с помощью работающий *Web*-серверов конференции и клиента откомпилированные файлы интерфейсов и задания,
- на указание файла политики – это файл в текущем каталоге.



На этом этапе при регистрации заглушки в службе реестра нужный файл интерфейса будет загружен с *Web*-сервера конференции по согласованию интерфейсов:



```

C:\Windows\system32\cmd.exe
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
MeetingHosts\WebHost>java -cp . NanoHTTPD -P 8090
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togias
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8090 from "d:\Sporov\Lections\CrossPlatform\ForCite\Le
ctionJavaRMI\CodeNew\DistribApp12020\MeetingHosts\WebHost\."
Hit Enter to stop.

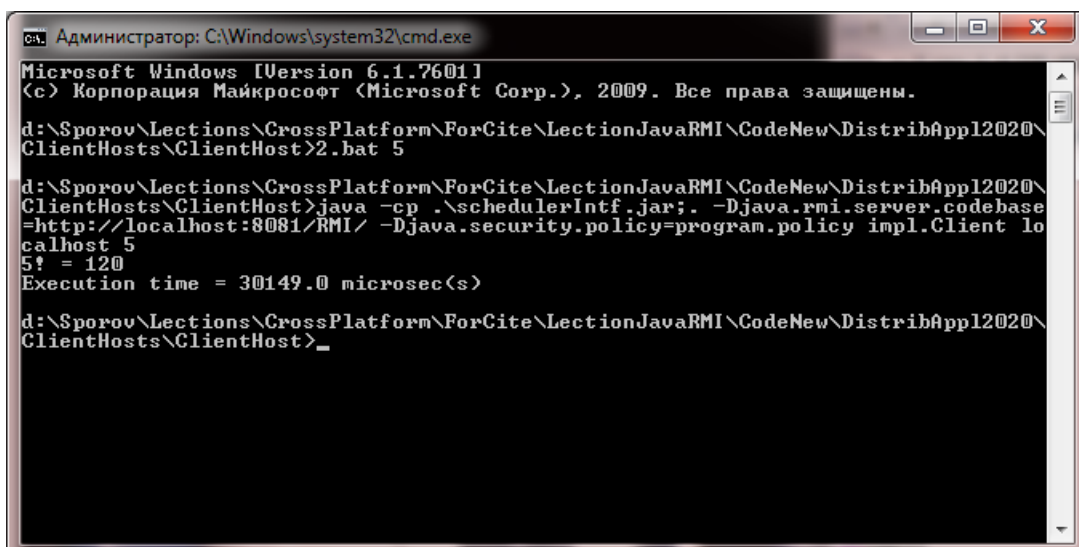
GET '/RMI/schedulerIntf.jar'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/1.0.0.2'
  
```

Сейчас серверный объект полностью установлен в *RMI* готов к приему запросов на вызов удаленных методов от клиентов. Можно начинать запуск клиентской части приложения. Для этого нужно перейти в каталог ClientHost и из него запустить команду запуска клиентской части приложения:

```

java -cp .\schedulerIntf.jar;.
-Djava.rmi.server.codebase=http://localhost:8081/RMI/
-Djava.security.policy=program.policy
impl.Client localhost 5
  
```

На экран будет выведено терминальное окно клиентской части, где будет указан результат выполнения удаленного метода:



```

Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>2.bat 5

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>java -cp .\schedulerIntf.jar;. -Djava.rmi.server.codebase
=http://localhost:8081/RMI/ -Djava.security.policy=program.policy impl.Client lo
calhost 5
5? = 120
Execution time = 30149.0 microsec(s)

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>_
  
```

Выполнение удаленного метода обеспечивают *Web*-серверы, предоставляя необходимые класс-файлы:

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

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
MeetingHosts\WebHost>java -cp . NanoHTTPD -P 8090
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togi
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8090 from "d:\Sporov\Lections\CrossPlatform\ForCite\Le
ctonJavaRMI\CodeNew\DistribApp12020\MeetingHosts\WebHost\."
Hit Enter to stop.

GET '/RMI/schedulerIntf.jar'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/10.0.2'
GET '/RMI/schedulerIntf.jar'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/10.0.2'
```

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

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ServerHosts\WebHost>java -cp . NanoHTTPD -P 8081
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togi
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8081 from "d:\Sporov\Lections\CrossPlatform\ForCite\Le
ctonJavaRMI\CodeNew\DistribApp12020\ServerHosts\WebHost\."
Hit Enter to stop.

GET '/RMI/impl/ResultImpl.class'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8081'
HDR: 'user-agent' = 'Java/10.0.2'
```

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

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\WebHost>java -cp . NanoHTTPD -p 8082
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togi
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8082 from "d:\Sporov\Lections\CrossPlatform\ForCite\Le
ctonJavaRMI\CodeNew\DistribApp12020\ClientHosts\WebHost\."
Hit Enter to stop.

GET '/RMI/impl/Factorial.class'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8082'
HDR: 'user-agent' = 'Java/10.0.2'
```

Можно второй раз запустить клиентскую часть приложения:

```

Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>2.bat 5

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>java -cp .\schedulerIntf.jar;. -Djava.rmi.server.codebase
=http://localhost:8081/RMI/ -Djava.security.policy=program.policy impl.Client lo
calhost 5
5? = 120
Execution time = 30149.0 microsec(s)

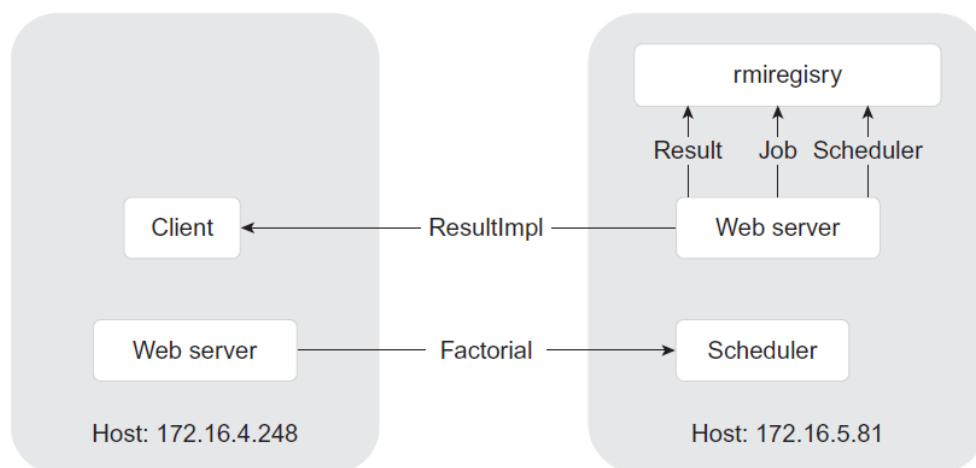
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>2.bat 8

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>java -cp .\schedulerIntf.jar;. -Djava.rmi.server.codebase
=http://localhost:8081/RMI/ -Djava.security.policy=program.policy impl.Client lo
calhost 8
8? = 40320
Execution time = 853.0 microsec(s)

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\DistribApp12020\
ClientHosts\ClientHost>

```

Упрощенную схему работы приложения можно продемонстрировать следующим рисунком:



При запуске приложения нужно обращать внимание на корректную последовательность запуска частей и правильность указания команд (все команды должны быть указаны в одну строку – разбивка по строкам в тексте выполнена только для удобства демонстрации).

Приложение завершается внешним образом – закрываем все открытые консольные окна. Как можно завершить работу *RMI* распределенного приложения разберем на следующем занятии.